Isolation in Public Clouds: Threats, Challenges and Defenses

By

Venkatanathan Varadarajan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 30th November 2015

Dedicated to my brother, parents and the love of my life without whom this would not have been possible.

ISOLATION IN PUBLIC CLOUDS: THREATS, CHALLENGES AND DEFENSES

Venkatanathan Varadarajan

Under the supervision of
Professor Thomas Ristenpart and Professor Michael Swift
At the University of Wisconsin–Madison

Many applications that are in day-to-day use by customers over the Internet are hosted in the public clouds. Public infrastructure-as-a-service clouds, such as Amazon EC2, Google Compute Engine (GCE) and Microsoft Azure allow clients to run virtual machines (VMs) on shared physical infrastructure. This practice of multi-tenancy improves efficiency by multiplexing resources among disparate customers at low costs. Unfortunately, it also introduces the risk of sharing a physical server to run both sensitive customer applications and VMs that may belong to an arbitrary and potentially malicious users. Such a scenario uniquely arises because of multi-tenancy and the openness of public clouds.

The current management infrastructure of these public clouds is driven towards improving performance and efficiency and the security of these systems often takes the back seat in this drive forward. As a result it is unclear what the degree of isolation that these clouds provide against malicious users. In this dissertation, we focus on one of the main security threats to public clouds, *cross-VM attacks*, and evaluate how state-of-the-art cloud infrastructure fares against these attacks. The thesis of this dissertation is that, *"the practice of multi-tenancy in public clouds demands stronger isolation between VMs on a single host in the presence of malicious users."*

Any cross-VM attack involves two steps: *placing* an adversary controlled VM on the same host as one of the victim VMs, and then *breaching* the isolation boundary to either steal sensitive victim information or affect its performance for greed or vandalism. In the first part of the dissertation, we start by evaluating the security of public clouds against cross-VM attacks on two levels: security against VM placement in cluster schedulers and isolation between multi-tenant VMs at the hypervisor. Amidst several challenges such as no known working co-residency detection mechanism and no prior work on analysis of placement policies of public clouds, we show that EC2, Azure and GCE are all vulnerable to adversarial VM placement attacks. In addition, we also investigate the repercussions of performance interference between co-located VMs sharing the same host. Unlike cross-VM side-channels that steal secrets across VM boundaries, we discover that a greedy user could also steal resources and benefit from improved performance at the expense of others. Both these new findings demonstrate that multi-tenancy in public clouds demands stronger isolation.

In the second part of this dissertation, we venture to improve isolation between VMs in the hypervisor. A straightforward solution is *hard isolation* that strictly partitions hardware resources between VMs. However, this comes at the cost of reduced efficiency. We investigate the principle of *soft isolation*: reduce the risk of sharing through better scheduling. We demonstrate this design principle by using it to defend against cross-VM attacks. With extensive experimentation, we show that such a mechanism not only prevent a dangerous cross-VM attacks, but also incurs negligible performance overhead.

In summary, this dissertation shows that cross-VM co-location attacks still remain a problem amidst the deployment of advanced virtualization techniques in modern clouds, hence demands stronger isolation. We demonstrate how this stronger isolation be achieved to defend against a class of cross-VM attack without compromising on efficiency.

Acknowledgments

It should come as no surprise that this dissertation and the associated Ph.D. is a result of a team effort. There are several members in this team who have contributed towards my Ph.D. either knowingly or unknowingly. Here, I record my heartfelt gratitude towards their participation in this endeavor. I also warn you to brace yourself for an emotionally overloaded acknowledgment.

First, my family. I like to start with my elder brother, Dr. Ramakrishnan Varadarajan. Yes, the other and the first doctorate in our the family, who without his knowledge enlightened me about the career path to innovate and push the frontier in a field of study. He also got his Ph.D. in Computer Science (Databases) making it only easier for me to follow him. In fact he got his Post Doctorate at UW-Madison, further raising the bar in the sibling rivalry to new heights! Hence, he remains the root cause of my Ph.D. without whose experience, encouragement and mentorship this would not have been possible. Next, my parents – Krishnan Varadarajan and Suganthy Pichumani. It is impossible to describe the pain and the compromises they had to go through to send the only other son far away for five long years, that too in their sixties. Of course, they never once vocalized their pain to me and always encouraged me to pursue research. I cannot thank them enough for their support. Finally, my close friend and now to-be-wife, Harini Kumar. Sometimes you may not be able to share everything with your family but you can do so with your close friends. Harini was one such friend, who offered encouragement and support to me throughout this endeavor. There were also times she had to endure immeasurable pain to support me in this endeavor. I am forever indebted to her for her support and thank her for her immense trust in me.

My mentors in my undergrad school, Prof. Ranjini Parthasarathy, Prof. Rajeshwari Sridhar, Prof. Bama and Prof. Pandu Rangan, at the right time also encouraged me to explore research at the early stages and sometimes provided platform to publishing research. I thank them for identifying and nurturing my potential to do research.

I cannot thank my advisors, Prof. Michael Swift and Prof. Thomas Ristenpart, enough for giving me this opportunity. If not for Prof. Swift, I would not have had chance to work with Prof. Ristenpart and let alone direct me towards the fascinating area of cloud security. I came in with almost zero experience in research. I am often baffled on how Prof. Swift trusted that I am capable of good quality research in short duration of an independent study. I could not have asked for such a fair, friendly, caring and open-minded advisor. I have lots of anecdotes that demonstrates each of those qualities (mail me if you are interested to know about them). Prof. Ristenpart is another super star advisor. Knowing his name is a sufficient head turner in conferences, let alone being his advisee. There were times when I would go about introducing myself as, "the student of Ristenpart, you know the author of the famous, *Hey you get off my cloud* paper". Being co-advised by both of them was an amazing experience.

Finding an important problem is already a daunting part of a Ph.D., but looking at it from numerous angles and predicting the gravity of the problem is totally a whole another thing. Both of them have been a live demonstration of how to do it. Without their expertise, I would not have made several successful publications in such a relatively short time period of 3 years. I owe it to them. They were immensely patient with me and my mediocre presentation skills. In the process I also successfully inherited several (good) OCDs from them, esp. for paper writing and presentations, and often got commendations for them. One of those good qualities that got etched in me is having the courage to pull the plug on a low quality work and always aiming to work for the overall good of the research community. For instances, while facing a question of working on one strong paper or several low quality papers, they always chose to do the stronger version. I have never seen anyone work on a paper so hard even after it got accepted into a conference by always trying to add value to it. Perfection was their worst enemy. Such good qualities is what you expect to learn from your academic fathers, and I am very glad I got to grow under these two great ones. Without their constant guidance and financial support this dissertation would not have been possible.

Thanks to all my collaborators, Thawan Kooburat, Benjamin Farley, Prof. Yinqian Zhang, and Prof. Ari Juels, who all helped in many of the research endeavors that contributed to this dissertation. I also thank the preliminary exam committee members, Prof. Aditya Akella, Prof. David Wood, Prof. Ari Juel, and my advisors, for all their valuable comments that helped shape this dissertation. I am also thankful for the short-lived system and security group led by my advisors with the weekly group meetings that helped in bouncing off ideas and have fun discussions. I specifically want to thank other graduate students: Sankaralingam Paneerselvam (Sankar), Ramakrishnan Durairajan, Srinath Sridharan, Adam Everspaugh, Vijay Chidambaram, Thanumalyan Sankaralingam, Haris Volos, Zev Weiss, Asim Kadav, Mohit Saxena, Matthew Renzelmann, Matt Fredrikson, Drew Davidson, Liang Wang, and Rahul Chatterjee, for helping with various things research. This includes patiently helping with proof-reading number of my manuscripts, attending practice talks, spending time to brainstorm ideas and expecting nothing in return! Particularly, I want to thank Zev Weiss, and Sankar. Zev Weiss, during the early days of my Ph.D. introduced me to many essential tools like emacs, screen, etc., as I did not have the often assumed technical acumen that is acquired via

working experience or self-learning. In hindsight, this played a critical in increasing my productivity in the later years. It was pleasure working with him on course projects, as I learned something new in every encounter with him. Next is Sankar. I often (literally) turned to Sankar, who's office desk was right behind me, to bounce of ideas and questions on kernel hacking. Because he was the goto guy for Linux kernel related stuff. Particularly, I thank him for being patient with me and my nagging for four full years.

It might come as a surprise to him, but Prof. Remzi Arpaci-Dusseau was like an invisible guardian over my shoulder. There were several instances where the emails he forwarded to all graduates opened up many opportunities both for my internships and full-time job search. I thank him for those amazing opportunities. Further, organizing the Wisconsin Institute on Software-defined Datacenters Of Madison (WISDOM)[1] and workshop along with other professors in the department, brought in even more opportunities specifically for students in my area of interest.

Finally, friends in Madison. Family members and others in India often worry that sending someone to do Ph.D. would eventually make them insane. It is true, and friends are the ones who keep you sane during the Ph.D. experience. During my first year here, my brother and sister-in-law, Vaishnavi Sampath made Madison feel like home. I slowly started growing a network of friends because of whom I got introduced to weight-training, racquetball, badminton, volleyball, and indoor rock-climbing. The frequent get-togethers on weekends with potluck suppers often became part of life here. Such activities helped me take essential breaks away from the monotonous academic pursuits. I would like to these friends: Ramakrishnan Durairajan, Sankaralingam Paneerselvam, Uthra Srinath, Srinath Sridharan, Sibin Philip, Vijay Chidambaram, Thanumalyan Sankaralingam, Sandeep Vishwanathan, Surya Narayanan,

[1]wisdom.cs.wisc.edu

Madhav Venkateswaran, Ragothaman Avanasi, Tyler Harter, Madhu Ramanathan, Ragunanth Venkatesan, Chembian Parthiban, Swarna Chandrasekar and many others. Specifically, I am indebted to Srinath and Uthra for their generosity in many occasions during their time in Madison and beyond. Apart from these friends, I would also like to thank my other fictional F.R.I.E.N.D.S: Chandler, Monica, Ross, Rachel, Joey and Phoebe for filling my initial years of solitude away from home with joy and laughter.

Contents

Abstract **ii**

Acknowledgments **iv**

Contents **ix**

List of Figures and Tables **xiii**

1 Introduction **1**

1.1 Isolation in Modern Public Clouds *3*

1.1.1 Influencing VM Placement *4*

1.1.2 Stealing Performance from Neighboring VMs *6*

1.2 Improving Isolation *9*

1.3 Summary of Contributions *11*

1.4 Outline *13*

2 Background **15**

2.1 Overview: Public Clouds *16*

2.1.1 Compute Cloud API *17*

2.1.2 Pricing Model *20*

2.1.3 Other cloud services *21*

2.2 Virtualization *22*

2.2.1 Compute Virtualization *22*

2.2.2 Network Virtualization *28*

2.2.3 Storage Virtualization *29*

2.3 Cluster Scheduler *30*
2.4 Side-channels in Public Clouds *33*
2.5 Other Non-IaaS Public Clouds *35*
2.5.1 Container-based Clouds *36*

3 Threat Model **38**
3.1 Cloud Provider *38*
3.2 Attacker *39*
3.3 Victim *40*

4 Motivation **42**
4.1 Evaluating Modern Clouds for Co-location Attacks *43*
4.1.1 Placement Policies used in Public Clouds *44*
4.1.2 Prior Work on Co-location Detection *45*
4.1.3 Challenges in Modern Clouds *46*
4.2 (Lack of) Performance Isolation between VMs *49*
4.3 Need for Defenses against Cross-VM Side-channels *51*
4.3.1 An Example Cross-VM Side-channel *52*
4.3.2 Prior Defenses and their Deficiencies *53*

5 Co-location in Public Clouds **56**
5.1 Definitions *57*
5.1.1 Defining Placement Vulnerability *61*
5.2 Detecting Co-Residence *62*
5.2.1 Co-residency Tests *62*
5.2.2 Cooperative Co-residency Detection *67*
5.2.3 Uncooperative Co-residency Detection *71*
5.3 Placement Vulnerability Study *76*
5.3.1 Experimental Methodology *76*
5.3.2 Effect of Number of Instances *81*
5.3.3 Effect of Instance Launch Time *83*

5.3.4 Effect of Data Center Location *87*
5.3.5 Effect of Launch Origin Machine *88*
5.3.6 Other Observations *89*
5.3.7 Cost of Launch Strategies *90*
5.4 Summary of Placement Vulnerabilities *92*
5.4.1 Limitations *94*
5.5 Placement Vulnerability in PaaS *95*

6 Stealing Performance from Neighboring VMs 97
6.1 Resource-freeing Attacks *98*
6.2 Contention Measurements *102*
6.3 RFA for Cache versus Network *106*
6.3.1 Evaluation on Local Testbed *111*
6.3.2 Evaluation on EC2 *116*
6.4 Discussion *122*

7 Soft-isolation: Improving Isolation in Public Clouds 126
7.1 MRT Guarantees and Soft Isolation *128*
7.2 Side-channels under MRT Guarantees *131*
7.3 Performance of MRT Mechanism *139*
7.3.1 Methodology *139*
7.3.2 Latency Sensitivity *140*
7.3.3 Batch Efficiency *142*
7.3.4 System Performance *145*
7.3.5 Summary *147*
7.4 Integrating Core-State Cleansing *149*
7.4.1 Design and Implementation *149*
7.4.2 Evaluation *151*

8 Related Work 156
8.1 Co-location in Public Clouds *156*

8.1.1 VM Placement Vulnerability Studies *156*
8.1.2 Defenses Against Co-location *157*
8.1.3 Co-residency Detection Techniques *158*
8.2 Lack of Performance Isolation in Public Clouds *160*
8.2.1 Performance Interference *161*
8.2.2 Performance Isolation *161*
8.2.3 Gaming Schedulers *162*
8.2.4 Side-channel Attacks *162*
8.3 Side-channel Attacks and Defenses *163*
8.3.1 Attacks *163*
8.3.2 Defenses *165*

9 Conclusion and Lessons Learned 168
9.1 Summary *169*
9.2 Conclusion *172*
9.3 Lessons Learned *173*
9.3.1 CPU Scheduling is Delicate *173*
9.3.2 Security Starts at the Hardware *178*

Bibliography 180

A Appendix A 199
A.1 Modeling the Effects of MRTs *199*

List of Figures and Tables

Figure 1.1 **Typical IaaS Public Cloud Architecture.** *Shows two important pieces of cloud software infrastructure that are responsible for providing isolation from malicious users: placement policy and hypervisor. A PaaS or a container-based cloud has a very similar architecture, but a hypervisor is replaced by an operating system and VMs by containers. More on this in Chapter 2.* 4

Figure 1.2 **Resource Contention in Virtualized Systems.** *The bar graph shows the worst-case performance degradation observed when running a microbenchmark in one VM that stresses a per-core resource (x-axis) with and without the same microbenchmark on another VM that is scheduled on the same package or socket but on different core. Note that the observed contention for CPU resource was in fact zero even when the VMs ran on the same host. This result was gathered on a machine with the following configuration: 4-core, 2-package 2.66 GHz Intel Xeon E5430 with 6MB of shared L2 cache per package and 4GB of main memory, with Xen configured on top of it with guest VM configuration: 1 GB memory, 40% cap and weight 256.* 8

Figure 2.1 **Instance Types in EC2, GCE and Azure Public Clouds.** *Popular public clouds have numerous predefined VM configurations under different tiers. The type (*) mentioned is an author defined tiers for VM types commonly found across public clouds. Shared-core instances are configured to consume only a fraction of a physical core and hence often share the same core with another tenant's VM. X-Optimized instance types satisfy special requirements for a cloud application like access to GPUs or better network or memory configuration than other available instance types.* x *in the instance type takes set of values forming a hierarchy of instance types. For example, t1.micro and t1.medium in EC2 are two instance types under same hardware configuration (t1 - machine generation with shared core) but different VM configuration.* `micro` *VM instances have 1 virtual CPU with 1GB of memory whereas* `medium` *instances have 2 virtual CPUs with 4GB of memory. This is the snapshot of instances types available as of October 2015.* 18

Figure 2.2 **VM Configurations of EC2 Instance Types..** *Only shows a subset of the available instance types in Amazon EC2 [13] but diverse enough to give an idea of various VM configurations available in the modern public clouds. EBS stands for Elastic Block Stores, which are network attached Hard Disk Drives (HDDs). Historically, EC2 provided an architecture agnostic CPU metric (called Elastic Compute Unit or ECU) but in 2014 EC2 exposed raw vCPU metrics that reflect the physical CPU limits. This is the reason the above CPU configuration varies for different instance types.* 18

Figure 2.3 **Public Cloud Pricing for EC2, GCE, Azure and Rackspace.** *Price per hour listed here are the price for on-demand small instances (EC2: t2.small, GCE: g1-small, Azure: Standard-A1, Rackspace: General1-1).* 20
Figure 2.4 **Process vs. Virtual Machine Abstraction.** 24

Figure 4.1 **Histogram of minimum network round trip times between pairs of VMs.** *The frequency is represented as a fraction of total number of pairs in each category. The figure does not show the tail of the histogram.* 47
Figure 4.2 **Resource contention in Xen.** *Percentage increase in workload run times indicated in row when contending with workload indicated in column. Percentage is computed as run time with contention over run time on otherwise idle machine. For network, run time is the time to serve a fixed number of requests. A dash means there was no significant performance degradation. In this experiment, the VMs are pinned to the same core. Here LLC stands for Last Level Cache.* 50

Table 5.1 **List of placement variables.** 58
Figure 5.2 **Memory-locking – Sender.** 63
Figure 5.3 **Memory-probing – Receiver..** 64
Table 5.4 **Local Testbed Machine Configuration.** *All are Intel machines. SMT stands for Simultaneous Multi-Threaded cores (in Intel parlance, Hyper-threads). Ways x Sets x Word Size gives the LLC size. The word size is 64 bytes on all these x86-64 machines. Here, NUMA stands for Non-Uniform Memory Access.* 65

Table 5.5 **Memory-probing and -locking on testbed machines.** *Slowdown relative to the baseline performance observed by the receiver averaged across 10 samples. Same – sender and receiver on different cores on the same socket, Diff. – sender and receiver on different cores on different sockets. Xeon E5645 machine had a single socket.* 66

Figure 5.6 **Distribution of performance degradation of memory-probing test.** *For varying number of pairs on each cloud (GCE:29, EC2:300, Azure:278). Note the x-axis plots performance degradation. Also for EC2 x-axis range is cut short at 20 pairs for clarity.* 68

Table 5.7 **Machine configuration in public clouds.** *The machine configurations observed over all runs with small instance types. GCE did not reveal the exact microarchitecture of the physical host (*). Ways $\times$ Sets $\times$ Word Size gives the LLC size. The word size for all these x86-64 machines is 64 bytes.* 69

Figure 5.8 **Distribution of performance degradation of memory-probing and -locking tests.** *On AMD machines in Azure with 40 pairs of nodes. Here NC stands for non-coresident and C, co-resident pairs. Note that the x-axis plots performance degradation.* 70

Figure 5.9 **An example victim web application architecture.** . . 72

Figure 5.10 **Co-residency detection on an uncooperative victim.** *The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-resident attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The load is in the number of concurrent users, where each user on average generates 20 HTTP requests per second to the webserver.* 74

Figure 5.11 **Varying number of webservers behind the load balancer.** *The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-residency attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The error bars show the standard deviation over 5 samples.* 75
Table 5.12 **Distribution of number of co-resident pairs on GCE.** 79
Table 5.13 **Distribution of number of co-resident pairs on EC2.** 80
Table 5.14 **Distribution of number of co-resident pairs on Azure.** *Region: East US 1.* 80
Figure 5.15 **Chances of co-residency of 10 victim instances with varying number of attacker instances.** *All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per run configuration with at least 3 runs per time of day.* 81
Figure 5.16 **Chances of co-residency of 30 attacker instances with varying number of victim instances.** *All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per run configuration with at least 3 runs per time of day.* 83

Figure 5.17 **Chances of co-residency with varying delays between victim and attacker launches.** *Solid boxes correspond to zero delay (simultaneous launches) and gauze-like boxes correspond to 1 hour delay between victim and attacker launches. We did not observe any co-resident instances for runs with zero delay on EC2. All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US). Results are over at least 9 runs per run configuration with at least 3 runs per time of day.* 84

Table 5.18 **Distribution of number of co-resident pairs and success rate or chances of co-residency for shorter delays under 20x20 run configuration in EC2.** *A delay with 0+ means victim and attacker instances were launched sequentially, i.e. attacker instances were not launched until all victim instances were running. The results averaged are over 9 runs with 3 runs per time of day.* 85

Figure 5.19 **Chances of co-residency over long periods.** *Results include 9 runs over two weeks with 3 runs per time of day under 20x20 run configuration. Note that we only conducted 3 runs for 32 hour delay as opposed to 9 runs for all other delays.* 85

Table 5.20 **Effect of time of day.** *Chances of co-residency when an attacker changes the launch time of his instances. The results were aggregated across all run configurations with 1 hour delay between victim and attacker launch times. All times are in PT.* 86

Figure 5.21 **Median number of co-resident pairs across two regions.** *The box plot shows the median number of co-resident pairs excluding co-residency within the same account. Results are over at least 3 run per run configuration (x-axis).* 87

Figure 5.22 **Distribution of number of co-resident instances per host on Azure.** *The results shown are across all the runs. We saw at most 2 instances per host in EC2 and at most 3 instances per host in GCE.* 88

Figure 5.23 **Launch strategy and co-residency detection execution times.** *The run configurations* $v \times a$ *indicates the number of victims vs. number of attackers launched. The error bars show the standard deviation across at least 7 runs.* 90

Table 5.24 **Cost of running a launch strategy.** *Maximum cost column refers to the maximum cost we incurred out of all the runs for that particular configuration and cloud provider (in dollars). The cost per hour of small instances at the time of this study were: 0.05, 0.026 and 0.06 dollars for GCE, EC2 and Azure, respectively. The minimum and maximum costs are in* ***bold****.* 91

Table 5.25 **Probability of co-residency under the reference placement policy.** 92

Table 5.26 **Cost benefit analysis.** $N = 1000$, $P(a_{type}) = 0.026$, *which is the cost per instance-hour on EC2 (the cheapest). For simplicity* $T_d(v, a) = (v * a) * 3.85$, *where* 3.85 *is fastest average time to detect co-residency per instance-pair. Here,* $v \times a$ *is the run configuration of the strategy under test. Note that the cost benefit is the additional cost incurred under the reference policy, hence is equal to cost incurred by* $a' - a$ *additional VMs.* 92

Table 6.1 **Xen configuration in our local testbed..** 103

Table 6.2 **Resource-specific workloads used to test contention..** 104

Table 6.3 **Resource Contention in Xen..** *Percentage increase in workload run times indicated in row when contending with workload indicated in column. Percentage is computed as run time with contention over run time on otherwise idle machine. For network, run time is the time to serve a fixed number of requests. A dash means there was no significant performance degradation. (Top) The VMs are pinned to the same core. (Middle) The VMs are pinned to different cores on the same package. (Bottom) The VMs are pinned to different packages.* 105

Table 6.4 **Performance Interference of a Webserver on LLCProbe.** *Runtimes (in microseconds) and percentage increase in performance degradation of LLCProbe (foreground) workload as a function of request rate to victim (background). For* Fixed Core *both VMs are pinned to the same core and for* Floating Core *Xen chooses where to execute them.* 108

Figure 6.5 **Performance of LLCProbe under Pinned and Floating VM Configurations..** *"Baseline" measures baseline performance when no traffic was sent to the victim; it is shown in each grouping for comparison. "No-RFA" measures performance when no RFA requests were sent. (Left) Performance when LLCProbe and web server VMs are pinned to same core. (Right) Performance when they float amongst cores. Error bars indicate one standard deviation.* 110

Figure 6.6 **Cumulative VM Runtime Distributions of Webserver and LLCProbeVMs..** *Runtime distribution of (top) the web server domain (with load 2,000 rps) and (bottom) the LLCProbe domain under both no RFA and with RFA 320 in pinned core case.* 113

Figure 6.7 **Offered vs. Observed Load on Webserver with Varying RFA Intensities..** *Here all the VMs float across all cores.* 114

Figure 6.8 **Normalized performance for SPEC workloads on our local testbed..** *Normalized performance is calculated with the baseline runtime over runtime for various RFA intensities. All values are at a web server request rate of 3000 rps.* 115

Table 6.9 **Summary of EC2 machines and Number of Co-resident Instances..** *For m1.small instances running under our EC2 accounts.* 118

Figure 6.10 **Normalized Performance Across All Machines on EC2..** *Here workload performance is normalized with baseline runtime under zero performance interference.* 119

Figure 6.11 **Average Runtimes of LLCProbe, bzip2, mcf, and sphinx Benchmarks Across 9 EC2 machines..** *Baseline has no traffic to victim, while No-RFA and 512 RFA have foreground request rate of 2000 rps.* 121

Figure 7.1 **Logic underlying the Xen MRT mechanism.** 130

Table 7.2 **Hardware configuration in local test bed.** 132

Figure 7.3 **Heatmaps of I-cache set timing as observed by a prime-probe attacker.** *Displayed values are from a larger trace of 10,000 timings. (a) Timings for idle victim and no MRT. (b)–(d) Timings for varying MRT values with the victim running.* . . . 133

Figure 7.4 **Heatmaps of I-cache set timings as observed by a prime-probe attacker for 10x slower victim computations.** *Displayed values are from a larger trace of 9,200 timings.* 134

Listing 7.5 **Modular exponentiation algorithm used in libgcrypt version 1.5.0.** *Note that the control flow followed when* $e_i = 1$ *is lines* $5 \to 6 \to 7 \to 8 \to 9$ *and when* $e_i = 0$ *is lines* $5 \to 6$*; denoted by the symbols* `1` *and* `0`*, respectively.* 135

Table 7.6 **The average and minimal number of ElGamal secret key bits operated upon between two attacker preemptions for a range of MRT values.** *Over runs with 40K preemptions.* 136

Table 7.7 **Rate of preemption with various MRT.** *Here the function called is the Modular-Exponentiation implementation in libgcrypt with a 2048 bit exponent. Note that for zero MRT the rate of preemption is very high that victim computation involving a single bit was preempted multiple times.* 136

Figure 7.8 **CDF of L2 data-loads per time slice experienced by OpenSSL-AES victim.** *L2 data loads are performance counter events that happen when a program requests for a memory word that is not in both L1 and L2 private caches (effectively a miss). When running along-side a Prime+Probe attacker, these data-cache accesses can be observed by the attacker.* 137

Table 7.9 **Xen configurations used for performance experiments.** 140

Table 7.10 **Workloads used in performance experiments.** 141

Figure 7.11 **95th Percentile Latency of Various Latency Sensitive Workloads.** *Under non-work-conserving scheduling.* 142

Figure 7.12 **95th Percentile Request Latency of *Data-Caching* Workload with Various Competing Micro-benchmarks.** *Under non-work-conserving scheduling.* 143

Figure 7.13 **25th, 50th, 75th, 90th, 95th and 99th Percentile Latency of *Data-Caching* Workload when run alongside *Nqueens*.** *Under non-work-conserving scheduling.* 144

Figure 7.14 **Average runtime of *graph500* workload when run alongside various competing workloads and under work-conserving scheduling.** *Averaged over 5 runs.* 145

Figure 7.15 **Average runtime of various batch workloads under non-work conserving setting.** *Note that for SPECjbb higher is better (since the graph plots the throughput instead of runtime). All data points are averaged across 5 runs.* 146

Figure 7.16 **Normalized Performance in a non-work-conserving configuration with 5msMRT.** *Normalized to performance under zero-MRT case. The left three workloads report 95th percentile latency and the right three report runtime, averaged across 5 runs. In both cases lower is better.* 148

Figure 7.17 **Instructions used to add noise.** *The assembly code is shown using X86 GAS Syntax.* `%rax` *holds the address of the random predicate used in the* `test` *instruction at the relative address* `0x2`*. The moves in the basic blocks* `<L13>` *and* `<L14>` *reads the data in the buffer, which uses up the corresponding D-cache set.* . . 151

Figure 7.18 **Median and 95th percentile latency impact of the cleansing scheduler under *worst-case* scenario.** *Here all the measured workloads are feed by a client at 500 requests per second. The error bars show the standard deviation across 3 runs.* 153

Figure 7.19 95th **percentile latency impact of the cleansing scheduler with varying load on the server.** *(a) Data-Caching and (b) Data-Serving. The error bars show the standard deviation across 3 runs.* 154

Figure 7.20 **Median latency impact of the cleansing scheduler with varying load on the servers.** *(a) Data-Caching and (b) Data-Serving. The error bars show the standard deviation across 3 runs.* 154

Listing A.1 **Pseudo-code of two sample procedures that leaks control-flow and data-access information.** PROCEDURE1 *leaks data-access information and* PROCEDURE2 *leaks control-flow information.* 200

1 Introduction

Public cloud computing has become a cheap, viable alternative for many small and medium enterprises [25–30, 68] considering the high upfront cost of setting up and maintaining a private datacenter. Its numerous benefits have also attracted large enterprises to host their workloads [27, 124, 135] on popular Infrastructure-as-a-Service (IaaS) public clouds like Amazon EC2 (EC2) [9], Microsoft Azure (Azure) [37], Google Compute Engine (GCE) [73], and Rackspace [146]. These public clouds offer a simple interface to create a virtual datacenter with compute (e.g., virtual machines or VMs), storage and network resources under a flexible pay-per-use pricing model. This enables any user to scale their virtual datacenter from a few tens to thousand VMs in matter of seconds for as less as 10-20 cents per VM per hour, which is impossible in a privately managed infrastructure. These enticing properties of public clouds are possible because of sharing the physical infrastructure maintained by a cloud provider with multiple arbitrary users over the Internet. This practice of *multi-tenancy* enables cloud provider to achieve high resource utilization [42, 118], which in turn drives down service cost for its users. For these reasons public clouds and the practice of multi-tenancy are here to stay for decades to come.

With the future of many commercial applications depending on the public clouds, there is one important problem that the cloud providers ought to solve to make them safe and usable – *isolating* the use of shared resources between multi-tenant workloads or tasks. Two user tasks are perfectly isolated if one task cannot *know* about the execution of the other task and its use of all the shared resources. This is an essential as any arbitrary user over the Internet with a valid credit card number and an email address can use the cloud infrastructure. Such an arbitrary user may also potentially share the same machine that is used to run the VMs that are part of medical [30, 34, 124], banking [26], e-commerce [25, 28, 68], or even government applications [33–35] running on these public clouds.

What could go wrong with sharing the same infrastructure with a potentially malicious user? Without proper isolation, an average user could inflict performance degradation on a performance-sensitive enterprise application, which may lead to huge monetary loss [18]. Apart from degradation or denial of service attacks, an adversary could surreptitiously steal sensitive information ranging from trade secrets [152] to private cryptographic keys [93, 187, 191, 192] without the knowledge of the victim or the cloud provider. We call these attacks as *co-location* or *co-residency* attacks as the adversary uses co-residency to affect either performance or security guarantees of the co-located tenants.

A successful co-location attack involves two steps: *place* and *breach*. The first step (place) in any co-location attack is placing VMs in the cloud datacenter such that some key resources are shared with one of the target victims. A straightforward choice of shared resource is a physical host[1]. Hence, we define co-location of two VMs as VMs that share a single physical host (we also refer to co-located VMs as neighbors). Following the placement step is breaching the isolation boundary to either affect the performance guarantees or steal secrets from the co-located target VM. For instance, it is not impossible for a neighboring VM to do a Denial-of-Service (DoS) attack on all VMs running on that host by just running a single machine instruction multiple times [170, 182][2].

This dissertation challenges the security of the state-of-the-art public

[1]A physical host in a datacenter is equivalent to 1 Rack Unit (or 1U)[42]

[2]This is a vulnerability in many x86 architectures that indirectly lock the shared memory bus, severely affecting all applications that depends on memory bandwidth.

clouds by taking the perspective of a malicious user who aims to use the strongest suite of public clouds, *multi-tenancy* against its tenants. The thesis that we want to address in this dissertation is: *"the practice of multi-tenancy in public clouds demands stronger isolation guarantees between VMs in the presence of malicious users."* To systematically evaluate this thesis, we study the two main pieces of software infrastructure, the placement policy or cluster scheduler [169][3] and the per-host hypervisor [168], and test their security against co-location attacks.

In the second part, we investigate ways to improve isolation that prevents these co-location attacks. Many straight-forward defense mechanisms require avoiding resource sharing and relinquish all the desirable efficiency properties along with it or give up on isolation or security altogether. We find that practical secure systems can be designed that hit a sweet spot that gets the best of both worlds.

1.1 Isolation in Modern Public Clouds

There are two important pieces of cloud software infrastructure (as shown in Figure 1.1) that ideally should make the co-location attacks hard if not prevent them in a multi-tenant public cloud. First is the *placement algorithm* that places VMs or user tasks on a physical host in a large cluster of hosts (or datacenter). This placement algorithm controls which tenant VMs are co-located on the same host (phenomenon is termed as *co-residency*). The second piece is the *hypervisor* that uses virtualization technology to provide isolation between execution of these multi-tenant VMs sharing various resources like CPU, CPU caches, memory, disk storage, network, etc.

[3][170] is the full version of this work.

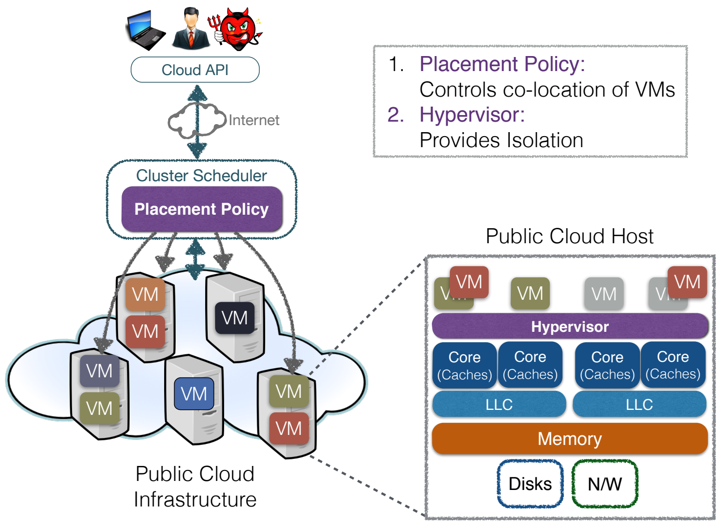


Figure 1.1: **Typical IaaS Public Cloud Architecture.** *Shows two important pieces of cloud software infrastructure that are responsible for providing isolation from malicious users: placement policy and hypervisor. A PaaS or a container-based cloud has a very similar architecture, but a hypervisor is replaced by an operating system and VMs by containers. More on this in Chapter 2.*

1.1.1 Influencing VM Placement

First step in any co-location attack involves co-locating an adversary controlled VM on the same host as one of the target tenant VMs in the cloud. *Can an adversary influence co-location with cloud interface available to any average cloud user?* This is an important question that needs to be answered to gauge the security property of the placement algorithm. Current cloud interfaces do not explicitly provide any control over VM-placement in the physical infrastructure and only provide interfaces to launch and terminate VMs.

There are two essential requirements for a successful VM co-location with a target victim: a launch strategy that increases chance of co-location and a co-location detection mechanism that detects a successful co-location.

Although cloud user may not explicitly control the placement policy, he/she may directly control values for certain input variables that are part of the cloud API (e.g., type of VM, choice of datacenter, time of launch) and indirectly control other variables (e.g., churn in the system time of day of the launch, day of the week). A set of values for these placement variables form a launch strategy. Note that a successful co-location is useless if it is impossible for the adversary to detect co-location with a victim VM in the wild.

In the past there are several studies [152, 182, 190] (some published in 2009) that found naïve launch strategies that involves launching large number of VMs until it results in a successful co-location. Further, co-location detection was also as simple as looking at the publicly accessible IP addresses of the victim and the attacker VMs. Such simple strategies no longer work in modern clouds either because of evolving (and unknown) placement algorithm or because of new countermeasures (e.g., Virtual Private Clouds) that makes it harder to detect co-location [93, 169]. There is also the ever increasing scale of the public clouds both in terms of the datacenter size and the cloud user base that makes it even harder to control placement by any single cloud user since 2008.

All these aspects of the public cloud seem to promote a false sense of security against malicious users. Counter-intuitively, we show that *an adversary can influence co-location and sometimes do so at a cost as low as 14 cents in some clouds* [169]. We do this study by: (1) enumerating various placement variables in the cloud VM launch API that may influence placement, (2) Assign values to these variable which forms a launch strategy, and (3) execute the launch strategy with two distinct accounts where one is a proxy for a victim and another the attacker. We repeat this on three live public clouds (EC2, Azure and GCE) and observe co-location between the two accounts to measure the placement policies' degree of susceptibility to co-location. We then quantify the cost of co-location and co-location success rate for each launch strategy. This forms the basis of our first-of-its-kind placement vulnerability study on public clouds. We also did brief experiments and found similar results on Platform-as-a-Service (PaaS) public cloud (e.g., Heroku [86]) that use containers for isolation as opposed to VMs.

Apart from showing that cheap launch strategies exists, we also show that an average user could also detect co-location with any victim VM even when they are part of a large multi-tiered cloud application. We use an existing covert-channel [182] that uses contention on shared memory bus for covert communication. But, instead of requiring cooperation from the victim, which is required in a covert-channel mechanism, we expose this contention via victim's application performance. That is just by measuring victim's application performance while simultaneously creating memory contention on the co-located attacker VM, one could reliably detect co-location without any cooperation from the victim or the cloud provider. To the best of our knowledge, this is the only uncooperative co-residency detection mechanism that is known to work in the modern clouds.

These findings reflect the state of the placement policies used in public clouds and how they may only optimize for efficiency or performance and that they fail to address security concerns that especially arises in the public clouds because of their openness and multi-tenancy properties.

1.1.2 Stealing Performance from Neighboring VMs

A poorly designed placement algorithm that is vulnerable to adversarial VM co-location may still be safe from co-location attacks if the hypervisor multiplexing multi-tenant VMs on a single host provides a strong VM isolation. Recall our definition of isolation, if one VM cannot know anything about the execution of other VMs or affect their performance via shared resources then it is said to be strongly isolated and hence secure from malicious users/VMs. But in reality hypervisors are far from providing even moderate levels of isolation between VMs. In fact, a malicious user can steal secret keys used in cryptographically secure protocols [93, 152, 187, 191, 192] by just observing a VM's use of one or more shared resources on that host. These are called *side-channel attacks*, as the name implies these are unintended communication channels in the system that may be used to siphon data.

There have been numerous works from security researchers demonstrating how to steal secrets from co-located VMs. But another intriguing question is: *can a greedy user steal performance from neighboring VMs?* This is also an important question that we seek to answer because in these public clouds performance often may directly translate to money as it may benefits one user at the expense of other VMs. This is also uniquely motivated by fixed pay-per-min or -hour pricing model, where a customer pays the same cost irrespective of the amount of useful work completed in a unit time.

Let there be two applications running in two separate VMs but on the same host and these two applications (or in general the VMs) compete for a set of resources on that host. It is well known that because of lack of proper isolation of some shared resources; a VM may perceive varying performance based on how many other VMs are competing for the same resource. This subclass of (lack of) isolation is also called (lack of) performance isolation. A comprehensive study that we did on a local testbed revealed that because of poor performance isolation in virtualized systems a competing VM can slow down the performance of another VM by $3\times$-$6\times$ compared to when run in isolation [168]. A subset of the results from this study is shown in Figure 1.2. With such huge performance degradation for even a short duration of time may translate into huge monetary loss or poor quality of service while incurring the same cost. Such significant performance loss may be sufficient to incentivize a greedy cloud user to workaround and relieve the situation.

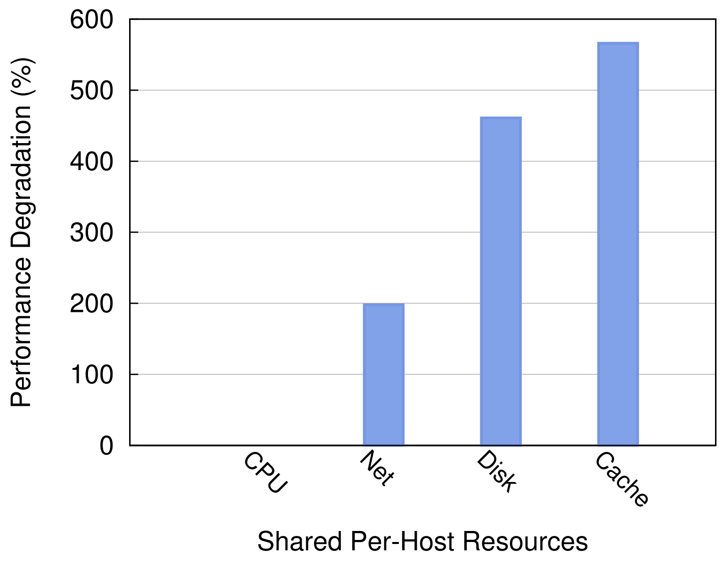


Figure 1.2: **Resource Contention in Virtualized Systems.** *The bar graph shows the worst-case performance degradation observed when running a microbenchmark in one VM that stresses a per-core resource (x-axis) with and without the same microbenchmark on another VM that is scheduled on the same package or socket but on different core. Note that the observed contention for CPU resource was in fact zero even when the VMs ran on the same host. This result was gathered on a machine with the following configuration: 4-core, 2-package 2.66 GHz Intel Xeon E5430 with 6MB of shared L2 cache per package and 4GB of main memory, with Xen configured on top of it with guest VM configuration: 1 GB memory, 40% cap and weight 256.*

In this dissertation, we found that an aggressive user can interfere with workloads running in neighboring VMs (victims) such that additional resources are freed up for his own VM's (beneficiary) consumption at no cost. We call such a class of attacks *Resource-Freeing Attacks* (RFA). A successful resource-freeing attack aims to shift the resource usage of the victim(s) *away* from the target, competing resource that the beneficiary cares about and *towards* a bottleneck resource, thus significant reducing the contention on the target shared resource.

Let us look at an example scenario that demonstrate RFAs. For instance, let us assume that two VMs are competing for memory bandwidth and in addition to the memory bandwidth, let the victim also relies on sequential disk bandwidth. It is well known that a workload's performance may rely on multiple and diverse set of resources. A resource-freeing attack in this scenario creates a disk bottleneck by periodically issuing random disk writes, which naturally frees up memory bandwidth as the victim waits for the disk to complete its sequential accesses. Note that the beneficiary VM's performance should not depend on disk bandwidth for a profitable RFA.

The above example is one of the many instances of RFAs possible in a poorly isolated system. All these RFAs help point out two important deficiencies in public cloud systems. First, a poorly designed pricing/-billing model may negatively incentivize adversaries to commit malicious activities. This is because a simple pricing model such as the pay per hour of usage does not account for the performance interference from other users and end up charging each user the same irrespective of the useful work done per unit time. Second, using schedulers that permit unlimited use of idle (or unused) resources (*i.e.,* work-conserving scheduler) increase the profitability of the RFAs. Thus, this work provides fresh insight into scheduler design paradigms that improves security.

1.2 Improving Isolation

In the light of the above two new findings that in contrary to traditional wisdom even large and busy public clouds are vulnerable to co-location attacks, the demand to improve isolation has never been higher.

A straightforward defense mechanism that can provide perfect isolation between VMs is: avoid any sharing between them. This could be realized either by strictly partitioning resources between users at the hypervisor or by having dedicated hardware for each tenant VM. Hence we call such strict form of isolation as *hard isolation*. Although this solves all the problems that arise in the presence of malicious tenants, it does it at a high cost and is impractical for multiple reasons. One, cloud providers no longer can multiplex resources and increase the service cost for customers [31]. Second, it does not bank on the idea that not all workloads use 100% of the resources all the time. Thus not sharing resources reduces the resource utilization of the datacenter [42, 118], which in turn increases the service cost.

This brings us to the problem of finding a solution that does not give up the pillars that form the public clouds – openness, multi-tenancy, sharing and low service cost. In other words, the challenge here is to find a defense mechanism that hits a sweet spot between efficiency and security. We investigate a new design principle called *soft-isolation*: reduce the risk of sharing through better scheduling.

We demonstrate soft-isolation paradigm by designing a hypervisor CPU scheduler to defend against one of the most dangerous class of co-location attacks: the cross-VM side-channel attacks that exploit per-core shared states to steal secrets from neighboring VMs [93, 191]. A successful cross-VM side-channel attack involves two main steps: (1) frequent interruption of the execution of a VM sharing the same per-core resource (e.g. TLB, L1 I/D-cache) and (2) measuring the time taken to access that shared resource using high precision time stamp counters (TSCs). As these per-core resources are typically stateful, the measured timing profile of the shared resource indirectly reveals the resource usage information of the other VM(s) sharing the same resource. This is the simplified version of a cross-VM side-channel attack although the actual attack is much more complex involving machine learning and multiple stages of post-processing of the timing profile.

As noted earlier, one essential requirement for the success of the side-channel attacks is the frequent preemptions, which were allowed by the state-of-the-art hypervisor CPU schedulers for the benefit of interactive workloads. But an attacker could abuse this scheduler feature to preempt a victim VM as quick as 10μs. To solve this problem, we adopted the soft-isolation principle that guides us to ratelimit dangerous cross-VM preemptions instead of avoiding any cross-VM preemptions (hard-isolation), which may be achieved by pinning each VM to different CPU core. We achieved this via a simple scheduler primitive we called the Minimum RunTime (MRT) guarantee. With MRT guarantee, a VM is allowed to run for a guaranteed time interval (MRT value) irrespective of any outstanding interrupts or preemptions. We discovered that a modest MRT value of 1-5ms was sufficient to prevent any known attacks [166]. Further extensive experimentation with microbenchmarks and realistic cloud applications revealed that a MRT value of 5ms incurs zero overhead on average and less than 7% overhead on the tail latency of interactive workloads like memcached [65] and cassandra [111].

1.3 Summary of Contributions

The work presented in this dissertation makes several contributions to the area of cloud computing security. Some of the major contributions are listed below.

- We systematically evaluated the placement policies used in public clouds, quantifying the susceptibility of some launch strategies towards co-location. This is the first-of-its-kind elaborate study that gives insight into how current efficiency optimized placement algorithms might have other security implications, in this case, cheap VM placement attack strategies. Apart from quantifying the cost of co-location and we also designed a framework to compare the security of placement policies used in distinct clouds against VM placement attacks. Further the findings also freshly motivate several previously published co-location attacks, as VM co-location is a prerequisite for the success of many co-location attacks.

- Apart from demonstrating that VM placement can be influenced in a live and busy public cloud systems, we also showed that co-location can be detected with any victim VM in a public cloud by just using quirks in the microarchitecture of current generation X86 machines.

- The comprehensive measurement study on the level of performance degradation due to resource contention helps gauge the quality of performance isolation in virtualized systems. The results help bolster the fact that even in 2015 state-of-the-art hypervisors still poorly isolate VM executions on a single host.

- We discover and demonstrate a new class of attacks called Resource-Freeing Attacks that could be used by a greedy customer to relieve massive contention (3×-6×) on essential shared resources and boast his performance at the neighbor's expense. This work exposes problems in the design of resource schedulers in a public cloud setting and shows how a flaw in the billing system could incentivize malicious activity in the public cloud.

- We evaluate the ability of system software to mitigate a class of cross-VM side-channel attacks (Prime+Probe) through scheduling. We propose a new design principle called soft isolation that aims to reduce the risk of sharing by avoiding dangerous cross-VM interactions when sharing the same processor. With extensive experimental measurements, we demonstrate that such a mechanism can effectively prevent existing Prime+Probe side-channel attacks at negligible overhead.

1.4 Outline

The rest of this dissertation is organized into the following sections.

- **Background.** In Chapter 2 we provide a general background to understand this dissertation, giving an overview of public clouds, virtualization, challenges in providing isolation, and background on cross-VM attacks in public clouds. That said, any readers who possess basic understanding of virtualization, public clouds, and cross-VM attacks can safely skip this chapter.

- **Threat model.** In Chapter 3 we define our assumptions about the cloud provider, the attacker and the victim and justify how these assumptions are valid and reasonable in a modern public cloud environment.

- **Motivation.** In Chapter 4 we list three specific problems that we tackle in this dissertation that we briefly described in this chapter and also argue why these problems help in validation our thesis about isolation in public clouds.

- **Evaluating and improving isolation in public clouds.** Chapters 5, 6 and 7 are the meat of this dissertation, where we go in greater detail about the three studies that we did about isolation in public clouds. First, in Chapter 5 we categorically show that targeted VM co-location is practical even in modern clouds and why researchers and cloud providers need to worry about co-location attacks. Second, in Chapter 6 we demonstrate how lack of isolation between VMs could incentivize new attacks in public clouds. Third, in Chapter 7 we propose a new design paradigm called soft-isolation that aims to improve isolation between co-resident VMs without compromising on efficiency. We also demonstrate on how this design principle can be employed to defend against the dangerous cross-VM side-channel attacks in the cloud. We record the results of all the three studies in their respective chapters.

- **Related work.** In Chapter 8 we survey many prior and contemporary works that are related to the ideas presented in this dissertation and compare them with our works.

- **Conclusion and lessons learned.** Finally, in Chapter 9 we summarize the results of the studies presented and argue how they help support the thesis of this dissertation. We also present some of the valuable lessons learned during the research endeavors.

2 Background

Historically, any enterprise that rely on compute resources to do simple web hosting invested in physical machine resources starting as a small cluster of few tens of machines and growing into a large private datacenter as the enterprise grows. The emergence and popularity of Amazon EC2 that enabled renting a fraction of a machine on-demand and pay only for the usage (per hour) [9] changed this requirement of an enterprise. The most challenging aspect of these Infrastructure-as-a-Service (IaaS) cloud providers is renting out a fraction of a single machine to disparate customers. This scenario is very similar to what operating systems faced in the era of mainframes where many users ran tasks on a shared mainframe computer with the expectation that operating system provided fair share of resources for every user and to isolated one user's task execution and data from another. In this era of cloud computing, instead of a single mainframe computer there is a cluster of racks with many nodes per racks that forms the shared infrastructure. The operating system for this cluster of nodes is responsible for where to place a task (placement) in this cluster, and isolate the tasks that share the same physical resources (isolation) among other responsibilities. The shared cloud resources include but are not limited to the per-node resources like CPU, memory, disk, and datacenter-wide resources like shared networks that connect different nodes and to the Internet. The user task is a Virtual Machine (VM), which is equivalent to a user process in the mainframe era.

In this chapter, we will provide a broad background on this operating system that manages the cloud but specifically focusing on how the system is designed to isolate resource usage of disparate users of the cloud. This background is essential to understand the new attacks, vulnerabilities uncovered in this dissertation and defense mechanisms that aim at improving the isolation guarantees in public clouds. Particularly, we will learn:

- How a cloud user interacts with the cloud and how is one billed?
- What is virtualization and how is it used to provide isolation between users?
- How does a cluster scheduler work and what is its role in providing isolation?
- What are some of the security challenges that public clouds face?

2.1 Overview: Public Clouds

To get a quick bird's eye view of a public cloud let us follow the life cycle of a VM, which is the fundamental scheduling entity for a cloud user.

Life cycle of a VM. A cloud user starts a VM by invoking the *cloud API* through which a desired set of parameters describing the configuration of the VM is communicated to the cloud provider. The provider initiates the resource allocation for the new VM; this process is called *VM provisioning*. VM provisioning consists of two steps: VM placement, which selects the physical host to run a VM using a *VM placement algorithm* followed by bootstrapping the VM on that host, which involves the per-host resource manager carving out physical resources for running the VM. The resulting VM-to-host mapping we call the *VM placement* and the per-host resource manager is called the hypervisor that is responsible for allocation and isolation of resources between VMs on that host. The VM interacts with the hypervisor to set up network and storage resources or request new services through other cloud management infrastructures, which we collectively refer to as the *cloud fabric*. The cloud fabric as a whole is also responsible for resource accounting, billing and performance monitoring for that VM. During the lifetime of the VM, it and its resources can be reconfigured. The cloud fabric may also choose to transparently migrate the VM to a different host. When the VM has completed its workload or when the compute resource is no longer needed, it saves any necessary state to a persistent storage. Then the cloud user invokes the cloud API to initiate a VM termination call to the cloud fabric. The cloud fabric reclaims all the resources associated with the VM, ending the life cycle of a VM.

Cloud Interface. All public clouds provide a management interface for customers to rent resources either using specialized Software-as-a-Services like a database engine [20], performance monitoring tool [6], load balancer [16], etc. or raw compute or storage resources in form of VMs or virtual disks. Typically, users start by registering an account with the cloud provider that at least require a credit card or bank account credentials and an email address. Then the users invoke the cloud interface to specify their resource requirements using the management interface or cloud API. There are several different APIs for accessing various services[1] but we restrict our discussion to the Cloud API that helps a user to configure, launch and terminate VMs in the public cloud (in IaaS parlance often referred to as Compute API).

2.1.1 Compute Cloud API

A typical cloud API consists many calls to the public cloud fabric that aid in VM configuration, which is not limited to launching and terminating VMs but attaching/detaching storage volumes, create/delete VM snapshots,

[1]For instance, GCE had more than 100 Cloud APIs that were available at the time when this dissertation was written.

Type*	EC2	GCE	Azure
Shared-core	t1.x, t2.x	f1-micro, g1-small	basic A0-A4
Standard	x.medium	n1-standard-N	standard A0-A7
Large	x.large, x.Nxlarge	n1-highcpu-N	D1-D4 & D11-D14
X-Optimized	memory, GPU, I/O	n1-highmem-N	network, memory

Figure 2.1: **Instance Types in EC2, GCE and Azure Public Clouds.** *Popular public clouds have numerous predefined VM configurations under different tiers. The type (*) mentioned is an author defined tiers for VM types commonly found across public clouds. Shared-core instances are configured to consume only a fraction of a physical core and hence often share the same core with another tenant's VM. X-Optimized instance types satisfy special requirements for a cloud application like access to GPUs or better network or memory configuration than other available instance types.* x *in the instance type takes set of values forming a hierarchy of instance types. For example, t1.micro and t1.medium in EC2 are two instance types under same hardware configuration (t1 - machine generation with shared core) but different VM configuration.* `micro` *VM instances have 1 virtual CPU with 1GB of memory whereas* `medium` *instances have 2 virtual CPUs with 4GB of memory. This is the snapshot of instances types available as of October 2015.*

Instance Type	CPU	Memory	Network	Storage
t2.micro	10% of 1 vCPU up to 3.3GHz	1GB	Low	EBS HDD
t2.medium	40% of 2 vCPU up to 3.3GHz	4GB	Low	EBS HDD
m3.medium	1 vCPU, 2.5 GHz	3.75GB	Moderate	4GB SSD
m3.xlarge	4 vCPUs, 2.5 GHz	15GB	High	80GB SSD
c4.xlarge	4 vCPUs, 2.9 GHz	7.5GB	High	EBS HDD

Figure 2.2: **VM Configurations of EC2 Instance Types..** *Only shows a subset of the available instance types in Amazon EC2 [13] but diverse enough to give an idea of various VM configurations available in the modern public clouds. EBS stands for Elastic Block Stores, which are network attached Hard Disk Drives (HDDs). Historically, EC2 provided an architecture agnostic CPU metric (called Elastic Compute Unit or ECU) but in 2014 EC2 exposed raw vCPU metrics that reflect the physical CPU limits. This is the reason the above CPU configuration varies for different instance types.*

assigning IP addresses, etc. (e.g., EC2 API [10]). We focus on two important calls that affect the life cycle of a VM: VM launch and VM terminate. When a user wishes to launch a VM (in EC2 called `RunInstances` [11]), it is required to pass several parameters that define the VM and its configurations. This include: VM instance type, datacenter location, VM image, number of VMs of this type apart from various common parameters that identify the user with credentials like security tokens and/or API keys. Each public cloud has several instance types that are basically predefined VM configurations with their own associated VM capacities. Some of the common instance types are shown in Figure 2.1. To get an idea of VM configurations associated with a VM type, we take a closer look at the configurations of a subset of instance types available in Amazon EC2 (shown in Figure 2.2). There is a spectrum of instance types that are ideal for different workloads. For example, m3.medium instance are ideal for small and mid-sized databases and c4.xlarge for high-performance front-end webservers, video encoding, gaming and distributed analytics [13].

Apart from instance types, users also have different choices for datacenter location. A datacenter location typically consists of two dimensions: 1. the actual geographical location of the datacenter, e.g., US-east (also called regions) and 2. the availability zone within a datacenter. Availability zones are cluster of machines in a region that are designed to fail independent of any failure in another availability zone in the same region. Here failures include network, power or other management failures that might happen in a datacenter. For example, EC2 has 4 regions in the US with 2-5 availability zones in each of these regions and has several regions in Europe and Asia. To fully utilize the presence of availability zones, customers are recommended to spawn redundant components of their application in at least 2 if not 3 availability zones. This increases the availability of their application as they could service user requests amidst unpredictable systems failure in other availability zones. It is important to note that

Cloud*	Price per hour	Granularity
EC2	\$0.026	per hour
GCE	\$0.027	per minute
Azure	\$0.060	per minute
Rackspace	\$0.037	per hour

Figure 2.3: **Public Cloud Pricing for EC2, GCE, Azure and Rackspace.** *Price per hour listed here are the price for on-demand small instances (EC2: t2.small, GCE: g1-small, Azure: Standard-A1, Rackspace: General1-1).*

users do not have the flexibility to choose a particular availability zone that another user may reside as the naming of availability zones differs from one customer account to another[2].

2.1.2 Pricing Model

The above cloud API enables any user to request VM instances of any type on-demand. The pricing model with which users are billed for the use of these instances varies across cloud providers. Figure 2.3 lists the prices for a similar instance type on four popular public clouds. Typically, all providers follow the pay-as-you-go pricing model where there are no upfront costs and users pays for the actual usage per unit time. The granularity at which each providers charges may be as high as one hour to as low as every minute. For instance, if a user starts an instance and run for only 10 minutes, EC2 and Rackspace charges the user for the whole hour but GCE and Azure charge only for the 10 minutes. Similarly, each providers charges for network resource per unit of data transmitted and for storage per unit of data per unit time. Overall, the pricing model is highly flexible compared to the cost incurred for setting up a datacenter and the maintenance cost associated with it. Interestingly, this simplistic pricing model also has some downsides like homogeneous pricing for heterogeneous hardware. For instance, there is a possibility that the same

[2]This is known to be true for Amazon EC2.

instance type might run on two different machines and be charged the same cost even though the performance may differ even when run the same workload inside those instances (e.g., m3.medium in EC2 may also run on an older generation microarchitecture) [61].

2.1.3 Other cloud services

One of the many advantages of the public clouds is its elasticity – ability to scale the size of the virtual datacenter of any user dynamically with minimal effort. In fact there are several higher-level services that monitor load and automatically launch or terminate instances based on the workload [32, 74, 150]. These services internally use the same mechanisms as users to configure, launch and terminate VMs but agree upon the cloud application configuration and needs ahead of time.

There are also other services to: 1. load balance client request over the Internet among a set of front-end servers [16], 2. provide Content Delivery Networks (CDNs) [5], 3. Database services [20] and many more.

Public vs. private clouds. Typically, an enterprise invests in a cluster of machines or a datacenter to satisfy its computing needs and is responsible for its maintenance as well. Historically, these clusters are managed using a mix of off-the-shelf and in-house enterprise tools. Recently, for ease of maintenance, these clusters are also managed using the same software management infrastructure (e.g., OpenStack [138]) that are employed in public clouds and hence, virtualize the cluster of machines. Unlike public clouds, the access to the datacenter is limited to only the employees of the enterprise. Such a private software/hardware infrastructure forms a private cloud. On the other hand, a public cloud has similar challenges as a private cloud but is accessible to any user over the Internet. In this dissertation, we focus on security concerns that arise in a public cloud in the presence of malicious users. Although many of the problems and solutions discussed here may also apply to private clouds, such discussions are out of the scope of this dissertation.

2.2 Virtualization

Virtualization is a method that allows multiple users or programs to share the same resources. It provides an illusion that a user has all of the physical resources but in fact the physical resource may be simultaneously shared between multiple users or each user uses the resource for a fraction of a time (time-multiplexing). This illusion is often called the virtual counter-part of the actual resource (e.g., CPU vs. Virtual CPU). For example, Operating Systems (OS) in the era of mainframes virtualized a single mainframe into multiple processes each of which had access to all the resources of the mainframe like memory, disks, CPU etc. Hence a process essentially is a virtual mainframe. In the public cloud setting, virtualization plays a major role in enabling sharing of a cluster of machines among disparate users over the Internet. Below we will take a close look at how three major resources (compute, network, storage) are virtualized in the cloud.

2.2.1 Compute Virtualization

Compute Virtualization is a combination of software (operating system) and hardware protection mechanisms that provide a task abstraction enabling resource sharing between disparate users. This is true since the era of mainframes where OSes with minimal hardware support virtualize all resources (e.g., CPU, memory, disk, network). Note that an OS that provides process abstraction is far away from the physical machine layer and is designed to support various system management tasks that are hidden behind the process abstraction. For example, all processes used the system call interface to off-load management tasks to the OS like memory, devices like disk, network, etc. But unlike operating systems that use a *process* abstraction, IaaS providers use a virtual machine (VM) abstraction. Figure 2.4 depicts these differences in the perspective of an application running in a process versus one running inside a VM environment. As the name suggests, the virtual machine is the lowest form of abstraction, where the system exposes the same interface a bare physical machine provides, machine instructions. Hence, unlike a process abstraction, VMs require only minimal management support from the system software. For the same reason, the VM abstraction also promises highest level of freedom enabling any user to independently run an operating system of their choice irrespective of whom they are sharing the physical machine with or the type of the machine they are running on. Here the system software that manages these VMs is called a *hypervisor*. These hypervisors are solely responsible for two main function – efficiently sharing resources and isolation between disparate VMs. In this section, we will take a close look at how the hypervisor strives to do both these functions.

Isolation. Recall the definition of isolation (in Chapter 1): two user tasks are perfectly isolated if one task cannot *know* about the execution of the other task and its use of all the shared resources. It is important to note this is a stronger form of isolation than is expected from an ideal hypervisor. In reality, it is hard if not impossible to achieve this strict form of isolation and we often refer to two forms of isolation called *logical* and *performance isolation*. Two VMs are logically isolated if one of the VMs cannot *name* or *identify* an instance of a resource that is also shared by the other VM. Examples for a shared namespace that violate logical isolation between two VMs are: a common physical host IP address or common hypervisor identifier (e.g., Dom0 identifier in Xen [152]). Two VMs are performance isolated if a VM is unable to *know* the presence of another VM by measuring the performance of a shared resource. A lack of performance isolation is realized, for instance, when a VM perceives a significant reduction in

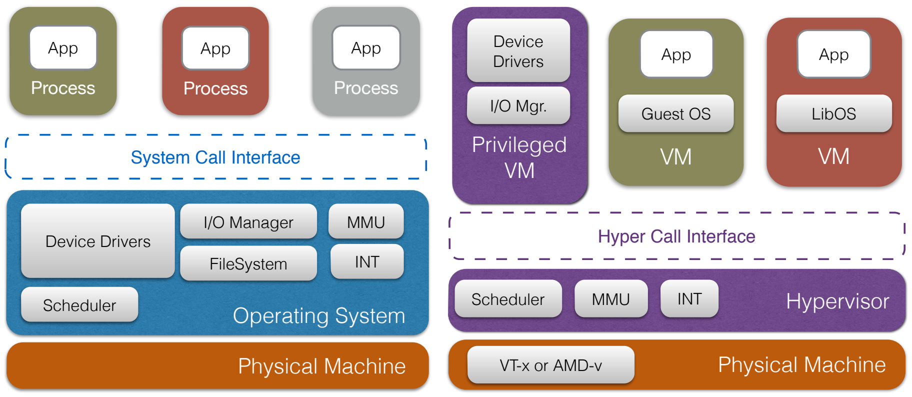


Figure 2.4: **Process vs. Virtual Machine Abstraction.**

CPU bandwidth when sharing the CPU with another VM relative to the bandwidth observed when run in isolation. There are other special forms of isolation like thermal or power isolation, where instead of performance, dissipation of heat or consumption of power, respectively, could also aid in the knowledge of other VMs in the system. Such highly specialized forms of isolation are out of scope of this dissertation. Ideally, a hypervisor aims to achieve both logical and performance isolation but in reality achieving perfect performance isolation is much harder than logical isolation.

Types of hypervisors. There are two different types of hypervisors that allow a full-fledged OS (also called as the guest OS) to run inside a VM. Type I or bare-metal hypervisors run directly on the physical hardware (e.g., Xen [40], KVM/QEMU [45, 105], HyperV [173], VMware ESX [130]). On the other hand, a Type II or hosted hypervisors run on top of a host OS using nested virtualization [46] hardware support or binary translation support [119] (e.g., VMware Workstation [163], VirtualBox [181]). In this dissertation, we will focus only on type I hypervisors, as the other type is unsuitable for a public cloud environment.

The popular public clouds differ in their choice of hypervisors; GCE uses KVM, Azure: HyperV, EC2 and Rackspace: Xen. Although all these hypervisors aim to achieve the same goals of isolation and close to bare-metal efficiency, they slightly differ in their system architecture and design. There are several metrics by which Hypervisors and their security guarantees are evaluated. One common metric is the Trusted Computing Base (TCB). A TCB is the minimal set of system software that is responsible for security guarantees of the system and is essential trusted. Hence, in general the size of the TCB could be used as a proxy to gauge the security of the system (smaller the better).

Xen [40] and HyperV [92, 173] follow a micro-kernel like system where the auxiliary management software is separated from the actual hypervisor and hence brandishes a very thin Trusted Computing Base (TCB). On the other hand, the KVM hypervisor, which comprises of the whole Linux OS and a kernel module (`kvm.ko`), has a large TCB. Although it has a large TCB, the reused Linux OS source is highly mature and taps into all the contributions to the mainline Linux OS. In this dissertation, we will focus on one of the hypervisors (Xen) that is freely accessible and is used by the most popular and a successful public cloud (EC2).

Xen Hypervisor

In order to understand the sources and effects of performance interference, we describe the Xen hypervisor mechanisms and policies for sharing resources between guest virtual machines while still providing performance isolation. A key reason to use VM-based virtualization in cloud computing is their ability to logical isolation between customer applications. This is because a VM-abstraction virtualizes everything about a machine, detaching a VM from its physical host. On top of this, achieving performance isolation is a primary goal for hypervisors like Xen hypervisor used in EC2 [40]. To this end, Xen and other hypervisors focus on fairly allocating CPU time and memory capacity [175]. However, other hardware resources such as memory bandwidth, processor cache capacity, network, and disk have received less attention.

CPU. The scheduler views VMs as a set of virtual CPUs (VCPUs), which are scheduled entities (like threads or processes in OS) and its task is to determine which VCPUs should be run on each physical CPU at any given time. The Xen scheduler provides both fair-share allocation of CPU and low-latency dispatch for I/O-intensive VCPUs. The commonly used CPU scheduler is the credit scheduler I [54].

The scheduler gives VCPUs *credits* at a pre-determined rate. The credits represent a share of the CPU and provide access to the CPU. Every $10ms$ a periodic scheduler tick removes credits from the currently running VCPU and if it has none remaining, switches to the next VCPU in the ready queue. VCPUs are given more credits periodically (typically every $30ms$). Thus, if a CPU-bound process runs out of credit, it must suspend for up to $30ms$ until it receives new credits to run. A VCPU that runs for short periods may never run out of credit, although the total amount it can accrue is limited.

In order to support low-latency I/O, Xen implements a *boost* mechanism that raises the priority of a VM when it receives an event (loosely translated as interrupt), which moves it towards the head of the ready queue. This allows it to preempt the running VM and respond to an I/O request immediately. However, a VM that has run out of credits cannot receive this boost priority. The boost mechanism is a key component that is often abused in many security attacks [166, 168, 191].

The credit scheduler supports a *work-conserving* mode, in which idle

CPU time is distributed to runnable VCPUs, and a *non-work-conserving* mode, in which VMs' CPU time is capped. The latter mode reduces efficiency but improves performance isolation. Though Amazon does not report which mode it uses, our experiments indicate that EC2 uses non-work-conserving scheduling.

On a multiprocessor, Xen can either *float* VPCUs, letting them execute on any CPU, or *pin* them to particular CPUs. When floating, Xen allows a VCPU to run on any CPU *unless* it ran in the last $1ms$, in which case it is rescheduled on the same core to maintain cache locality. Experiments show that EC2 configures Xen to float VCPUs across cores.

Memory. Xen isolates memory access primarily by controlling the allocation of memory pages to VMs. In cloud settings, Xen is often configured to give each VM a static number of pages. It does not swap pages to disk, actively manage the amount of memory available to each VM, or use deduplication to maximize use of memory [175]. Furthermore, x86 hardware does not provide the ability to enforce per-VCPU limits on memory bandwidth, hence Xen does not manage memory usage. Although recent Intel microarchitectures (since Broadwell, 2015) provide mechanisms to manage LLC usage in the OS (called Cache Allocation Technology (CAT) [94, 95]), current version of Xen is yet to utilize such facilities.

Devices. By default, Xen seeks fair sharing of disk and network by processing batches of requests from VMs in round-robin order [40]. For disks, this can lead to widely varying access times, as sets of random requests may incur a longer delay than sequential accesses. Further, Xen defaults to a work conserving scheduler for other devices, so performance can also degrade if another VM that was not using a device suddenly begins to do so. However, we observe that EC2 sets caps on the network bandwidth available to an m1.small instance at around 300 Mbps, but does not cap disk bandwidth (more details in Chapter 6).

2.2.2 Network Virtualization

Compute virtualization described above is essential to provide isolation for majority of the resources used by a VM but arbitrating and configuring datacenter network is equally important. Unlike other per-host resources, the network spans the whole datacenter and into the Internet. The network is often the only entry point for various security attacks and are hardest to secure because of the inherent distributed nature of the network (with a cluster of routers and switches).

Typically the cloud infrastructure, by default, configures each VM with at least two IP addresses, a *public* IP address for communication over the Internet, and a private or *internal* IP address for intra-datacenter communications. Cloud providers aim to fully virtualize the network by using a combination of several technologies [55]: network overlays for ease of virtualization in multi-tenant clouds [55], Virtual Local Area Networks (VLANs) [171] for network isolation for small networks, Software Defined Networking (SDN) for ease of management and control [53, 104, 121], Virtual Private Networks (VPNs) for secure inter-datacenter communication [154], middleboxes[3] and network firewalls for enforcing security policies. Understanding and evaluating security properties provided by these technology is not the focus of this dissertation. Readers interested in gaining deeper understanding of the network virtualization techniques are directed to relevant publications [55]. Overall, a combination of these technologies enables a dynamic, agile network that isolates each user's traffic from another. This enables a user to configure and manage the network as part of their own Virtual Private Cloud (VPC) [22, 44], where the user is given the view of a private cloud with a cluster of VMs whose interaction with each other and the Internet can be monitored and isolated from others.

[3]Network Function Virtualization is another related keyword that would be useful for any interested readers.

2.2.3 Storage Virtualization

In addition to virtualizing network, often cloud providers benefit from virtualizing storage systems as well. There are several ways to provide storage resources to a VM. A straightforward method is to expose a logical partition of a disk on that host to each VM running on the same host. Although this results in a simple storage manager, it has several drawbacks such as expensive VM migration, and complex VM snapshot process. For instance, on a VM failure the recovery process becomes complex without the availability of the host as the VM's persistent state is on the host's local disk. Current public clouds increasingly offer network-attached disks, where a virtual block-storage driver on the host emulates a local disk partition that transparently communicates with a disk cluster. Here the disk cluster could either be part of a Storage Area Network (SAN) or a Network-attached Storage (NAS). Although the details and distinction between these two types of decentralized storage architectures is out of scope of this dissertation, they help optimize for all the drawbacks of local disk storage or direct-attached storage system. Amazon EC2 calls their network-attached disks as Elastic Block Store (EBS) [8] and the local disks as ephemeral disks, offering them as scratch disks whose state are not persistent across VM termination. The added benefit of these network-attached disks is that it provides the flexibility of choosing either a Solid State Disks (SSDs) or Hard Disk Drives (HDDs) from any host with the added benefit of almost infinite disk capacity.

Apart from these storage virtualization that operates at the block-level, some cloud providers also provide specialized storage like object stores that are accessible via a separate API. For example, Amazon Web Services offers an object store called Amazon Simple Storage Service (S3) [21] and Google Cloud offers Google Cloud Storage [72]. Both of them expose a simple put-get key-value store style interface. These object stores do not provide rich querying interfaces and store unstructured binary blob associated with a key. There are also other NoSQL (e.g. DynamoDB [7], BigTable [70]) and SQL (e.g., Amazon RDS [20]) storage services that provide rich interfaces than the simple object stores. There are also storage services for specific use cases like data archiving and long term backup storage (e.g., Amazon Glacier [17]).

2.3 Cluster Scheduler

Cluster schedulers are responsible for selecting where to place VMs in a datacenter. The mapping of VM to host is called a VM placement and the logic that decides this mapping is the placement algorithm or scheduler. The primary goal of a VM placement scheduler is maximizing datacenter efficiency and resource utilization of its servers [42]. Apart from this, the scheduler also controls co-location of multi-tenant VMs and hence is indirectly responsible for isolation in public clouds.

The placement for a specific VM may depend on many factors: the load on each machine, the number of machines in the data center, the number of concurrent VM launch requests, etc. as defined by the *placement policy* for that public cloud. As mentioned earlier, the VM placement policies and the associated *placement algorithm* that implements it in public clouds aim to increase data center efficiency, quality of service, or both. For instance, a policy that aims to increase data center utilization may pack launched VMs on fewer machines. Similarly policies that optimize the time to provision a VM, which involves fetching an image over the network to the physical machine and booting, may choose the last machine that used the same VM image, as it may already have the VM image cached on local disks. Policies may vary across cloud providers, and even within a provider based on the nature of the VM launch requests.

Current public clouds do not disclose these placement policies and algorithms used and the average user entrusts the cloud provider to provide best-effort safety measures. The existence of an open source cloud management software, OpenStack [138] sheds light on the structure of a state-of-the-art cluster scheduler used in the wild. This is because a number of public clouds (and some private clouds) use a version of OpenStack to manage their datacenter (e.g., Rackspace [146], HPE Helion [89]).

OpenStack cluster scheduler. OpenStack's cluster scheduler (also called nova-scheduler) decides the placement of a VM in two stages. First, from the list of all hosts available in the datacenter it *filters* out hosts that cannot satisfy the VM launch request. The scheduler can be configured with more than one filter, each filter for a corresponding policy restriction. Second, it *weighs* and ranks the list of remaining hosts based on configured policy and chooses the host with highest weight to place the requested VM. Note that there is also a non-weight based scheduler in the second stage called the chance scheduler, which as the name implies randomly selects one of the filtered hosts.

It is evident that the choice of the filters used and the host weighing policy defines the placement policy under this OpenStack scheduler architecture. There are several options for the filters and host ranking for a cloud provider, and a few of them are described below (for the full list see the Openstack documentation [139]).

By default, OpenStack is configured to use to the filter scheduler with the following filters:

- *Availability zone filter*: selects only the machines that are in the requested availability zone,
- *RAM filter*: selects machines with sufficient physical memory,
- *Image properties filter*: selects machines that have the machine architecture, hypervisors that are compatible with the image of choice,

- *Affinity and Anti-affinity filter*: selects machines that already hosts a group of VMs (affinity) and avoids certain other VMs and their hosts (anti-affinity).

Regarding the weighing options, one can define a policy that enables either spreading of VMs across all available machines after filtering or stack VMs on single machine before moving to the next. Here the former increases machine utilization whereas later saves power. By default, a host's weight is calculated based on the amount of unused RAM and the weighing policy is defined using a multiplier that defines how the weight of that host grows with the amount of its unused RAM capacity. For example, a negative multiplier promotes packing VMs on the same host.

In the current version of OpenStack there are very few filters that help configure security policies. For instance, there is a filter to remove list of machine that hosts VMs that belong to untrusted users, although it is not clear how a user is defined trusted or untrusted. Overall, there are no known policies in OpenStack for managing co-location of multi-tenant VMs.

VM migration. During the lifetime of a VM, a VM may not always run on a single host on which it was placed at start. Cloud providers may also migrate VMs to another machine. There are several reasons that may trigger a VM migration event – power cycling hosts for maintenance or system upgrade purposes, efficiency reasons to pack multiple long running VMs on a small number of hosts, or security reasons to avoid long periods of co-location with a group of VMs. Although there are benefits for migrating VMs, migration affects the availability of the VM even if it is for a brief time [132]. For this reason may cloud providers turn to live VM migration as a last resort (e.g., rolling critical security update or other maintenance activity) [71]. Google Cloud acknowledges the possibility of VM migration for maintenance and provides a choice to the user to do live VM migration (the default option) or to terminate and re-launch the VM that is affected by the maintenance activity [71]. Other cloud providers provide only notifications in case of such disruptive VM migration activities. Hence, the VM migration is often not visible to the user and at the time of this writing no cloud providers provided any user interface to trigger VM migrations.

2.4 Side-channels in Public Clouds

Any lack of isolation in the cloud infrastructure manifests as a security concern in a public cloud environment in the presence of malicious users. In this dissertation we focus on cross-VM attacks where a malicious VM is co-located with the victim VM and exploits lack of isolation to steal sensitive information. In this section, we provide a background on side-channel attacks in this section.

Side-channels. Two users when on different machines may communicate through network protocols by using their Internet Protocol (IP) addresses, when on the same machine may use system provided communication channels (e.g., using shared memory) for communication. There are also unintended communication channels that may exist in a system that could be exploited by two users. These are called side-channels. Side-channels in a public cloud setting across VMs exist when there is a lack of proper isolation of shared resources used by those VMs. One can classify side-channels based on the lack of isolation they exploit, as: logical side-channels and performance side-channels.

Logical side-channels allow information leakage via naming or identification of a shared resource that is explicitly observable, e.g., IP addresses, timestamp counter values. For instance, it is known that an OS assigns a file descriptor or IDs for opened files. Often, the file descriptors are numbered starting from 0 and incrementally assigned to opened files. This file-descriptor name space is shared across all process. Two processes can use this shared namespace to communicate by opening and closing one or more files and denote them as 0 or 1, respectively.

Performance side-channels are created when performance variations due to resource contention are observable. For example, a shared disk can be used as a side-channel by using the performance characteristics of the sequential and random accesses, where sequential accesses complete faster than random accesses because of the spatial locality of the sequential addresses and the seek latency of magnetic disk heads. A task doing sequential access will observe a significant drop in its disk throughput if another task does a random disk read. Here the performance of the first task could be used to *know* the access done by the second task and hence establishing a performance side-channel.

Side-channel attacks. Note that in all the above examples, we considered two tasks cooperating and using the side-channel for communication. Such cooperative use of side-channel are often referred as *covert-channels*. There are also cases where the side-channels can be used to extract information that is potentially secret to another user. Such uses of a side-channel make it a security attack (i.e., a side-channel attack). For instance, the above disk performance side-channel may also be used to surreptitiously learn about disk requests done by another user (or a victim), which may indirectly leak some secret information (e.g., cryptographic secret keys used for confidential communication).

In public clouds, even though virtualization provides isolation between two VMs (good logical isolation, limited performance isolation), it is known that an attacker can use various performance side-channels across the VM boundary to steal cryptographic secrets (more on this in the later chapters).

Types of side-channel attacks. Side-channel attacks can be broadly classified into three classes: time-, trace-, and access-driven. Time-driven attacks arise when an attacker can glean useful information via repeated observations of the (total) duration of a victim operation, such as the time to compute an encryption (e.g., [2, 47, 52, 83, 107]). Trace-driven attacks work by having an attacker continuously monitor a cryptographic operation, for example via electromagnetic emanations or power usage leaked to the attacker (e.g., [66, 106, 145].

The third and the most damaging class of side-channel attacks are the access-driven side-channel attacks, in which the attacker is able to run a program on the same physical server as the victim. These abuse stateful components of the system shared between attacker and victim program. They have proved damaging in a wide variety of settings [3, 78, 141, 143, 152, 187, 191, 192]. In the cross-process setting, the attacker and victim are two separate processes running within the same operating system. In the cross-VM setting, the attacker and victim are two separate VMs running co-resident (or co-tenant) on the same server. The cross-VM setting is of particular concern for public IaaS [152, 191] and PaaS clouds [192], where it has been shown that an attacker can obtain co-residence of a malicious instance on the same server as a target [152].

2.5 Other Non-IaaS Public Clouds

Infrastructure-as-a-Service public clouds are not the only type of public clouds that are available. A second type of public cloud that is increasing in popularity is the Platform-as-a-Service (PaaS) clouds that, unlike IaaS clouds, export a process-level abstraction that is higher up the software stack. Unlike IaaS where users are granted full control of a VM, PaaS providers often managed compute tasks (or instances) for the execution of hosted web applications, and allow multiple such instances to share the same operating system. Typically, a PaaS user uploads application source code (e.g., Java, PHP, Python, Ruby) which are then deployed in one or more instance under the provider-managed host OS. Example of these PaaS clouds include, Heroku [86], Google App Engine [67], OpenShift by RedHat [137], Amazon Elastic Beanstalk [32] etc.

There are two major classes of PaaS clouds. One, that provides a restricted interface as defined by the programming language runtimes like Java Virtual Machine runtime or Python runtime interpreter. They often are run in sandboxed environment where two different PaaS instances' data and execution are isolated from each other and certain cloud providers further restrict the runtime APIs for security reasons [69]. An alternative approach popular in PaaS clouds is to provides IaaS like flexible interface with either process-level isolation via file system access controls, or increasingly as Linux containers (e.g., LXC [117], OpenVZ [140]). Linux containers use existing process-level isolation mechanisms (like cgroups [51]) available in the Linux OS to run isolated systems (or containers) that are managed by the same shared OS. They aim to provide flexible VM like abstraction but do it higher up the stack. The system architecture is similar to what is pictured in Figure 2.4, where processes are replaced with containers.

2.5.1 Container-based Clouds

An ardent reader would have noticed that a container-based PaaS clouds when compared to IaaS clouds, inherently do not provide (logical) isolation by design because of a lightweight, relatively flimsy process abstraction (refer to Figure 2.4). This is because, it is easier to ensure and enforce isolation at a lower level of the stack as the potential attack surface is naturally small with a well-defined interface with potentially untrusted code above the stack. On the other hand, a process-level abstraction has a large shared system software with loosely defined and large system-call interface that needs to be secured (refer to [192] for a detailed discussion). Despite this disadvantage, their increasingly popularity could be attributed to the significantly lower overhead of running an application inside a process (vs. inside a VM), and the ease of quickly launching and scaling PaaS instances. Emergence of application packaging ecosystems like Docker [58, 123] and open source application orchestration systems like Kubernetes [110] that ease PaaS application management have also contributed to its popularity. Often cloud providers work around the security limitations of PaaS clouds by running the PaaS instances of a single tenant inside a VM to get the best of both worlds. In this dissertation, we primarily focus on IaaS cloud and leave PaaS clouds for future work.

3 Threat Model

In this chapter we list the assumptions that we make about the public cloud provider, the adversary and the victim and justify these assumptions. This defines our threat model that we use throughout this dissertation.

3.1 Cloud Provider

Cloud provider is trusted. In our threat model, we assume that the cloud provider is trusted and strives to provide best-effort security and performance guarantees. We also assume that the cloud provider does not collude with adversaries, or have malicious insiders that may work for the benefit of an adversary. This is a reasonable assumption if we consider the level of trust we already place on the cloud provider. Hypervisors that manage VMs are under complete control of the cloud provider. As a result they have access to all data (memory, disk, network) associated with any VM run on these hypervisors. In addition, the cloud provider also has physical access to the machines that run customer workloads. Further, it is in the best interest for the cloud provider to make its infrastructure trustworthy to keep its business profitable. Hence, we trust the cloud provider. Although designing systems obviates the necessity to trust the hypervisor is also interesting and challenging, it is not the focus of this dissertation.

We assume a Type I hypervisor is employed by these cloud providers and the hypervisor is trusted as much as the cloud provider who manages it. That is, the cloud provider is motivated to avoid poor configurations of the hypervisor for the benefit of the users. For example, we assume that Simultaneous MultiThreading (SMT) and memory deduplication or sharing are disabled in the hypervisor, which are known to enable several classes of co-location attacks (e.g., [1] and [188], respectively).

3.2 Attacker

Adversary as any cloud user. We assume that the attacker has no affiliation of any form with the cloud provider. This partly follows from the above assumption of trusting the cloud provider. But, it also means that the adversary has no internal knowledge of the workings of the public cloud systems, (e.g., placement policies that are responsible for the VM placements, hypervisor configurations, cloud user information). As the cloud provider is trusted to do the right thing, it is not necessary to disclose any such knowledge to its customers. That said, it is only reasonable to assume that the adversary has access to any public knowledge that is explicitly revealed by the provider (e.g., explicit documentation by the provider) or implicitly accessible via usage of the cloud services through the existing interfaces that is available to any regular user.

Adversary is cost-conscious. We also assume that an adversary has enough credentials to request a number of user accounts for these public clouds. Even though there may be per-account limits that a cloud provider imposes (e.g., maximum of 30 active VMs per account), an adversary has access to an unlimited number of accounts. This is reasonable as an attacker may have access to number of user credentials, i.e., stolen credit cards and email addresses, which is sufficient to create a large number of user accounts. Hence, the adversary has no limit on the number of VMs he could launch at any given time. No resource limited cloud provider can defend against a resource unlimited adversary. For example, if a cloud provider has a maximum of 50,000 servers for hosting user VMs, a resource unlimited adversary can always achieve co-location with a victim by launching the maximum number of VMs required to achieve co-location with every VM in the public cloud. Such a scenario models a naïve, brute adversary and there are alternative mechanisms to defend against such adversaries [125]. In this dissertation, we are concerned about a sophisticated adversary who is resource conscious in terms of money and time spent on a successful attack. Such an adversary is also motivated to do so because of the low risk of detection associated with such cost conscious attacks.

Adversary controls VM's software stack. The adversary controls the entire software stack of its instances (all layers above the hypervisor), including the guest operating system and applications. Although live VM migration could be deployed by the cloud providers to improve isolation, we assume they are not used due to their disadvantages (refer to § 2.3). We later show through experiments that this configuration assumption is in fact true with many popular public cloud providers.

3.3 Victim

Victim reconnaissance. Regarding the information about the victim, we assume the adversary only has access to publicly available information as allowed by the cloud infrastructure. For instance, public IP address of a victim can be obtained by a DNS lookup and if the cloud infrastructure allows translation of a public IP address into an internal IP address, then the adversary has access to that information as well. Similarly, any public interface exposed by the victim's VMs is both accessible and identifiable by the attacker with help of certain tools (e.g., port scanning).

In this dissertation, we do not explicitly explore the ways in which an attacker can conduct victim reconnaissance. We assume that the adversary has access to tools to identify a set of target victims (and their VMs), and all associated information about the victim required for the attack. For example, to use an efficient launch strategy, the attacker might require either to know victim VMs' launch time or to directly trigger their launches. The latter is possible by increasing load in order to cause the victim to scale up by launching more instances as many users use auto-scaling features of the cloud (refer to § 2.1.3).

4 Motivation

With the background on public clouds presented in Chapter 2, we are aware of the benefits of multi-tenancy and how the state-of-the-art cloud systems share resources between disparate users. Although there is significant attention towards improving performance and efficiency of the public cloud infrastructure there is limited attention to evaluate the infrastructure for security and their isolation guarantees. In this chapter we will motivate three important problems that support our thesis: *"the practice of multi-tenancy in public clouds demands stronger isolation guarantees between VMs in the presence of malicious users."* The three problems we focus in this dissertation are:

1. Are co-location attacks impractical in modern clouds? (§ 5)
2. Are there unique opportunities for malicious users to exploit the lack of performance isolation for monetary or performance gains? (§ 6)
3. Can isolation be improved for security against malicious VMs without compromising the efficiency benefits of sharing? (§ 7)

In the following sections, we will motivate why these questions are important to answer and why it is important to do so now.

4.1 Evaluating Modern Clouds for Co-location Attacks

As we mentioned earlier, recall that co-location attacks involve two steps: place and breach. The side-channel attacks are examples of the breach step, which breaks the isolation boundary imposed by the hypervisor to steal secrets. Less understood is the ability of adversaries to arrange for co-residency in the first place. In general, doing so consists of using a *launch strategy* together with a mechanism for *co-residency detection*. Here a launch strategy is a clever invocation of the cloud interface that increases the chances of co-location. As no cloud provider share co-residency status of two disparate VMs, the co-residency detection mechanism helps in learning whether a launch strategy was successful in achieving co-residency with the victim VM(s).

The only prior work on obtaining co-residency [152] exposed that Amazon EC2 was vulnerable to simple network-topology-based co-residency checks and simple launch strategies like launching 20 VMs at the time of the victim VM launch. When such simple, advantageous launch strategies exist, we say the cloud suffers from a *placement vulnerability*. Since then, Amazon has made several changes to their infrastructure, including improving placement policies and removing the ability to do the simplest co-residency check. It should be noted that there are several other popular public clouds besides Amazon EC2 that hosts sensitive customer applications. Whether placement vulnerabilities exist in other public clouds has, to the best of our knowledge, never been explored.

The current state of the public clouds begs the question: is VM co-location possible in modern security-hardened clouds? To motivate this question, we will take a closer look at two aspects of modern clouds: 1. why prior work on co-location detection no longer work, giving a false sense of security to these attacks, 2. why we need a systematic study of placement policies for testing their resilience to co-location attacks.

4.1.1 Placement Policies used in Public Clouds

When a user launches a VM, the cluster scheduler chooses the physical host in the whole datacenter to provision the VM. The algorithm that decides this is called the *VM placement algorithm* and the resulting VM-to-host mapping we call the *VM placement*. The placement for a specific virtual machine may depend on many factors: the load on each machine, the number of machines in the data center, the number of concurrent VM launch requests, etc. The specifics of these factors that influence the VM placement form the *placement policy*.

VM placement algorithms used in public clouds often aim to increase data center efficiency, quality of service, or both by realizing some *placement policy*. For instance, a policy that aims to increase data center utilization may pack launched VMs on a single machine before choosing the next machine (called *stacking*). On the other hand, in order to avoid interference between similar workloads running inside VMs of the same user or avoid dependent failures, provider may also choose to *spread* the VMs on different hosts. Policies may vary across cloud providers, and even within a provider.

Public cloud placement policies, although undocumented, often exhibit behavior that is externally observable. One example is *parallel placement locality* [152], in which VMs launched from different accounts within a short time window are often placed on the same physical machine. Two instances launched sequentially, where the first instance is terminated before the launch of the second one, are often placed on the same physical machine, a phenomenon called *sequential placement locality* [152]. This may correspond to a policy which optimizes for VM-image to machine affinity, to speed up provision by using the already cached VM image and/or disk state in that host.

An adversary could capitalize on these advantageous strategies that exploit aspects of the placement policy. This in turn could benefit the adversary by reducing the cost and risk of detection of such attacks. Surprisingly, there here has been no study that investigated the security aspect of these efficiency-optimized placement policies. Although these placement policies and their associated placement behaviors are not explicitly exposed by the cloud provider, they can be systematically measured with careful experiments. Understanding the state of the art placement policies help both drive future works on designing a placement policy with the right balance between efficiency and security, and also provide a framework for systematically evaluating the new and improved placement policies in the future.

4.1.2 Prior Work on Co-location Detection

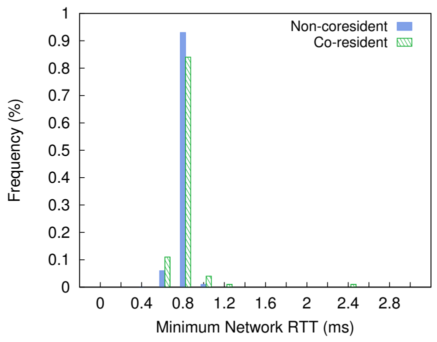
Ristenpart et al. [152] proposed several co-residency detection techniques and used them to identify several launch strategies in Amazon EC2. As co-resident status is not reported directly by the cloud provider, these detection methods usually exploit side-channels (either logical or performance side-channels described in Section 2.4). Ristenpart et al. discovered a logical side-channel via shared IP address namespace used in Amazon EC2. Particularly, each VM is assigned two IP addresses, a *public* IP address for communication over the Internet, and a private or *internal* IP address for intra-datacenter communications. The EC2 cloud infrastructure allowed translation of public IP addresses to their internal counterparts. This translation revealed the topology of the internal data center network, which allowed a remote adversary to map the entire public cloud infrastructure and determine, for example, the availability zone and instance type of a victim. Furthermore, co-resident VMs tended to have adjacent internal IP addresses.

Similarly, a performance side-channel over the network has also been used for detecting co-residence [152, 168]. This is because hypervisors often directly relay network traffic between VMs on the same host, providing detectably shorter round-trip times than between VMs on different hosts. Co-residency detection mechanisms are not limited to the above two techniques. Covert channels, as a special case of side-channels, can be established between two VMs that are cooperating in order to detect co-residency. For purposes of co-residency detection, covert channels based on shared hardware resources, such as last level caches (LLCs) or local storage disks, can be exploited by one VM to detect performance degradation caused by a co-resident VM [152]. Covert channel detection techniques require control over both VMs, and we later refer to such approaches as *cooperative co-residency detection*. For this reason, such a cooperative co-residency detection mechanism is usually used only in experimentation rather than in practical attacks.

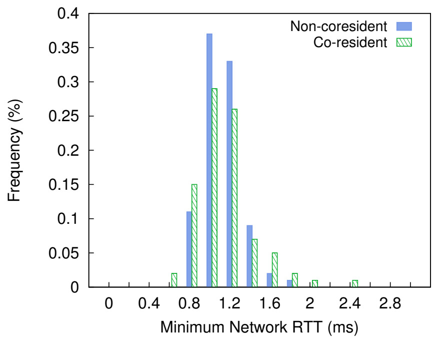
4.1.3 Challenges in Modern Clouds

Applying many of the detection techniques mentioned above is no longer feasible in modern clouds. In part due to the vulnerability disclosure by Ristenpart et al. [152], modern public clouds have adopted new technologies that enhance the isolation between cloud tenants and thwart known co-residence detection techniques. In the network layer, virtual private clouds (VPCs) have been broadly employed for data center management [22, 44]. With VPCs, internal IP addresses are private to a cloud tenant, and can no longer be used for mapping VMs location in the cloud infrastructure as described in the previous section[1]. Although EC2 allowed this in the older generation instances (called EC2-classic), this is no longer possible under Amazon VPC setting. In addition, VPCs require communication between tenants to use public IP addresses for communication. As shown in Figure 4.1, the network timing test is also defeated, as using pub-

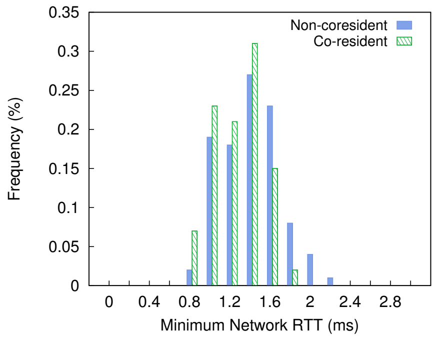
[1]This technique is termed as cloud cartography [152].



a: *GCE*



b: *EC2*



c: *Azure*

Figure 4.1: **Histogram of minimum network round trip times between pairs of VMs.** *The frequency is represented as a fraction of total number of pairs in each category. The figure does not show the tail of the histogram.*

lic IP addresses seems to involve routing in the data center network rather than short-circuiting through the hypervisor. Here, the ground-truth of co-residency is detected using a memory-based covert-channel (described later in Chapter 5). Notice that there is *no* clear distinction between the frequency distribution of the network round trip times of co-resident and non-coresident pairs on all three clouds.

In the system layer, persistent storage using local disks is no longer the default. For instance, many Amazon EC2 instance types do not support local storage [12]; GCE and Azure provide only local Solid State Drives

(SSD) [38, 75], which are less susceptible to detectable delays from long seeks. In addition, covert channels based on last-level caches [152, 168, 184, 190] are less reliable in modern clouds that use multiple CPU packages. Two VMs sharing the same machine may not share LLCs to establish the covert channel. Hence, these LLC-based covert-channels can only capture a subset of co-resident instances.

As a result of these technology changes, none of the prior techniques for detecting co-residency reliably work in modern clouds, compelling us to develop new approaches for our study. Apart from the lack of a working co-residency detection mechanism, there is no prior work that systematically evaluated the placement policies employed in public clouds for placement vulnerabilities.

Summary. Without a working co-residency mechanism, one may prematurely conclude that co-location attacks are impractical in security-hardened modern clouds. But taking a perspective of a highly motivated attacker, it is now more important to answer the question: *Are co-location attacks impractical in modern clouds?* If yes, it is a wakeup call for security researchers to concentrate on other important problems considering a plethora of previous works on co-location [93, 152, 182, 184, 188, 191, 192, 195]. If no, how could one evaluate different public clouds for placement vulnerability and quantify the cost such attacks? In Chapter 5 we show that not only that three most popular public clouds are vulnerable to VM placement but also that such placements are very practical and extremely cheap to exploit in a live cloud.

4.2 (Lack of) Performance Isolation between VMs

Cloud computing provides high efficiency in part by multiplexing multiple customer workloads onto a single physical machine. For example, Amazon's Elastic Compute Cloud (EC2) [19] runs multiple customer virtual machines (VMs) on a single host. For small instances, they offer each guest VM roughly 40% of a single CPU by time slicing. Similarly, access to the local disk, network, memory, and cache are all shared by virtual machines from multiple customers (as described in Section 2.2.1).

However, with this efficiency comes performance interference[2]. When two customer applications share a machine, they contend for access to resources. Existing hardware and software virtualization mechanisms do not provide perfect performance isolation. For example, running two applications that make heavy use of memory bandwidth can degrade the performance of both. Past work has demonstrated the existence and amount of this interference [41, 129].

As a result, there have been numerous proposals on how to construct hypervisors or processors that better isolate customer applications from each other. For example, fine-grained accounting of CPU usage [79], network traffic [158] or disk-queue utilization [77] can decrease the amount of interference. Unfortunately, the inherent tension between efficiency and isolation means that, in practice, cloud computing systems continue to provide poor isolation as witnessed below.

Extent of Resource Contention

With the goal of determining the worst-case resource contention between two VMs, we ran experiments on a local Xen testbed. The testbed machine

[2]We use performance interference and resource contention interchangeably.

Resources	CPU	Net	Disk	Memory	LLC
CPU	-	5	-	-	-
Net	-	194	-	-	-
Disk	-	-	455	-	-
Memory	-	6	-	-	-
LLC	8	539	72	38	34

Figure 4.2: **Resource contention in Xen.** *Percentage increase in workload run times indicated in row when contending with workload indicated in column. Percentage is computed as run time with contention over run time on otherwise idle machine. For network, run time is the time to serve a fixed number of requests. A dash means there was no significant performance degradation. In this experiment, the VMs are pinned to the same core. Here LLC stands for Last Level Cache.*

is a 4-core, 2-package 2.66 GHz Intel Xeon E5430 with 6MB of shared L2 cache per package and 4GB of main memory. This is representative of some of the architectures used by EC2[3]. We designed microbenchmarks that stress each type of resources and measured the contention as experienced by another VM for the same or a different resource. The result of this experiment is shown in Figure 4.2. We see significant degradation in performance for multiple resources even when run on separate VMs on the same machine. This demonstrates that Xen is not able to completely isolate the performance for many shared resources. More details on the experiment, system setup, and microbenchmarks are in Chapter 6.

Out of all resources, CPU and Memory show the least interference indicating that Xen does a good job accounting for CPU usage and that the physical hardware limits contention for memory bandwidth. However, for all other resources, there are competing workloads that substantially degrade performance. For instance, an LLC-intensive workload suffers a worst-case performance degradation of 500% when either run with a network- or memory-intensive workload. This is because competing workloads either interrupt frequently (Net) or move a lot of data through

[3]This is at the time of the study in 2011.

the cache (Memory).

Summary. We found that certain cache-sensitive workloads take 5x longer when contending with other memory-intensive workloads. Unlike in private data centers, resource contention in public clouds arises between disparate customers. Unique to the public cloud setting, then, is the incentive for greedy customers to attempt to free up this resource contention for their application. This is because, irrespective of the performance degradation perceived by the VMs, cloud providers charge all the VMs under the same simple pricing model – pay per unit time (described in Section 2.1). This gives rise to the question: *In the presence of performance interference, can a malicious user steal resources by interfering with neighboring VMs?* A naïve example of an attack would be crashing co-resident VMs, but this requires knowledge of an exploitable vulnerability and would be easily detectable. We are interested in whether there exist more subtle strategies for freeing up resources. In Chapter 6 we show that a new class of attacks called Resource-Freeing Attacks (RFAs) that exploit resource contention to improve ones own performance at the expense of others.

4.3 Need for Defenses against Cross-VM Side-channels

A poorly designed placement algorithm that is vulnerable to adversarial VM co-location may still be safe from co-location attacks if the hypervisor multiplexing multi-tenant VMs on a single host provides better performance isolation. As we saw above, existing hypervisors do not provide perfect because of the tension between efficiency (via sharing) and isolation. In this section, we will take a close look at the particular problem of cross-VM side-channels (refer to Section 2.4 for background) and find that there may be new unexplored directions to improve isolation without compromising on efficiency.

4.3.1 An Example Cross-VM Side-channel

Zhang, Juels, Reiter, and Ristenpart (ZJRR) [191] demonstrated the first cross-VM access-driven side-channel attack with sufficient granularity to extract ElGamal secret keys from the victim. They use a version of the classic Prime+Probe technique [141]: the attacker first *primes* the cache (instruction or data) by accessing a set of addresses that fill the entire cache. It then yields the CPU, causing the hypervisor to run the victim, which begins to evict the attacker's data or instructions from various cache. As quickly as possible, the attacker preempts the victim, and then *probes* the cache by again accessing a set of addresses that cover the entire cache. By measuring the speed of each cache access, the attacker can determine which cache lines were displaced by the victim, and hence learn some information about which addresses the victim accessed.

The ZJRR attack builds off a long line of cross-process attacks (c.f., [3, 4, 78, 141, 143]) all of which target per-core microarchitectural state. When simultaneous multi-threading (SMT) is disabled (as is typical in cloud settings), such per-core attacks require that the attacker time-shares a CPU core with the victim. In order to obtain frequent observations of shared state attacks abuse scheduler mechanisms that prioritize interactive workloads in order to preempt the victim. For example, ZJRR use inter-processor interrupts to preempt every 16µs on average. In their cross-process attack, Bangerter et al. abuse the Linux process scheduler [78].

Requirements for a successful attack. In this context, the best known attacks rely on:

1. *Shared per-core state* that is accessible to the attacker and that has visibly different behavior based on its state, such as caches and branch predictors.

2. *The ability to preempt the victim VM* at short intervals to allow only a few changes to that hardware state.

3. *Access to a system clock* with enough resolution to distinguish microarchitectural events (e.g., cache hits and misses).

These conditions are all true in contemporary multi-tenant cloud settings, such as Amazon's EC2. Defenses can target any of these dependencies, and we discuss some existing approaches next.

4.3.2 Prior Defenses and their Deficiencies

Past work on defenses against such side-channel attacks identified the above requirements for successful side-channels and tried to obviate one or more of the above necessary conditions for attacks. We classify and summarize these techniques below.

An obvious solution is to prevent an attacker and victim from sharing hardware, which we call *hard isolation*. Partitioning the cache in hardware or software prevents its contents from being shared [103, 149, 157, 179, 180]. This requires special-purpose hardware or loss of various useful features (e.g., large pages) and thus limits the adoption in a public cloud environment. Assigning VMs to run on different cores avoids sharing of per-core hardware [101, 118, 164], and assigning them to different servers avoids sharing of any system hardware [152]. A key challenge here is identifying an attacker and victim in order to separate them; otherwise this approach reduces to using dedicated hardware for each customer, reducing utilization and thus raising the price of computing.

Beyond hard isolation are approaches that modify hardware to add noise, either in the timing or by obfuscating the specific side-channel information. The former can be accomplished by removing or modifying timers [115, 120, 172] to prevent attackers from accurately distinguishing between microarchitectural events, such as a cache hit and a miss. For example, StopWatch [115] removes all timing side-channels and incurs a worst-case overhead of 2.8x for network intensive workloads. Specialized hardware-support could also be used to obfuscate and randomize processor cache usage [109, 180]. All of these defenses either result in loss of high-precision timer or require hardware changes.

Goals. All the above mentioned prior works either compromise on efficiency from resource sharing or add significant overhead or result in loss of a valuable feature making it hard for cloud providers to deploy the defense mechanism. For the same reason, none of the previously proposed defense mechanisms have not be deployed in public clouds to the best of our knowledge. Hence, it is important to address this lack of a deployable and effective defense mechanism against side-channels. The desirable properties of such a defense mechanism include:

1. Retain benefits of sharing, i.e., hard-isolation is not an option,
2. Do not compromise on any valuable features (e.g., high-resolution timers),
3. Easily deployable for cloud provider, i.e., no changes to hardware,
4. Zero or negligible overhead on legitimate workloads.

Notice that in addition to sharing resources and having access to fine-grained clocks, shared-core side-channel attacks also require the ability to measure the state of the cache *frequently*. For example, the ZJRR attack on ElGamal preempted the victim every 16μs on average [191]. With less frequent interruptions, the attacker's view of how hardware state changes in response to a victim becomes obscured. Perhaps surprisingly, then, is the lack of any investigation of the relationship between CPU scheduling policies and side-channel efficacy. In particular, scheduling may enable what we call *soft isolation*: limiting the frequency of potentially dangerous cross-VM interactions. (We use the adjective soft to indicate allowance of occasional failures, analogous to soft real-time scheduling.)

Summary. Although better co-location aware cluster scheduler may increase the cost of a co-location attack, VM co-location is always possible when the cloud provider is resource limited. This leads to the inevitable need to strengthen isolation between VMs at the hypervisor. Perhaps surprisingly, then, is the lack of a low overhead defense mechanism that does not compromise on efficiency benefits of sharing or other valuable features of the cloud. This raises the important question, *is it possible to design a hypervisor that achieves both improved isolation without compromising efficiency?* In Chapter 7 we try to answer this question and design a simple hypervisor CPU scheduler primitive that achieves these goals by taking advantage of the above requirements of the cross-VM side-channel attacks. This gives rise to a new design principle called *soft-isolation* where instead of strictly partitioning resources, the system limits and manages dangerous interactions between multi-tenant VMs.

5 Co-location in Public Clouds

In this chapter, we provide a framework to systematically evaluate public clouds for placement vulnerabilities and show that three popular public cloud providers may be vulnerable to co-location attacks. More specifically, we set out to answer four questions:

- Can co-residency be effectively detected in modern public clouds?
- Are known launch strategies [152] still effective in modern clouds?
- Are there any new exploitable placement vulnerabilities?
- Can we quantify the cost (time and money) of an attack for an adversary to achieve a certain probability of success?

We started this study by exploring the efficacy of prior co-residency tests, which we described in Chapter 4.1.2. This motivated us to develop more reliable tests for our placement study (§ 5.2.1). We also find a novel test to detect co-residency with VMs uncontrolled by the attacker by just using their public interface even when they are behind a load balancer (§ 5.2.3).

We use multiple customer accounts across three popular cloud providers, launch VM instances under different scenarios that may affect the placement algorithm, and test for co-residency between all launched instances. We analyze three popular cloud providers, Amazon Elastic Compute Cloud (EC2) [9], Google Compute Engine (GCE) [73] and Microsoft

Azure (Azure) [37], for vulnerabilities in their placement algorithm. After exhaustive experimentation with each of these cloud providers and at least 190 runs per cloud provider, we show that an attacker can still successfully arrange for co-location (§ 5.3). We find new launch strategies in these three clouds that obtain co-location faster (10x higher success rate) and cheaper (up to $114 less) when compared to a secure reference placement policy.

Before describing the details of the study, we start by defining terms that we will frequently encounter in the study in the next section.

5.1 Definitions

VM placement. The provider's VM launch service receives from a client a desired set of parameters describing the configuration of the VM. The service then allocates resources for the new VM; this process is called *VM provisioning*. We are most interested in the portion of VM provisioning that selects the physical host to run a VM, which we call the *VM placement algorithms*. The resulting VM-to-host mapping we call the *VM placement*. The placement for a specific virtual machine may depend on many factors: the load on each machine, the number of machines in the data center, the number of concurrent VM launch requests, etc.

Placement variables. While cloud providers do not generally publish their VM placement algorithms, there are several variables under the control of the user that could affect the VM placement, such as time-of-day, requested data center, and number of instances. A list of some notable parameters is shown in Table 5.1. By controlling these variables, an adversary can partially influence the placement of VMs on physical machines that may also host a target set of VMs. We call these variables *placement variables* and the set of values for these variables form a *launch strategy*. An example launch strategy is to launch 20 instances 10 minutes

Type	Variable
Placement Parameters	# of customers # of instances launched per customer Instance type Data Center (DC) or Region Time launched Cloud provider
Environment Variable	Time of the day Days of the week Number of in-use VMs Number of machines in DC

Table 5.1: **List of placement variables.**

after triggering an auto-scale event on a victim application. This is, in fact, a launch strategy suggested by prior work [152].

Placement policies. VM placement algorithms used in public clouds often aim to increase data center efficiency, quality of service, or both by realizing some *placement policy*. For instance, a policy that aims to increase data center utilization may pack launched VMs on a single machine. Similarly policies that optimize the time to provision a VM, which involves fetching an image over the network to the physical machine and booting, may choose the last machine that used the same VM image, as it may already have the VM image cached on local disks. Policies may vary across cloud providers, and even within a provider.

Public cloud placement policies, although undocumented, often exhibit behavior that is externally observable. One example is *parallel placement locality* [152], in which VMs launched from different accounts within a short time window are often placed on the same physical machine. Two instances launched sequentially, where the first instance is terminated before the launch of the second one, are often placed on the same physical machine, a phenomenon called *sequential placement locality* [152]. These placement behaviors are artifacts of the two placement policies described earlier, respectively.

Other examples of policies and resulting behaviors exist as well. VMs launched from the same accounts may either be packed on the same physical machine to maximize locality (and hence co-resident with themselves) or striped across different physical machines to maximize redundancy (and hence never co-resident with themselves). In the course of normal usage, such behaviors are unlikely to be noticed, but they can be measured with careful experiments.

Launch strategies. An adversary can exploit placement behaviors to increase the likelihood of co-locating with target victims. As pointed out by Ristenpart et al. [152], parallel placement locality can be exploited by triggering a scale-up event on target victim by increasing their load, which will cause more victim VMs to launch. The adversary can then simultaneously (or after a time lag) launch multiple VMs some of which may be co-located with the newly launched victim VM(s).

Cost of a launch strategy. Quantifying the cost of a launch strategy is straightforward: it is the cost of launching a number of VMs and running tests to detect co-residency with one or more target victim VMs. To be precise, the cost of a launch strategy S is given by $C_S = a \cdot P(a_{type}) \cdot T_d(v, a)$. Here a is the number of attacker VMs of type a_{type} launched to get co-located with one of the v victim VMs. $P(a_{type})$ is the price of running one VM of type a_{type} for a unit time. $T_d(a, v)$ is the time (in billing units) to detect co-residency between all pairs of a attackers and v victim VMs, excluding pairs within each group. For simplicity, we assume that the attacker is running all instances until the last co-residency check completes or that, equivalently. When the time to finish co-residency checks is within the granularity of one unit of billing time (e.g., one hour on EC2), this is equivalent to a more refined model.

Reference placement policy. In order to define placement vulnerability, we need a yardstick to compare various placement policies and the launch strategies that they may be vulnerable to. To aid this purpose, we define a simple reference placement policy that has good security properties against co-residency attacks and use it to gauge the placement policies used in public clouds. Let there be N machines in a data center and let each machine have unlimited capacity. Given a set of unordered VM launch requests, the mapping of each VM to a machine follows a uniform random distribution. Let there be v victim VMs assigned to v unique machines among N, where $v \ll N$. The probability of *at least one* collision (i.e. co-residency) under the random placement policy and the above attack scenario when attacker launches a instances is given by $1-\left(1-v/N\right)^{a}$. We call this probability the reference probability.[1] Recall that for calculating the cost of a launch strategy under this reference policy, we also need to define the price function, $P(vm_{type})$. For simplicity, we use the most competitive minimum price offered by any cloud provider as the price for the compute resource under the reference policy. For example, at the time of this study, Amazon EC2 offered t2.small instances at \$0.026 per hour of instance activity, which was the cheapest price across all three clouds considered in this study.

Note that the reference placement policy makes several simplifying assumptions, but these only benefit the attacker. This is conservative as we will compare our experimental results to the best possible launch strategy under the reference policy. For instance, the assumption on unlimited capacity of the servers only benefits the attacker as it never limits the number of victim VMs an attacker could potentially co-locate with. We use a conservative value of 1000 for N, which is at least an order-of-magnitude less than the number of servers (50,000) in the smallest reported Amazon EC2 data centers [23]. Similarly, the price function of this placement policy

[1]This probability event follows a hypergeometric distribution.

also favors an attacker as it provides the cheapest price possible in the market even though in reality a secure placement policy may demand a higher price. Hence, it would be troubling if the state-of-the-art placement policies used in public clouds does not measure well even against such a conservative reference placement policy.

5.1.1 Defining Placement Vulnerability

Putting it all together, we define two metrics to gauge any launch strategy against a placement policy: (i) *normalized success rate*, and (ii) *cost-benefit*. The normalized success rate is the success rate of the launch strategy in the cloud under test normalized to the success rate of the same strategy under the reference placement policy. The cost-benefit of a strategy is the additional cost that is incurred by the adversary in the reference placement policy to achieve the same success rate as the strategy in the placement policy under test. We define that a placement policy has a *placement vulnerability* if and only if there exists a launch strategy with a normalized success rate that is greater than 1.

Note that the normalized success rate quantifies how easy it is to get co-location. On the other hand, the cost benefit metric helps to quantify how cheap it is to get co-location compared to a more secure placement policy. These metrics can be used to compare launch strategies under different placement policies, where a higher value for any of these metrics indicate that the placement policy is relatively more vulnerable to that launch strategy. An ideal placement policy should aim to reduce both the success rate and the cost benefit of any strategy.

Study overview. In this study, we develop a framework to systematically evaluate public clouds against launch strategies and uncover previously unknown placement behaviors. We approach this study by (i) identifying a set of placement variables that characterize a VM, (ii) enumerating the most interesting values for these variables, and (iii) quantifying the cost of such a strategy, if it in fact exposes a co-residency vulnerability. We repeat this for three major public cloud providers: Amazon EC2, Google Compute Engine, and Microsoft Azure. Note that the goal of this study is not to *reverse engineer* the exact details of the placement policies, but rather to identify launch strategies that can be exploited by an adversary.

5.2 Detecting Co-Residence

An essential prerequisite for the placement vulnerability study is access to a co-residency detection technique that identifies whether two VMs are resident on the same physical machine in a third-party public cloud. As we saw in Chapter 4, co-residency detection is challenging in the modern security hardened public clouds. Below, we investigate and discover new mechanisms to reliably detection VM co-locations.

5.2.1 Co-residency Tests

We describe in this subsection a pair of tools for co-residency tests, with the following design goals:

- Applicable to a variety of *heterogeneous* software and hardware stacks used in public clouds.
- Detect co-residency with *high confidence*: the false detection rate should be low even in the presence of background noise from other neighboring VMs.
- Detect co-residency *fast* enough to facilitate experimentation among large sets of VMs.

We chose a performance covert-channel based detection technique that exploits shared hardware resources, as this type of covert-channels are often hard to remove and most clouds are very likely to be vulnerable to it.

A covert-channel consists of a *sender* and a *receiver*. The sender creates contention for a shared resource and uses it to signal another tenant that potentially share the same resource. The receiver, on the other hand, senses this contention by periodically measuring the performance of that shared resource. A significant performance degradation measured at the receiver results in a successful detection of a sender's signal. Here the reliability of the covert-channel is highly dependent on the choice of the shared resource and the level of contention created by the sender. The sender is the key component of the co-residency detection techniques we developed as part of this study.

```
// allocate memory multiples of 64 bits
char_ptr = allocate_memory((N+1)*8)
//move half word up
unaligned_addr = char_ptr + 2
loop forever:
   loop i from (1..N):
     atomic_op(unaligned_addr + i, some_value)
   end loop
```

Figure 5.2: **Memory-locking – Sender.**

Memory-locking sender. Modern x86 processors support atomic memory operations, such as XADD for atomic addition, and maintain their atomicity using cache coherence protocols. However, when a locked operation extends across a cache-line boundary, the processor may lock the memory bus temporarily [182]. This locking of the bus can be detected as it slows down other uses of the bus, such as fetching data from DRAM. Hence, when used properly, it provides a timing covert channel to send a signal to another VM. Unlike cache-based covert channels, this technique works regardless of whether VMs share a CPU core or package.

We developed a sender exploiting this shared memory-bus covert-channel. The psuedocode for the sender is shown in Figure 5.2. The sender creates a memory buffer and uses pointer arithmetic to force atomic operations on unaligned memory addresses. This indirectly locks the memory bus even on all modern processor architectures [182].

```
size = LLC_size * (LLC_ways +1)
stride = LLC_sets * cacheline_sz)
buffer = alloc_ptr_chasing_buff(size, stride)
loop sample from (1..10): //number of samples
     start_rdtsc = rdtsc()
     loop probes from (1..10000):
         probe(buffer); //always hits memory
     end loop
     time_taken[sample] = (rdtsc() - start_rdtsc)
end loop
```

Figure 5.3: **Memory-probing – Receiver..**

Receivers. With the aforementioned memory-locking sender, there are several ways to sense the memory-locking contention induced by the sender in another co-resident tenant instance. All the receivers measure the memory bandwidth of the shared system. We present two types of receivers that we used in this study that works on heterogeneous hardware configurations.

Memory-probing receiver uses carefully crafted memory requests that always miss in the cache hierarchy and always hit memory. This is ensured by constricting the data accesses of the receiver into a *single LLC set*. In order to evade hardware prefetching, we use a pointer-chasing buffer to randomly access a list of memory addresses (pseudocode shown in Figure 5.3). The time needed to complete a fixed number of probes (e.g., 10,000) provides a signal of co-residence: when the sender is performing locked operations, loads from memory proceed slowly.

Memory-locking receiver is similar to the sender but measures the number of unaligned atomic operations that could be completed per unit time. Although it also measures the memory bandwidth, unlike the memory-probing receiver, it works even when the cache architecture of the machine is unknown.

The sender along with these two receivers form our two novel co-residency detection methods that we use in this study: *memory-probing test* and *memory-locking test* (named after their respective receivers). These comprise our co-residency test suite. Each test in the suite starts by running the receiver on one VM while keeping the other idle. The performance measured by this run is the baseline performance without contention. Then the receiver and the sender are run together. If the receiver detects decreased performance, the tests conclude that the two VMs are co-resident. We use a slowdown threshold to detect when the change in receiver performance indicates co-residence (discussed later in the section).

Machine Architecture	**Clock** (GHz)	**SMT Cores**	**LLC** (Ways x Set)	**Memory Architecture**
Core i5-4570	3.20	4	12 x 8192	UMA
Core i7-2600	3.40	8	18 x 8192	UMA
Xeon E5645	2.40	6	16 x 12288	UMA
Xeon X5650	2.67	12	16 x 12288	NUMA

Table 5.4: **Local Testbed Machine Configuration.** *All are Intel machines. SMT stands for Simultaneous Multi-Threaded cores (in Intel parlance, Hyper-threads). Ways x Sets x Word Size gives the LLC size. The word size is 64 bytes on all these x86-64 machines. Here, NUMA stands for Non-Uniform Memory Access.*

Evaluation on local testbed. In order to measure the efficacy of this covert-channel we ran tests in our local testbed. Results of running the memory-probing and -locking tests under different machine configurations (Table 5.4) are shown in Table 5.5. The hardware architectures of

Machine Architecture	**Cores**	**Memory Probing**	**Memory Locking**	**Socket**
Xeon E5645	6	3.51	1.79	Same
Xeon X5650	12	3.61	1.77	Same
Xeon X5650	12	3.46	1.55	Diff.

Table 5.5: **Memory-probing and -locking on testbed machines.** *Slowdown relative to the baseline performance observed by the receiver averaged across 10 samples. Same – sender and receiver on different cores on the same socket, Diff. – sender and receiver on different cores on different sockets. Xeon E5645 machine had a single socket.*

these machines are similar to what is observed in the cloud [61]. Across these hardware configurations, we observed a performance degradation of at least 3.4$\times$ compared to not running memory-locking sender on a non-coresident instance (i.e. a baseline run with idle sender), indicating reliability. Note that this works even when the co-resident instances are running on cores on different sockets, which does not share the same LLC (works on *heterogeneous* hardware). Further, a single run takes one tenth of a second to complete and hence is also *quick*.

With the emergence of new microarchitectures (Haswell and Skylake) that promise better performance isolation of shared resources, we sought out to measure the efficacy of the above described memory-probing co-residency test on a (then) latest Skylake machine. The test machine is a single socket, 4 core Intel Core i7-6700K with 8MB LLC. We observed a 2$\times$ degradation in performance when running the memory-probing test on this machine. Although the performance degradation is significantly lesser than what we observed on Ivy Bridge and earlier microarchitectures (4$\times$), it demonstrates that modern processors are still vulnerable to the abuse of the unaligned atomic operations.

Note that for this test suite to work in the real world, an attacker requires control over both the VMs under test, which includes the victim.

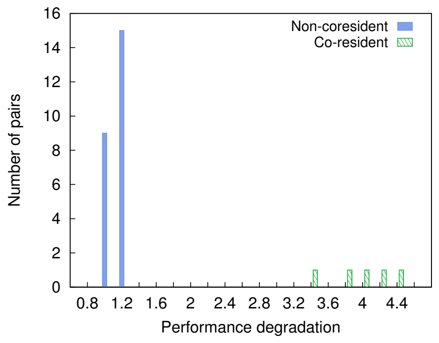
We call this scenario as co-residency detection under cooperative victims (in short, *cooperative co-residency detection*). Such a mechanism is sufficient to observe placement behavior in public clouds (Section 5.2.2). We further investigated approaches to detect co-residency under a realistic setting with an uncooperative victim. In Section 5.2.3 we show how to adapt the memory-probing test to detect co-location with one of the many webservers behind a load balancer.

5.2.2 Cooperative Co-residency Detection

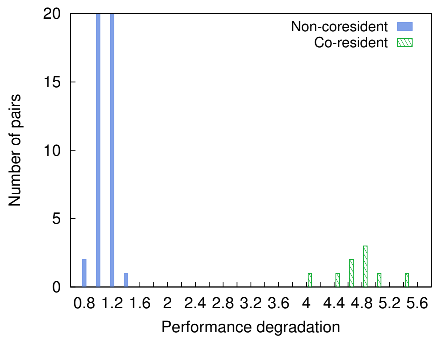
In this section, we describe the methodology we used to detect co-residency in public clouds. For the purposes of studying placement policies, we had the flexibility to control both VMs that we test for co-residence. We did this by launching VMs from two separate accounts and test them for pairwise co-residence. We encountered several challenges when running the co-residency test suite on three different public clouds - Google Computer Engine, Amazon EC2 and Microsoft Azure.

First, we had to handle noise from neighboring VMs sharing the same host. Second, hardware and software heterogeneity in the three different public clouds required special tuning process for the co-residency detection tests. Finally, testing co-residency for a large set of VMs demanded a scalable implementation. We elaborate on our solution to these challenges below.

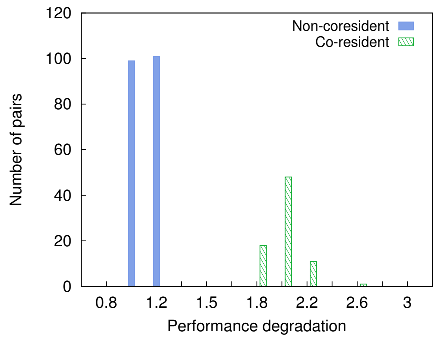
Handling noise. Any noise from neighboring VMs could affect the performance of the receiver with and without the signal (or baseline) and result in misdetection. To handle such noise, we alternate between measuring the performance with and without the sender's signal, such that any noise equally affects both the measurements. Secondly, we take ten samples of each measurement and only detect co-residence if the ratios of *both* the mean and median of these samples exceed the threshold. As each



a: *GCE*



b: *EC2*



c: *Azure – Intel machines*

Figure 5.6: **Distribution of performance degradation of memory-probing test.** *For varying number of pairs on each cloud (GCE:29, EC2:300, Azure:278). Note the x-axis plots performance degradation. Also for EC2 x-axis range is cut short at 20 pairs for clarity.*

run takes a fraction of a second to complete, repeating 10 times is still fast enough.

Tuning thresholds. As expected, we encountered different machine configurations on the three different public clouds (shown in Table 5.7) with heterogeneous cache dimensions, organizations and replacement policies [96, 99]. This affects the performance degradation observed by the receivers with respect to the baseline and the ideal threshold for detecting

Cloud Provider	**Machine Architecture**	**Clock** (GHz)	**LLC** (Ways $\times$ Set)
EC2	Intel Xeon E5-2670	2.50	20×20480
GCE	Generic Xeon*	2.60*	20×16384
Azure	Intel E5-2660	2.20	20×16384
Azure	AMD Opteron 4171 HE	2.10	48×1706

Table 5.7: **Machine configuration in public clouds.** *The machine configurations observed over all runs with small instance types. GCE did not reveal the exact microarchitecture of the physical host (*). Ways $\times$ Sets $\times$ Word Size gives the LLC size. The word size for all these x86-64 machines is 64 bytes.*

co-residency. This is important because the thresholds we use to detect co-residence yield false positives, if set too low, and false negatives if set too high. Hence, we tuned the threshold to each hardware we found on all three clouds.

We started with a conservative threshold of 1.5x and tuned to a final threshold of 2x for GCE and EC2 and 1.5x for Azure for both the memory-probing and -locking tests. Figure 5.6 shows the distribution of performance degradation under the memory-probing tests across Intel machines in EC2, GCE, and Azure. For GCE and EC2, a performance degradation threshold of 2 clearly separates co-resident from non-coresident instances. For all Intel machines we encountered, although we ran both memory-locking and -probing tests, memory-probing was sufficient to detect co-residency. For Azure, overall we observe lower performance degradation and the initial threshold of 1.5 was sufficient to detect co-location on Intel machines.

The picture for AMD machines in Azure differs significantly as shown in Figure 5.8. The distribution of performance degradation for both the memory-locking and the memory-probing tests shows that, unlike for other architectures, co-residency detection is highly sensitive to the choice of the threshold for AMD machines. This may be due to the more asso-

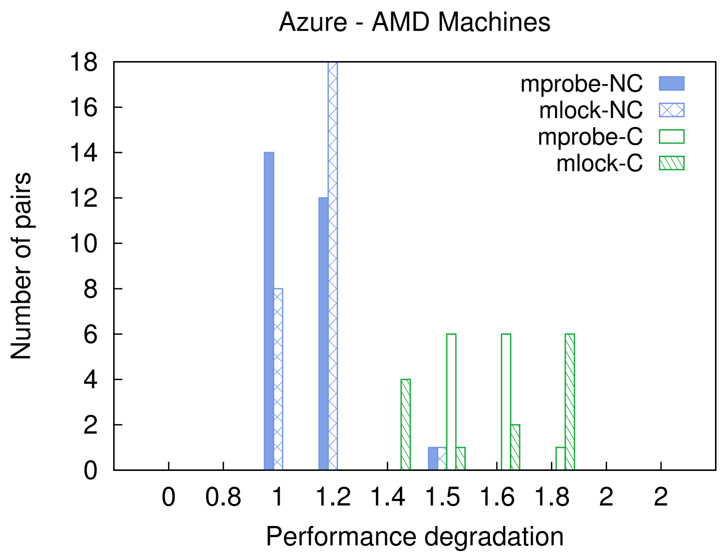


Figure 5.8: **Distribution of performance degradation of memory-probing and -locking tests.** *On AMD machines in Azure with 40 pairs of nodes. Here NC stands for non-coresident and C, co-resident pairs. Note that the x-axis plots performance degradation.*

ciative cache (48 ways vs. 20 for Intel), or different handling of locked instructions. For these machines, a threshold of 1.5 was high enough to have no false positives, which we verified by hand checking the instances using the two covert-channels and observed consistent performance degradation of at least 50%. We determine a pair of VMs as co-resident if the degradation in *either* of the tests is above this threshold. We did not detect any cross-architecture (false) co-residency detection in any of the runs.

Scaling co-residency detection tests. Testing co-residency at scale is time-consuming and increases quadratically with the number of instances: checking 40 VM instances, involves 780 pair-wise tests. Even if each run of the entire co-residency test suite takes only 10 seconds, a naïve sequential execution of the tests on all the pairs will take 2 hours. Parallel co-residency checks can speed checking, but concurrent tests may interfere with each other.

To parallelize the test, we partition the set of all VM pairs ($(\binom{}{v+a2})$) into sets of pairs with no VMs twice; we run one of these sets at a time and record which pairs detected possible co-residence. After running all sets, we have a set of candidate co-resident pairs, which we test sequentially. Parallelizing co-residency tests significantly decreased the time taken to test all co-residency pairs. For instance, the parallel version of the test on one of the cloud providers took 2.4 seconds per pair whereas the serial version took almost 46.3 seconds per pair (a speedup of 20x). While there are faster ways to parallelize co-residency detection, we chose this approach for simplicity.

Veracity of our tests. Notice that a performance degradation of 1.5x, 2x and 4x corresponds to 50%, 100% and 300% performance degradation. Such high performance degradation (even 50%) is clear enough signal to declare co-residency due to resource sharing. Furthermore, we hand checked by running the two tests in isolation on the detected instance-pairs for a significant fraction of the runs for all clouds and observed a consistent covert-channel signal. Thus our methodology did not detect any false positives, which are more detrimental to our study than false negatives. Although *co-residency* here implies sharing of memory channel, which may not always mean sharing of cores or other per-core hardware resources.

5.2.3 Uncooperative Co-residency Detection

Until now, we described a method to detect co-residency with a cooperative victim. In this section, we look at a more realistic setting where an adversary wishes to detect co-residency with a victim VM with accesses limited to only public interfaces like HTTP or a key-value (KV) store's put-get interface. We show that the basic cooperative co-residency detection can also be employed to detect co-residency with an uncooperative victim in the wild.

Attack setting. Unlike previous attack scenarios, we assume the attacker has no access to the victim VMs or its application other than what is permitted to any user on the Internet. That is, the victim application exposes a well-known public interface (e.g., HTTP, FTP, KV-store protocol) that allows incoming requests, which is also the only access point for the attacker to the victim. The front end of this victim application can range from caching or data storage services (e.g., memcached, cassandra) to generic webservers. We also assume that there may be multiple instances of this front-end service running behind a load balancer. Under this scenario, the attacker wishes to detect co-location with one or more of the front-facing victim VMs.

Co-residency test. We adapt the memory tests used in previous section by running the memory-locking sender in the attacker instance. For a receiver, we use the public interface exposed by the victim by generating a set of requests that potentially makes the victim VMs hit the memory

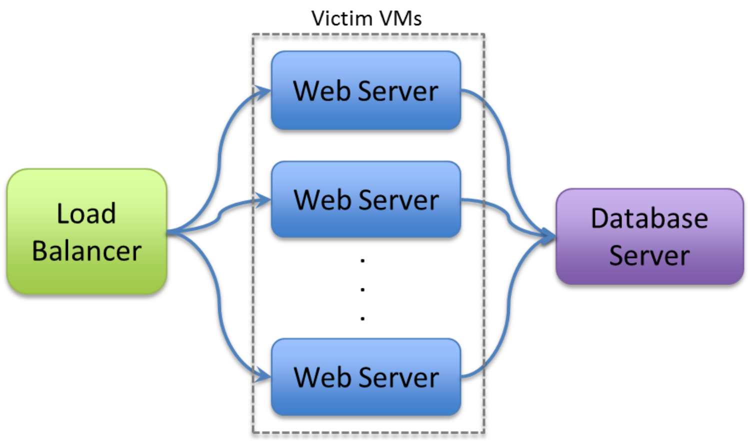


Figure 5.9: **An example victim web application architecture.**

bus. This can be achieved by looping through a large number of requests of sizes approximately equal or greater than the size of the LLC. This creates a performance side-channel that leaks co-residency information. This receiver runs in an independent VM under the adversary's control, which we call the co-residency detector.

Experiment setup. To evaluate the efficacy of this method, we used the Olio multi-tier web application [136] that is designed to mimic a social-networking application. We used an instance of this workload from CloudSuite [64]. Although Olio supports several tiers (e.g., memcached to cache results of database queries), we configured it with two tiers as shown in Figure 5.9, with each webserver and the database server running in a separate VM of type t2.small on Amazon EC2. Multiple of these webserver VMs are configured behind a HAProxy-based load balancer [82] running in an m3.medium instance (for better networking performance). The load balancer follows the standard configuration of using round-robin load balancing algorithm with sticky client sessions using cookies. We believe such a victim web application and its configuration is a reasonable generalization of real world applications running in the cloud.

For the attacker, we use an off-the-shelf HTTP performance measurement utility called `HTTPerf` [128] as the receiver in the co-residency detection test. This receiver is run inside a t2.micro instance (for free of charge). We used a set of 212 requests that included web pages and web objects (images, PDF files). We gathered these requests using the access log of manual navigation around the web application from a web browser.

Evaluation methodology. We start with a known co-resident VM pair using the cooperative co-residency detection method. We configure one of the VMs as a victim webserver VM and launch four more VMs: two webservers, one database server and a load balancer, all of which are not co-resident with the attacker VM.

Co-residency detection starts by measuring the average request latency

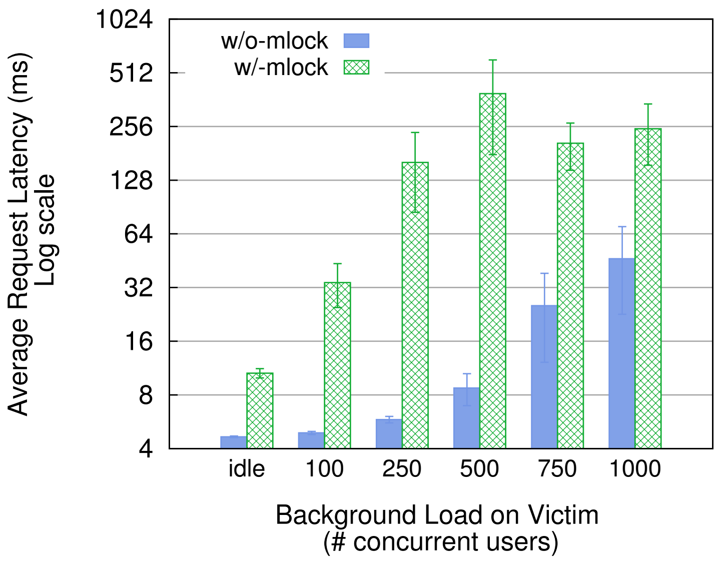


Figure 5.10: **Co-residency detection on an uncooperative victim.** *The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-resident attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The load is in the number of concurrent users, where each user on average generates 20 HTTP requests per second to the webserver.*

at the receiver inside the co-residency detector for the baseline (with idle attacker) and contended case with the attacker running the memory-locking sender. A significant performance degradation between the baseline and the contended case across multiple samples reveal co-residency of one of the victim VMs with the attacker VM. On Amazon EC2, with the above setup we observed an average request latency of 4.66ms in the baseline case and a 10.6ms in the memory-locked case, i.e., a performance degradation of $\approx 2.3\times$.

Background noise. The above test was performed when the victim web application was idle. In reality, any victim in the cloud might experience constant or varying background load on the system. False positives or

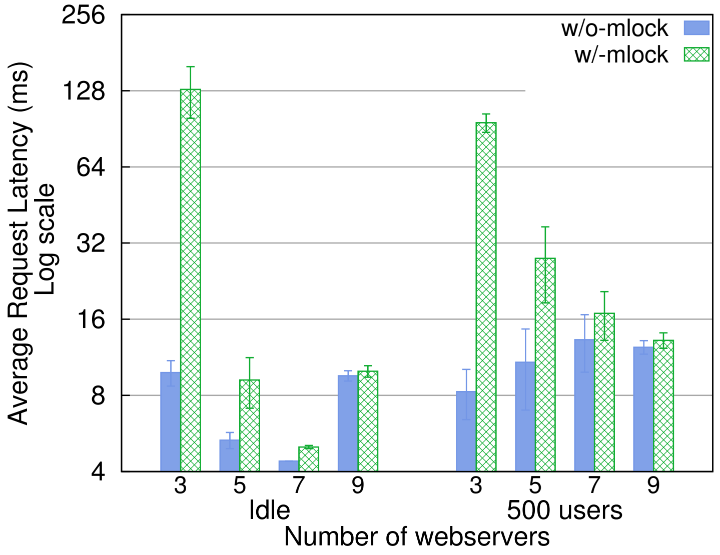


Figure 5.11: **Varying number of webservers behind the load balancer.** *The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-residency attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The error bars show the standard deviation over 5 samples.*

negatives may occur when there is spike in load on the victim servers. In such case, we use the same solution as in Section 5.2.2 — alternating between measuring the idle and the contended case.

In order to gauge the efficacy of the test under constant background load, we repeated the above experiment with varying load on the victim. The result of this experiment is summarized in Figure 5.10. Counterintuitively, we found that a constant load on the background server exacerbates the performance degradation gap, hence resulting in a clearer signal of co-residency. This is because running memory-locking on the co-resident attacker increases the service time of all requests as majority of the requests rely on memory bandwidth. This increases the queuing delay in the system and in turn increasing the overall request latency. Interestingly, this aforementioned performance gap stops widening at higher loads of 750 to 1000 concurrent users as the system hits a bottleneck (in our case a network bottleneck at the load balancer) even without running the memory-locking sender. Thus, detecting co-residency with a victim VM that is part of a highly loaded and bottlenecked application would be hard using this test.

We also experimented with increasing the number of victim webservers behind the load balancer beyond 3 (Figure 5.11). As expected, the co-residency signal grew weaker with increasing victims, and at 9 webservers, the performance degradation was too low to be useful for detecting co-residency.

5.3 Placement Vulnerability Study

In this section, we evaluate three public clouds, Amazon EC2, Google Compute Engine and Microsoft Azure, for placement vulnerabilities and answer the following questions: (i) what are all the strategies that an adversary can employ to increase the chance of co-location with one or more victim VMs? (ii) what are the chances of success and cost of each strategy? and (iii) how do these strategies compare against the reference placement policy introduced in Section 5.1?

5.3.1 Experimental Methodology

Before presenting the results, we first describe the experiment setting and methodology that we employed for this placement vulnerability study.

Experiment settings. Recall VM placement depends on several placement variables (shown in Table 5.1). We assigned reasonable values to these placement variables and enumerated through several launch strategies. A *run* corresponds to one launch strategy and involves launching multiple VMs from two distinct accounts (i.e., subscriptions in Azure and projects in GCE) and checking for co-residency between all pairs of VMs launched. One account was designated as a proxy for the victim and the other for the adversary. We denote a *run configuration* by $v \times a$, where v is the number of victim instances and a is the number of attacker instances launched in that run. We varied v and a for all v, $a \in \{10, 20, 30\}$ and restricted them to the inequality, $v \leqslant a$, as it increases the likelihood of achieving co-residency.

Other placement variables that are part of the run configuration include: victim launch time (including time of the day, day of the week), delay between victim and attacker VM launches, victim and attacker instance types and data center location or region where the VMs are launched. We repeat each run multiple times across all three cloud providers. The repetition of experiments is especially required to control the effect of certain environment variables like time of day. We repeat experiments for each run configuration over various times of the day and days of the week. We fix the instance type of VMs to small instances (t2.small on EC2, g1.small on GCE and small or Standard-A1 on Azure) and data center regions to us-east for EC2, us-central1-a for GCE and east-us for Azure, unless otherwise noted. All experiments were conducted over 3 months between December 2014 to February 2015.

For all experiments, we use a single Intel Core i7-2600 machine located at Wisconsin with 8 SMT cores to launch VM instances, log instance information and run the co-residency detection test suite unless otherwise noted.

Implementation and the Cloud APIs. In order to automate our experiments, we used Python and the libcloud[2] library [24] to interface with EC2 and GCE. Unfortunately, libcloud did not support Azure. The only Azure cloud API on Linux platform was a node.js library and a cross-platform command-line interface (CLI). We built a wrapper around the

[2]We used libcloud version 0.15.1 for EC2, and a modified version of 0.16.0 for GCE to support the use of multiple accounts in GCE.

CLI. There were no significant differences across different cloud APIs except that Azure did not have any explicit interface to launch multiple VMs simultaneously.

As mentioned in the experiment settings, we experimented with various delays between the victim and attacker VM launches (0, 1, 2, 4 …hours). To save money, we reused the same set of victim instances for each of the longer runs. That is, for the run configuration of 10x10 with 0, 1, 2, and 4 hours of delay between victim and attacker VM launches, we launched the victim VMs only once at the start of the experiment. After running co-residency tests on the first set of VM pairs, we terminated all the attacker instances and relaunched attacker VM instances after appropriate delays (say 1 hour) and rerun the tests with *the same set* of victim VMs. We repeat this until we experiment with all delays for this configuration. We call this methodology the *leap-frog method*. It is also important to note that zero delay here means parallel launch of VMs from our test machine (and not sequential launch of VMs from one account after another), unless otherwise noted.

In the sections below, we take a closer look at the effect of varying one placement variable while keeping other variables fixed across all the cloud providers. In each case, we use three metrics to measure the degree of co-residency: chances of getting at least one co-resident instance across a number of runs (or success rate), average number of co-resident instances over multiple runs and average coverage (i.e., fraction of victim VMs with which attacker VMs were co-resident). Although these experiments were done with victim VMs under our control, the results can be extrapolated to guide an attacker's launch strategy for an uncooperative victim. We also discuss a set of such strategic questions that the results help answer. At the end of this section, we summarize and calculate the cost of several interesting launch strategies and evaluate the public clouds against our reference placement policy.

Delay (hr.)	Config.	Mean	S.D.	Min	Median	Max
0	10x10	0.11	0.33	0	0	1
0	10x20	0.2	0.42	0	0	1
0	10x30	0.5	0.71	0	0	2
0	20x20	0.43	0.65	0	0	2
0	20x30	1.67	1.22	0	2	4
0	30x30	1.6	1.65	0	1	5
1	10x10	0.25	0.46	0	0	1
1	10x20	0.33	0.5	0	0	1
1	10x30	1.6	1.07	0	2	3
1	20x20	1.27	1.22	0	1	4
1	20x30	2.44	1.51	0	3	4
1	30x30	3	1.12	1	3	5

(a) *us-central1-a*

Delay (hr.)	Config.	Mean	S.D.	Min	Median	Max
0	10x10	2	1.73	1	1	4
0	10x20	2.67	1.53	1	3	4
0	10x30	3	2.65	1	2	6
0	20x20	3.67	1.53	2	4	5
0	20x30	2.75	2.06	0	3	5
0	30x30	12.33	2.08	10	13	14
1	10x10	2	1	1	2	3
1	10x20	2	1	1	2	3
1	10x30	2	1.73	1	1	4
1	20x20	4.67	5.51	1	2	11
1	20x30	3.75	2.5	1	3.5	7
1	30x30	10	3	7	10	13

(b) *europe-west1-b*

Table 5.12: **Distribution of number of co-resident pairs on GCE.**

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	$*$	0	0	0	0	0
1	10x10	0.44	0.73	0	0	2
1	10x20	1.11	1.17	0	1	3
1	10x30	1.4	1.43	0	1.5	4
1	20x20	3.57	2.59	0	3.5	9
1	20x30	3.78	1.79	1	4	7
1	30x30	3.89	2.09	2	3	9

(a) *us-east*

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	$*$	0	0	0	0	0
0	20x20	10.33	8.96	0	15	16
1	10x10	1.67	0.58	1	2	2
1	10x20	2.33	0.58	2	2	3
1	10x30	5.33	2.52	3	5	8
1	20x20	8.33	4.51	4	8	13
1	20x30	5.5	3.87	2	4.5	11
1	30x30	8.33	6.66	4	5	16

(b) *us-west-1 (CA)*

Table 5.13: **Distribution of number of co-resident pairs on EC2.**

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	10x10	15.22	19.51	0	14	64
0	10x20	3.78	4.71	0	3	14
0	10x30	4.25	6.41	0	2.5	19
0	20x20	9.67	8.43	0	8	27
0	20x30	2.38	1.51	1	2	5
0	30x30	24.57	36.54	1	6	99
1	10x10	2.78	3.87	0	1	12
1	10x20	0.78	1.2	0	0	3
1	10x30	0.75	1.39	0	0	3
1	20x20	0.67	1.66	0	0	5
1	20x30	0.86	0.9	0	1	2
1	30x30	4.71	9.89	0	1	27

Table 5.14: **Distribution of number of co-resident pairs on Azure.** *Region: East US 1.*

5.3.2 Effect of Number of Instances

In this section, we observe the placement behavior while varying the number of victim and attacker instances. Intuitively, we expect the chances of co-residency to increase with the number of attacker and victim instances.

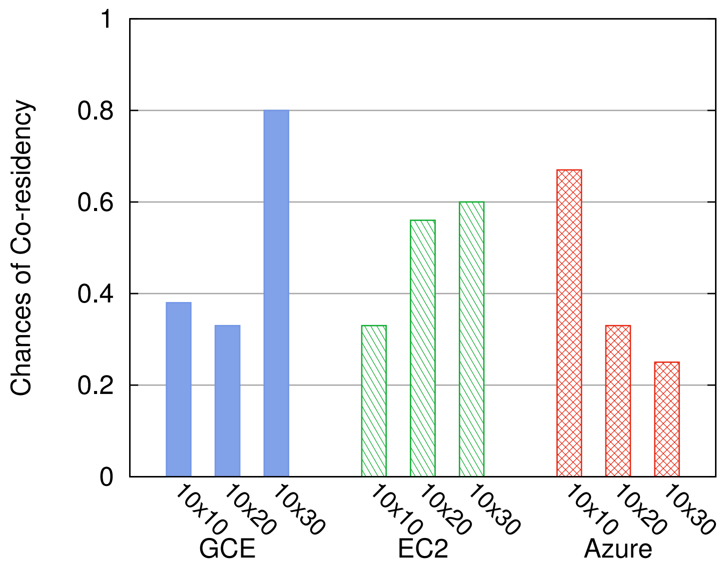


Figure 5.15: **Chances of co-residency of 10 victim instances with varying number of attacker instances.** *All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per run configuration with at least 3 runs per time of day.*

Varying number of attacker instances. Keeping all the placement variables constant including the number of victim instances, we measure the chance of co-residency over multiple runs. The result of this experiment helps to answer the question: How many VMs should an adversary launch to increase the chance of co-residency?

As is shown in Figure 5.15, the placement behavior changes across different cloud providers. For GCE and EC2, we observe that higher the number of attacker instances relative to the victim instances, the higher the chance of co-residency is. Table 5.12a and 5.13a show the distribution of number of co-resident VM pairs on GCE and EC2, respectively. The number of co-resident VM pairs also increases with the number of attacker instances, implying that the coverage of an attack could be increased with larger fraction of attacker instances than the target VM instances if the launch times are coordinated.

Contrary to our expectations, the placement behavior observed on Azure is the inverse. The chances of co-residency with 10 attacker instances are almost twice as high as with 30 attacker instances. This is also reflected in the distribution of number of co-residency VM pairs (shown in Table 5.14). Further investigation revealed a correlation between the number of victim and attacker instances launched and the chance of co-residency. That is, for the run configuration of 10x10, 20x20 and 30x30, where number of victim and attacker instances are the same, and with 0 delay, the chance of co-residency were equally high for all these configurations (between 0.9 to 1). This suggests a possible placement policy that collates VM launch requests together based on their request size and places them on the same group of machines.

Varying number of victim instances. Similarly, we also varied the number of victim instances by keeping the number of attacker instances and other placement variables constant (results shown in Figure 5.16). We expect the chance of co-residency to increase with the number of victims targeted. Hence, the results presented here help an adversary answer the question: What are the chances of co-residency with varying sizes of target victims?

As expected, we see an increase in the chances of co-residency with increasing number of victim VMs across all cloud providers. We see that the absolute value of the chance of co-residency is lower for Azure than other clouds. This may be the result of significant additional delay between victim and attacker launch times in Azure as a result of our methodology (more on this later).

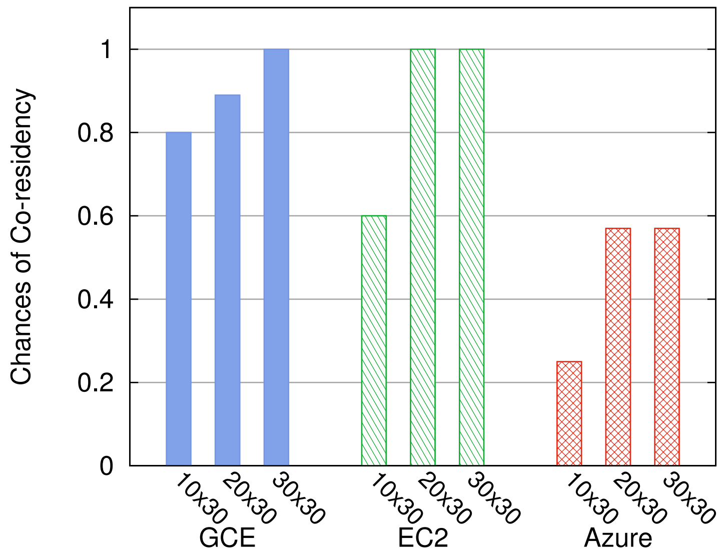


Figure 5.16: **Chances of co-residency of 30 attacker instances with varying number of victim instances.** *All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per run configuration with at least 3 runs per time of day.*

5.3.3 Effect of Instance Launch Time

In this section, we answer two questions that aid an adversary to design better launch strategies: How quickly should an attacker launch VMs after the victim VMs are launched? Is there any increase in chance associated with the time of day of the launch?

Varying delay between attacker and victim launches. The result of varying the delay between 0 (i.e., parallel launch) and 1 hour delay is shown in Figure 5.17. We can make two immediate observations from this result.

The first observation reveals a significant artifact of EC2's placement policy: VMs launched within a short time window are never co-resident on the same machine. This observation helps an adversary to avoid such a strategy. We further investigated placement behaviors on EC2 with

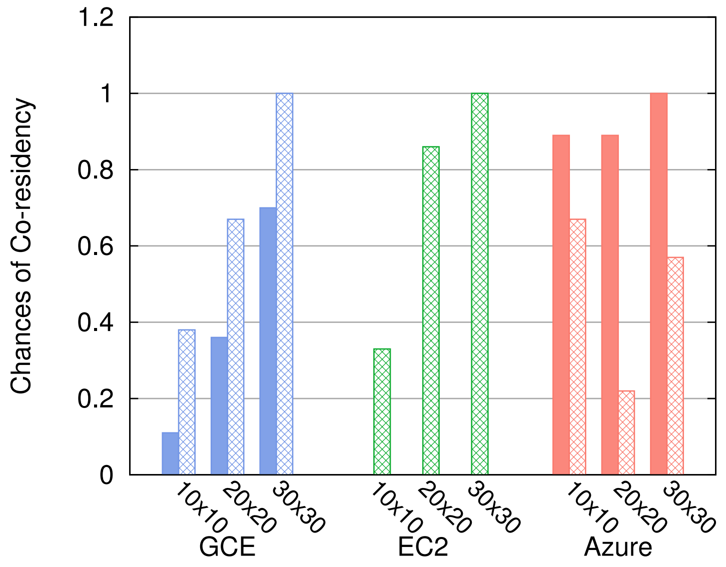


Figure 5.17: **Chances of co-residency with varying delays between victim and attacker launches.** *Solid boxes correspond to zero delay (simultaneous launches) and gauze-like boxes correspond to 1 hour delay between victim and attacker launches. We did not observe any co-resident instances for runs with zero delay on EC2. All these results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US). Results are over at least 9 runs per run configuration with at least 3 runs per time of day.*

shorter non-zero delays in order to find the duration of this time window in which there are zero co-residency (results shown in Table 5.18). We found that this time window is very short and that even a sequential launch of instances (denoted by 0+) could result in co-residency.

The second observation shows that non-zero delay on GCE and zero delay on Azure increases the chance of co-residency and hence directly benefits an attacker. It should be noted that on Azure, the launch delays between victim and attacker instances were longer than 1 hour due to our leap-frog experimental methodology; the actual delays between the VM launches were, on average, 3 hours (with a maximum delay of 10 hours for few runs). This higher delay was more common in runs with larger

Delay	Mean	S.D.	Min	Median	Max	Success rate
0+	0.6	1.07	0	0	3	0.30
5 min	1.38	0.92	0	1	3	0.88
1 hr	3.57	2.59	0	3.5	9	0.86

Table 5.18: **Distribution of number of co-resident pairs and success rate or chances of co-residency for shorter delays under 20x20 run configuration in EC2.** *A delay with 0+ means victim and attacker instances were launched sequentially, i.e. attacker instances were not launched until all victim instances were running. The results averaged are over 9 runs with 3 runs per time of day.*

number of instances as there were significantly more false positives, which required a separate sequential phase to resolve (see Section 5.2.2).

We also experimented with longer delays on EC2 and GCE to understand whether and how quickly the chance of co-residency drops with

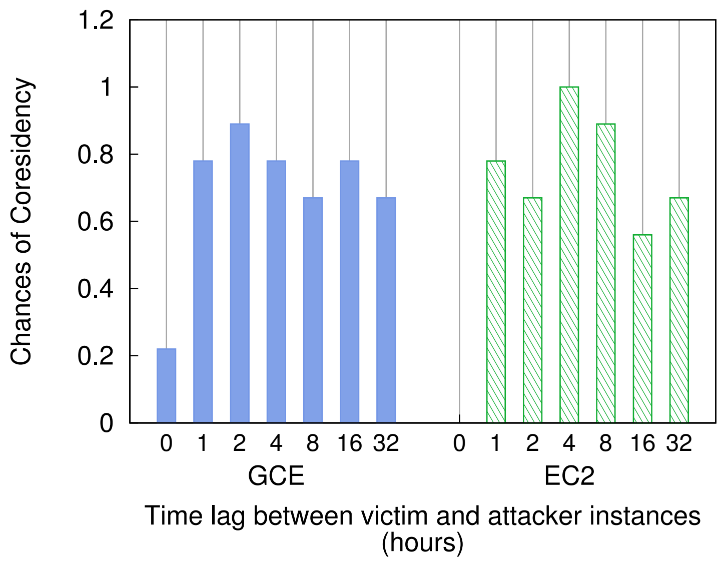


Figure 5.19: **Chances of co-residency over long periods.** *Results include 9 runs over two weeks with 3 runs per time of day under 20x20 run configuration. Note that we only conducted 3 runs for 32 hour delay as opposed to 9 runs for all other delays.*

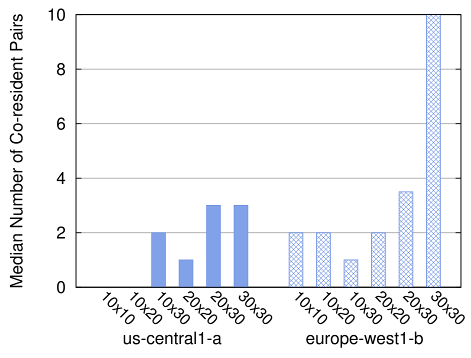
Chances of Co-residency			
Cloud	**Morning** 02:00 - 10:00	**Afternoon** 10:00 - 18:00	**Night** 18:00 - 02:00
GCE	0.68	0.61	0.78
EC2	0.89	0.73	0.6

Table 5.20: **Effect of time of day.** *Chances of co-residency when an attacker changes the launch time of his instances. The results were aggregated across all run configurations with 1 hour delay between victim and attacker launch times. All times are in PT.*

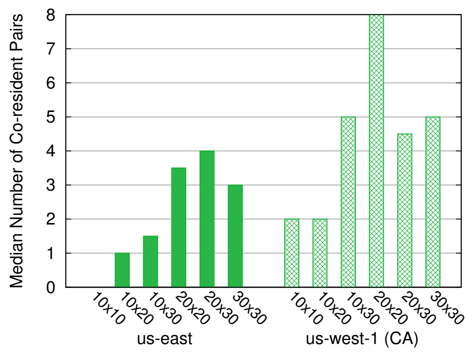
increasing delay (results shown in Figure 5.19). Contrary to our expectation, we did not find the chance of co-residency to drop to zero even for delays as high as 16 and 32 hours. We speculate that the reason for this observation could be that the system was under constant churn where some neighboring VMs on the victim's machine were terminated. Note that our leap-frog methodology may, in theory, interfere with the VM placement. But it is noteworthy that we observed increased number of unique co-resident pairs with increasing delays, suggesting fresh co-residency with victim VMs over longer delays.

Effect of time of day. Prior works have shown that churn or load is often correlated with the time of day [178]. Our simple reference placement policy does not have a notion of load and hence have no effect on time of day. In reality, with limited number of servers in datacenters and limited number of capacity per host, load on the system has direct effect on the placement behavior of any placement policy.

As expected, we observe small effect on VM placement based on the time of day when attacker instances are launched (results shown in Table 5.20). Specifically, there is a slightly higher chance of co-residency if the attacker instances are launched in the early morning for EC2 and at night for GCE.



a: *GCE*



b: *EC2*

Figure 5.21: **Median number of co-resident pairs across two regions.** *The box plot shows the median number of co-resident pairs excluding co-residency within the same account. Results are over at least 3 run per run configuration (x-axis).*

5.3.4 Effect of Data Center Location

All the above experiments were conducted on relatively popular regions in each cloud (especially true for EC2 [178]). In this section, we report the results on other smaller and less popular regions. As the regions are less popular and have relatively fewer machines, we expect higher co-residency rates and more co-resident instances. Figure 5.21 shows the median number of co-resident VM pairs placed in these regions alongside the results for popular regions. The distribution of number of co-resident instances is shown in Table 5.12b and 5.13b.

The main observation from these experiments is that there is a higher chance of co-residency in these smaller regions than the larger, more popular regions. Note that we placed at least one co-resident pair in all the runs in these regions. Also the higher number of co-resident pairs also suggests a larger coverage over victim VMs in these smaller regions.

One anomaly that we found during two 20x20 runs on EC2 between 30th and 31st of January 2015, when we observed an unusually large number of co-resident instances (including three VMs from the same account). We believe this anomaly may be a result of an internal management incident in the Amazon EC2 us-west-1 region.

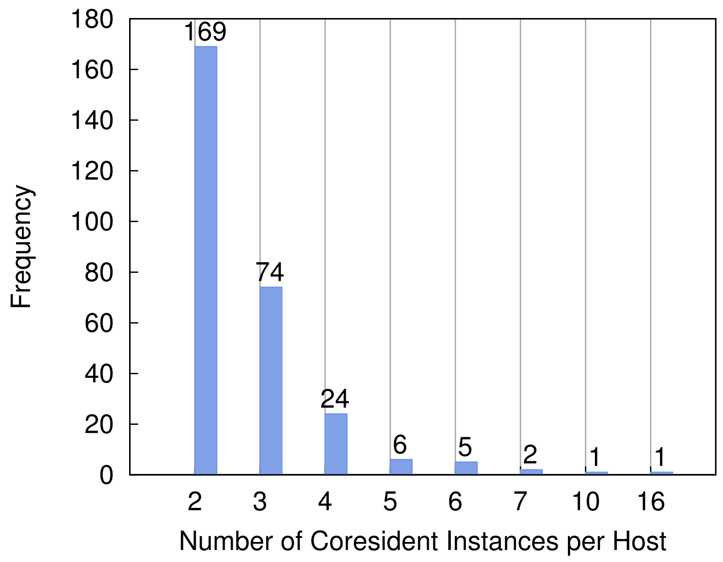


Figure 5.22: **Distribution of number of co-resident instances per host on Azure.** *The results shown are across all the runs. We saw at most 2 instances per host in EC2 and at most 3 instances per host in GCE.*

5.3.5 Effect of Launch Origin Machine

Until now, we simulated launch characteristics of a victim and an attacker by executing a launch strategy from a single machine (launch origin machine). It is possible that the location or the network address of the launch origin machine may influence co-location even when the VM launches are requested two different account. Although it is not clear what is the goal of such a placement policy, it is useful to understand whether the origin of launch requests is an essential criteria for a successful co-location. To verify this, we repeated one of the successful strategies – 30x30 with one hour delay – on Amazon EC2. But instead of launching VMs from the same workstation at Wisconsin, we launched VMs under each account from machines at different locations within the US. For the victim, we initiated the launch from a remote virtual machine in GCE in the us-east1-b region[3]. For the attacker, we used the same workstation at Wisconsin (central US). With three runs at three different times of day on a week day, we achieved co-location in 2 out of the 3 runs with a maximum of 3 co-resident pairs in one run. With the results gathered from previous experiments (conducted between December 2014 to February 2015), we expected all the three runs to result in at least one pair of co-resident instances between the attacker and the victim. We believe there are two reasons that we did not achieve co-location in all the three runs: 1) this new experiment was conducted in December 2015 almost one year after the first set of experiments and hence may be a result of any changes to the placement policy used by EC2, 2) Both set of experiments have their limitations. For example, they involve only a small number of runs (3 or 9). Nevertheless, the fact that we observe some co-resident instances is sufficient to show that a single launch origin machine is not a necessary criteria for successful co-location in Amazon EC2.

5.3.6 Other Observations

We report several other interesting observations in this section. First, we found more than two VMs can be co-resident on the same host on both Azure and GCE, but not on EC2. Figure 5.22 shows the distribution of number of co-resident instances per host. Particularly, in one of the runs, we placed 16 VMs on a single host.

Another interesting observation is related to co-resident instances from the same account. We term them as *self-co-resident instances*. We observed many self-co-resident pairs on GCE and Azure (not shown). On the other

[3]This is one of the new regions recently introduced in GCE at the time of this experiment in December 2015.

hand, we never noticed any self co-resident pair on EC2 except for the anomaly in us-west-1. Although we did not notice any effect on the actual chance of co-residence, we believe such placement behaviors (or the lack of) may affect VM placement.

We also experimented with medium instances and successfully placed few co-located VMs on both EC2 and GCE by employing similar successful strategies learned with small instances.

5.3.7 Cost of Launch Strategies

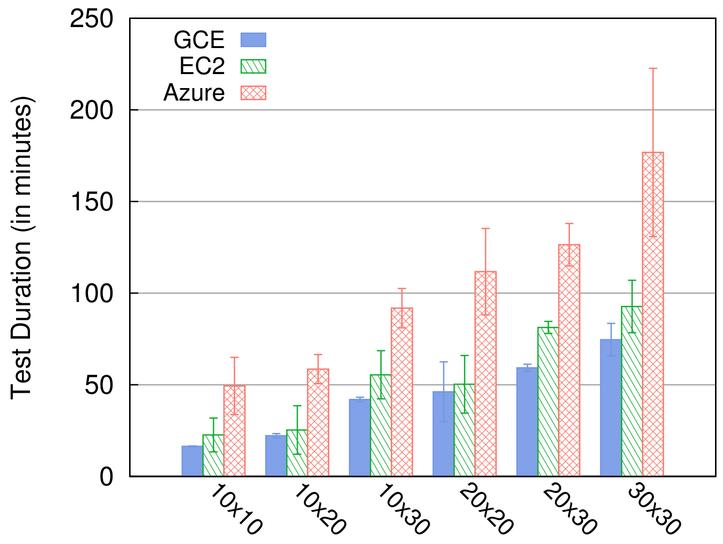


Figure 5.23: **Launch strategy and co-residency detection execution times.** *The run configurations $v \times a$ indicates the number of victims vs. number of attackers launched. The error bars show the standard deviation across at least 7 runs.*

Recall that the cost of a launch strategy from Section 5.1, $C_S = a * P(a_{type}) * T_d(v, a)$. In order to calculate this cost, we need $T_d(v, a)$ which is the time taken to detect co-location with a attackers and v victims. Figure 5.23 shows the average time taken to complete launching attacker

Run	**Average Cost ($)**			**Maximum Cost ($)**		
config.	GCE	EC2	Azure	GCE	EC2	Azure
10x10	**0.137**	0.260	0.494	0.140	0.260	0.819
10x20	0.370	0.520	1.171	0.412	0.520	1.358
10x30	1.049	0.780	2.754	1.088	1.560	3.257
20x20	0.770	0.520	2.235	1.595	1.040	3.255
20x30	1.482	1.560	3.792	1.581	1.560	4.420
30x30	1.866	1.560	5.304	2.433	1.560	**7.965**

Table 5.24: **Cost of running a launch strategy.** *Maximum cost column refers to the maximum cost we incurred out of all the runs for that particular configuration and cloud provider (in dollars). The cost per hour of small instances at the time of this study were: 0.05, 0.026 and 0.06 dollars for GCE, EC2 and Azure, respectively. The minimum and maximum costs are in **bold**.*

instances and complete co-residency detection for each run configuration. Here the measured co-residency detection is the parallelized version discussed in Section 5.2.2 and also includes time taken to detect co-residency within each tenant account. Hence, for these reasons the time to detect co-location is an upper bound for a realistic and highly optimized co-residency detection mechanism.

We calculate the cost of executing each launch strategy under the three public clouds. The result is summarized in Table 5.24. Note that we only consider the cost incurred by the compute instances because the cost for other resources such as network and storage, was insignificant. Also note that EC2 bills every hour even if an instance runs less than an hour [14], whereas GCE and Azure charge per minute of instance activity. This difference is considered in our cost calculation. Overall, the maximum cost we incurred was about $8 for running 30 VMs for 4 hours 25 minutes on Azure and a minimum of 14 cents on GCE for running 10 VMs for 17 minutes. We incurred the highest cost for all the launch strategies in Azure because of overall higher cost per hour and partly due to longer tests due to our co-residency detection methodology.

Run Config.	10x10	10x20	10x30	20x20	20x30	30x30
$\Pr[\,\mathbf{E}_a^v > 0\,]$	0.10	0.18	0.26	0.33	0.45	0.60

Table 5.25: **Probability of co-residency under the reference placement policy.**

5.4 Summary of Placement Vulnerabilities

In this section, we return to the secure reference placement policy introduced in Section 5.1 and use it to identify placement vulnerabilities across all the three clouds. Recall that the probability of at least one pair of co-residency under this random placement policy is given by $\Pr[\,\mathbf{E}_a^v > 0\,] = 1 - (1 - v/N)^a$, where $\mathbf{E}_a^v$ is the random variable denoting the number of co-location observed when placing a attacker VMs among $N = 1000$ total machines where v machines are already picked for the v victim VMs. First, we evaluate this probability for various run configurations that we experimented with in the public clouds. The probabilities are shown in Table 5.25.

Recall that a launch strategy in a cloud implies a placement vulnerability in that cloud's placement policy if its normalized success rate is greater than 1. The normalized success rate of the strategy is the ratio of the chance of co-location under that launch strategy to the probability of co-location in the reference policy ($\Pr[\,\mathbf{E}_a^v > 0\,]$). Below is a list of selected launch strategies that escalate to placement vulnerabilities using our reference policy with their normalized success rate in parenthesis.

(S1) In Azure, launch ten attacker VMs closely after the victim VMs are launched (1.0/0.10).

(S2) In EC2 and GCE, if there are known victims in any of the smaller datacenters, launch at least ten attacker VMs with a non-zero delay (1.0/0.10).

(S3) In all three clouds, launch 30 attacker instances, either with no delay

Strategy	ν & a	a'	**Cost benefit** ($)	**Normalized success**
S1 & S2	10	688	113.87	10
S3	30	227	32.75	1.67
S4(i)	20	105	4.36	2.67
S4(ii)	20	342	53.76	3.03
S5	20	110	4.83	1.48

Table 5.26: **Cost benefit analysis.** $N = 1000$, $P(a_{type}) = 0.026$, *which is the cost per instance-hour on EC2 (the cheapest). For simplicity* $T_d(\nu, a) = (\nu * a) * 3.85$, *where* 3.85 *is fastest average time to detect co-residency per instance-pair. Here,* $\nu \times a$ *is the run configuration of the strategy under test. Note that the cost benefit is the additional cost incurred under the reference policy, hence is equal to cost incurred by* $a' - a$ *additional VMs.*

(Azure) or one hour delay (EC2, GCE) from victim launch, to get co-located with one of the 30 victim instances (1.00/0.60).

(S4) (i) In Amazon EC2, launch 20 attacker VMs with a delay of 5 minutes or more after the victims are launched (0.88/0.33). (ii) The optimal delay between victim and attacker VM launches is around 4 hours for a 20x20 run (1.00/0.33).

(S5) In Amazon EC2, launch the attacker VMs with 1 hour after the victim VMs are launched where the time of day falls in the early morning, i.e., 02:00 to 10:00hrs PST (0.89/0.60).

Cost benefit. Next, we quantify the cost benefit of each of these strategies over the reference policy. As the success rate of any launch strategy on a vulnerable placement policy is greater than what is possible in the reference policy, we need more attacker instances in the reference policy to achieve the same success rate. We calculate this number of attacker instances a' using: $a' = \ln(1 - S_a^\nu)/\ln(1 - \nu/N)$, where, S_a^ν is the success rate of a strategy with run configuration of $\nu \times a$. The result of this calculation is presented in Table 5.26. The result shows that the best strategy, S1 and S2, on all three cloud providers is $114 cheaper than what is possible in the reference policy.

It is also evident that these metrics enable evaluating and comparing various launch strategies and their efficacy on various placement policies both on robust placements and attack cost. For example, note that although the normalized success rate of *S3* is lower than *S4*, it has a higher cost benefit for the attacker.

5.4.1 Limitations

Although we exhaustively experimented with a variety of placement variables, the results have limitations. One major limitation of this study is the number of placement variables and the set of values for the variables that we used to experiment. For example, we limited our experiments with only one instance type, one availability zone per region and used only one account for the victim VMs. Although different instance types may exhibit different placement behavior, the presented results hold strong for the chosen instance type. The only caveat that may affect the results is a placement policy that uses account ID for VM placement decisions. Since, we experimented with only one victim account (separate from the designated attacker account) across all providers our experiments, in the worst case, may have captured such a placement behavior that resulted in similar placement decisions for VMs from these two accounts. Given that we found similar results of co-location on three distinct public clouds such a worst-case scenario seems extremely unlikely.

Even though we ran at least 190 runs per cloud provider over a period of 3 months to increase statistical significant of our results, we were still limited to at most 9 runs per run configuration (with 3 runs per time of day). These limitations have only minor bearing on the results presented, if any, and the reported results are significant and impactful for cloud computing security research.

5.5 Placement Vulnerability in PaaS

While we mainly studied placement vulnerabilities in the context of IaaS, we also experimented with Platform-as-a-Service (PaaS) clouds. PaaS clouds offer elastic application hosting services. Unlike IaaS where users are granted full control of a VM, PaaS provides managed compute tasks (or instances) for the execution of hosted web applications, and allow multiple such instances to share the same operating system. These clouds use either process-level isolation via file system access controls, or increasingly Linux-style containers (see [192] for a more detailed description). As such, logical side-channels alone are usually sufficient for co-residency detection purposes. For instance, in PaaS clouds, co-resident instances often share the same public IP address as the host machine. This is because the host-to-instance network is often configured using Network Address Translation (NAT) and each instance is assigned a unique port under the host IP address for incoming connections.

We found that many such logical side-channel-based co-residency detection approaches worked on PaaS clouds, even on those using containers. Specifically, we used both system-level interrupt statistics via `/proc/interrupts` and shared public IP addresses of the instances to detect co-location in Heroku [87]. Note that both these techniques either require direct access to victim instances, or a software vulnerability to access `procfs` or initiate reverse connections, respectively.

Our brief investigation of co-location attacks in Heroku [86] showed that naïve strategies like scaling two PaaS web applications to 30 instances with a time interval of 5 minutes between them, resulted in co-location in 6 out of 10 attempts. Moreover, since the co-location detection was simple and quick including the time taken for application scaling, we were able to do these experiments free of cost. This result reinforces prior findings on PaaS co-location attacks [192] and confirms the existence of cheap launch strategies to achieve co-location and easy detection mechanisms to verify it. We do not investigate PaaS clouds further in the rest of this dissertation.

6 Stealing Performance from Neighboring VMs

In Chapter 4.2 we witnessed the extent of performance interference between VMs because of the lack of performance isolation. In this chapter, we will look at how a malicious user could abuse this lack of isolation for his own performance by using an attack we call Resource-Freeing Attack.

We explore an approach based on two observations. First, applications are often limited by a single bottleneck resource, such as memory or network bandwidth. Second, we observe that an application's use of resources can change unevenly based on workload. For example, a web server may be network limited when serving static content, but CPU limited when serving dynamic content.

A *resource-freeing attack* (RFA) leverages these observations to improve a VM's performance by forcing a competing VM to saturate some bottleneck. If done carefully, this can slow down or shift the competing application's use of a desired resource. For example, we investigate in detail an RFA that improves cache performance when co-resident with a heavily used Apache web server. Greedy users will benefit from running the RFA, and the victim ends up paying for increased load and the costs of reduced legitimate traffic.

We begin this work with a comprehensive study of the resource interference exhibited by the Xen hypervisor in our local testbed (§ 6.2). In addition to testing for contention of a single resource, these results show that workloads using different resources can contend as well, and that scheduling choices on multicore processors greatly affect the performance loss. We then develop a proof-of-concept resource-freeing attack for the cache-network contention scenario described above (§ 6.3). In a controlled environment, we determine the necessary conditions for a successful resource-freeing attack, and show that average performance of a cache-sensitive benchmark can be improved by as much as 212% when the two VMs always share a single core, highlighting the potential for RFAs to ease cache contention for the attacker. If VMs float among all cores (the default configuration in Xen), we still see performance gains of up to 60%. When applied to several SPEC benchmarks [84], whose more balanced workloads are less effected by cache contention, RFAs still provide benefit: in one case it reduces the effect of contention by 66.5% which translated to a 6% performance improvement.

Finally, we show that resource-freeing attacks are possible in uncontrolled settings by demonstrating their use on Amazon's EC2 (§ 6.3.2). Using co-resident virtual machines launched under accounts we control, we show that introducing additional workload on one virtual machine can improve the performance of our cache-sensitive benchmark by up to 13% and provides speedups for several SPEC benchmarks as well.

6.1 Resource-freeing Attacks

The interference encountered between VMs on public clouds motivates a new class of attacks, which we call resource-freeing attacks (RFAs). The general idea of an RFA is that when a guest virtual machine suffers due to performance interference, it can affect the workload of other VMs on the same physical server in a way that improves its own performance.

Attack setting. We consider a setting in which an *attacker* VM and one or more *victim* VMs are co-resident on the same physical server in a public cloud. There may be additional co-resident VMs as well. It is well known that public clouds make extensive use of multi-tenancy.

The RFAs we consider in Section 6.3 assume that the victim is running a public network service, such as a web server. This is a frequent occurrence in public clouds. Measurements in 2009 showed that approximately 25% of IP addresses in one portion of EC2's address space hosted a publicly accessible web server [151].

Launching RFAs that exploit a public network service require that the attacker knows with whom it is co-resident. On many clouds this is straightforward: the attacker can scan nearby internal IP addresses on appropriate ports to see if there exist public network services. This was shown to work in Amazon EC2, where for example m1.small co-resident instances had internal IP addresses whose numerical distance from an attacker's internal IP address was at most eight [151]. Furthermore, packet round-trip times can be used to verify co-residence. We expect that similar techniques work on other clouds, such as Rackspace.

The attacker seeks to interfere with the victim(s) to ease contention for resources on the node or nearby network. The attacker consists of two logical components, a *beneficiary* and a *helper*. The beneficiary is the application whose efficiency the attacker seeks to improve. The helper is a process, either running from within the same instance or on another machine, that the attacker will use to introduce new workload on the victim. Without loss of generality, we will describe attacks in terms of one victim, one beneficiary, and one helper.

We assume the beneficiary's performance is reduced because of interference on a single contended resource, termed the *target resource*. For example, a disk-bound beneficiary may suffer from competing disk accesses from victim VMs.

Conceptual framework. The beneficiary and the helper work together to change the victim's resource consumption in a manner that frees up the target resource. This is done by increasing the time the victim spends on one portion of its workload, which limits its use of other resources.

There are two requirements for an RFA. First, an RFA must raise the victim's usage of one resource until it reaches a *bottleneck*. Once in a bottleneck, the victim cannot increase usage of any resources because of the bottleneck. For example, once a web server saturates the network, it cannot use any more CPU or disk bandwidth. However, simply raising the victim to a bottleneck does not free resources; it just prevents additional use of them. The second requirement of an RFA is to *shift* the victim's resource usage so that a greater fraction of time is spent on the bottleneck resource, which prevents spending time on other resources. Thus, the bottleneck resource crowds out other resource usage. As an example, a web server may be sent requests for low-popularity web pages that cause random disk accesses. The latency of these requests may crowd requests for popular pages and overall reduce the CPU usage of the server.

There are two shifts in target resource usage that can help the beneficiary. First, if the victim is forced to use less of the resource, then there may be more available for the beneficiary. Second, even if the victim uses the same amount of the resource, the accesses may be shifted in time. For example, shifting a victim's workload so that cache accesses are consolidated into fewer, longer periods can aid the beneficiary by ensuring it retains cache contents for a larger percentage of its run time. A similar effect could be achieved for resources like the hard disk if we are able to provide the beneficiary with longer periods of uninterrupted sequential accesses.

Modifying resource consumption. The helper modifies the victim's resource usage and pushes it to overload a bottleneck resource. This can be done externally, by introducing new work over the network, or internally, by increasing contention for other shared resources.

A helper may introduce additional load to a server that both increases its total load and skews its workload towards a particular resource. The example above of requesting unpopular content skews a web server's resource usage away from the CPU towards the disk. This can create a bottleneck at either the server's connection limit or disk bandwidth. Similarly, the helper may submit CPU-intensive requests for dynamic data that drive up the server's CPU usage until it exceeds its credit limit and is preempted by the hypervisor.

The helper can also affect performance by increasing the load on other contended resources. Consider again a web server that makes use of the disk to fetch content. A helper running in the beneficiary's instance can introduce unnecessary disk requests in order to degrade the victim's disk performance and cause the disk to become a bottleneck. Similarly, the helper could slow the victim by introducing additional network traffic that makes network bandwidth a bottleneck for the server.

There exist some obvious ways an attacker might modify the workload of a victim. If the attacker knows how to remotely crash the victim via some exploitable vulnerability, then the helper can quite directly free up the target resource (among others). However this is not only noisy, but requires a known vulnerability. Instead, we focus on the case that the attacker can affect the victim only through use (or abuse) of legitimate APIs.

Example RFA. As a simple example of an RFA, we look at the setting of two web servers, running in separate VMs on the same physical node, that compete for network bandwidth. Assume they both serve a mix of static and dynamic content. Under similar loads, a work-conserving network scheduler will fairly share network capacity and give each web server 50% (indeed, our experiment show that Xen does fairly share network bandwidth).

However, if we introduce CPU-intensive requests for dynamic content to one web server that saturate the CPU time available to the server, we find that the other server's share of the network increases from 50% to 85%, because there is now less competing traffic. We note that this requires a work-conserving scheduler that splits excess network capacity across the VMs requesting it. A non-work conserving scheduler would cap the bandwidth available to each VM, and thus a decline in the use by one VM would not increase the bandwidth available to others.

6.2 Contention Measurements

In order to understand which resources are amenable to resource-freeing attacks in a Xen environment, we created a local testbed that attempts to duplicate a typical configuration found in EC2 (in particular, the m1.small instance type).

Testbed. Although Amazon does not make their precise hardware configurations public, we can still gain some insight into the hardware on which an instance is running by looking at system files and the CPUID instruction. Based on this, we use a platform consisting of a 4-core, 2-package 2.66 GHz Intel Xeon E5430 with 6MB of shared L2 cache per package and 4GB of main memory. This is representative of some of the architectures used by EC2.

We install Xen on the testbed, using the configurations shown in Table 6.1. Again, while we do not have precise knowledge of Amazon's setup for Xen, our configuration approximates the EC2 m1.small instance.

This configuration allows us to precisely control the workload by varying scheduling policies and by fixing workloads to different cores. In addition, it enables us to obtain internal statistics from Xen, such as traces of scheduling activities.

Table 6.2 describes the workloads we use for stressing different hardware resources. The workloads run in a virtual machine with one VCPU.

Xen Version	4.1.1
Xen Scheduler	Credit Scheduler 1
OS	Fedora 15, Linux 2.6.40.6-0.fc15
Dom0	4 VCPU / 6 GB memory / no cap / weight 512
DomU	8 instances each with 1 VCPU / 1 GB memory / 40% cap / weight 256
Network	Bridging via *Dom0*
Disk	5 GB LVM disk partition of a single large disk separated by 150GB

Table 6.1: **Xen configuration in our local testbed..**

In order to understand the impact of sharing a cache, we execute the workloads in three scenarios:

(i) *Same core* time slices two VMs on a single core, which shares all levels of processor cache.

(ii) *Same package* runs two VMs each pinned to a separate core on a single package, which shares only the last-level cache.

(iii) *Different package* runs two VMs floating over cores on different packages, which do not share any cache, but do share bandwidth to memory.

In addition, Xen uses a separate VM named *Dom0* to run device drivers. In accordance with usage guides, we provision *Dom0* with four VCPUs. As past work has shown this VM can cause contention [79, 190], we make it execute on a different package for the first two configurations and allow it to use all four cores (both cores in both packages) for the third.

Extent of Resource Contention. The goal of our experiments is to determine the contention between workloads using different hardware resources and determine whether enough contention exists to mount an

Workload	**Description**
CPU	Solving the N-queens problem for N = 14.
Net	Lightweight web server hosting 32KB static web pages cached in memory, 5000 requests per second from a separate client.
Diskrand	Requests for randomly selected 4KB chunk in 1 GB span.
Memrand	Randomly request 4B from every 64B of data from a 64MB buffer.
LLC	Execute LLCProbe, which sequentially requests 4B from every 64B of data within an LLC-sized buffer using cache coloring to balance access across cache sets.

Table 6.2: **Resource-specific workloads used to test contention..**

RFA. With perfect isolation, performance should remain unchanged no matter what competing benchmarks run. However, if the isolation is not perfect, then we may see performance degradation, and thus may be able to successfully mount an RFA.

Table 6.3 provides tables showing the results, which demonstrate that Xen is not able to completely isolate the performance of any resource. Across all three configurations, CPU and Memrand show the least interference, indicating that Xen does a good job accounting for CPU usage and that the processor limits contention for memory bandwidth.

However, for all other resources, there are competing workloads that substantially degrade performance. The two resources suffering the worst contention are Diskrand where run time increases 455% with contending random disk access; and LLC, where run time increases over 500% with Net and over 500% with Memrand. for Diskrand, competing disk traffic causes seeks to be much longer and hence slower. For LLC, competing workloads either interrupt frequently (Net) or move a lot of data through the cache (Memrand).

Same core	CPU	Net	Diskrand	Memrand	LLC
CPU	-	5	-	-	-
Net	-	194	-	-	-
Diskrand	-	-	455	-	-
Memrand	-	6	-	-	-
LLC	8	539	72	38	34
Same package	CPU	Net	Diskrand	Memrand	LLC
CPU	-	-	-	-	-
Net	-	198	-	-	-
Diskrand	-	-	461	-	-
Memrand	-	-	17	-	-
LLC	20	448	55	566	566
Diff. package	CPU	Net	Diskrand	Memrand	LLC
CPU	-	20	-	-	-
Net	-	100	-	-	-
Diskrand	-	-	462	-	-
Memrand	-	35	-	-	-
LLC	6	699	11	15	15

Table 6.3: **Resource Contention in Xen..** *Percentage increase in workload run times indicated in row when contending with workload indicated in column. Percentage is computed as run time with contention over run time on otherwise idle machine. For network, run time is the time to serve a fixed number of requests. A dash means there was no significant performance degradation. (Top) The VMs are pinned to the same core. (Middle) The VMs are pinned to different cores on the same package. (Bottom) The VMs are pinned to different packages.*

The three configurations differ mostly in the LLC results. In the same-core and different-package configurations, the contention with LLC is fairly small. On the same core, the conflicting code does not run concurrently, so performance is lost only after a context switch. On different packages, performance losses come largely from *Dom0* , which is spread across all cores. In the same-package configuration, though, the tests execute concurrently and thus one program may displace data while the other is running.

One pair of resources stands out as the worst case across all configurations: the degradation caused by Net on LLC. This occurs for three reasons: (i) the HTTP requests cause frequent interrupts and hence frequent preemptions due to boost; (ii) in the same-core and same-package configurations the web server itself runs frequently and displaces cache contents; and (iii) *Dom0* runs the NIC device driver in the different-package configuration. We will therefore focus our investigation of RFAs on the conflict between such workloads, and leave exploration of RFAs for other workload combinations to future work.

6.3 RFA for Cache versus Network

As we saw, a particularly egregious performance loss is felt by cache-bound workloads when co-resident with a network server. Unfortunately, co-residence of such workloads seems a likely scenario in public clouds: network servers are a canonical application (EC2 alone hosts several million websites [134]) while cache-bound processes abound. The remainder of the work seeks to understand whether a greedy customer can mount an RFA to increase performance when co-resident with one or more web servers.

Setting. We start by providing a full description of the setting on which we focus. The beneficiary is a cache bound program running alone in a VM with one VCPU. We use the LLCProbe benchmark as stand-in for a real beneficiary. LLCProbe is intentionally a synthetic benchmark and is designed to expose idealized worst-case behavior. Nevertheless, Its pointer-chasing behavior is reflected in real workloads [76]. We will also investigate more balanced benchmarks such as SPEC CPU2006 [84], SPECjbb2005 [161] and graph500 [76].

In addition to the beneficiary, there is a victim VM co-resident on the same physical machine running the Apache web server (version 2.2.22). It is configured to serve a mix of static and dynamic content. The static content consists of $4,096$ 32KB web pages (enough to overlow the 6MB LLC) containing random bytes. The dynamic content is a CGI script that can be configured to consume varying amounts of CPU time via busy looping. This script serves as a stand in for either an actual web server serving dynamic content on the web, or the effects of DoS attacks that drive up CPU usage, such as complexity attacks [56, 60]. The script takes a parameter to control duration of the attack, and spins until wall-clock time advances that duration. We note that this does not reflect the behavior of most DoS attacks, which take a fixed number of cycles, but we use it to provide better control over the web server's behavior. We confirmed that the behaviors exhibited also arise with CGI scripts performing a fixed number of computations.

The Apache server is configured with the *mod_mem_cache* module to reduce the latency of static content and FastCGI to pre-fork a process for CGI scripts. We also use the Multi-Processing Module for workers, which is a hybrid multithreaded multi-process Apache web server design used for better performance and for handling larger request loads.

To simulate load on the web server, we use a custom-built multi-threaded *load generator* that sends web requests for the static content hosted by the victim. Each client thread in the load generator randomly selects a static web page to request from the web server. The load generator includes a rate controller thread that ensures that the actual load on the web server does not exceed the specified request rate. The client uses 32 worker threads, which we empirically determined is enough to sustain the web server's maximum rate. Requests are synchronous and hence the load generator waits for the response to the previous request and then a timeout (to prevent sending requests too fast) before sending the next request. Since each thread in the load generator waits for a response from the web server before sending the next request, it may not meet the specified request rate if the server or the network bandwidth cannot sustain the load. The helper, which performs the actual RFA, is identical to the load generator except that it sends requests for the CGI script rather than for static pages.

Understanding the contention. We conduct experiments on our local testbed to understand the basic performance degradation experienced by LLCProbe as the web server's workload varies. We report the average time to probe the cache; one probe involves accessing every cacheline out of a buffer of size equal to the LLC. We measure the time per probe by counting the number of probes completed in 10 seconds.

To understand contention, we first pin the victim VM and the beneficiary VM to the same core and pin *Dom0* to a different package. The *Fixed Core* columns in Table 6.4 show the runtime per cache probe averaged over 3 runs for a range of background request rates to the web sever. The *increase* column shows the percent increase in probe time relative to running with an idle victim VM.

Request	Fixed Core		Floating Core	
Rate	Runtime	Increase	Runtime	Increase
0	4033	0	4791	0
100	4780	19%	5362	12%
1000	6500	61%	6887	44%
1500	7740	92%	7759	62%
2000	9569	137%	8508	78%
3000	18392	356%	16630	247%

Table 6.4: **Performance Interference of a Webserver on LLCProbe.** *Runtimes (in microseconds) and percentage increase in performance degradation of LLCProbe (foreground) workload as a function of request rate to victim (background). For* Fixed Core *both VMs are pinned to the same core and for* Floating Core *Xen chooses where to execute them.*

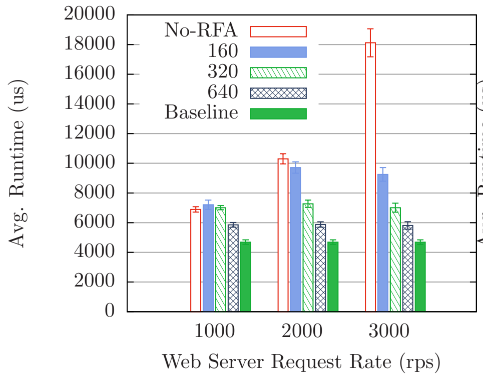
As the workload of the victim increases, we see a corresponding increase in the performance degradation of LLCProbe. To evaluate our hypothesis that the effect arises due to frequent interruptions, we use Xentrace [116] to record the domain switches that occur over a fixed period of time in which the LLCProbe VM runs. We analyzed the case of 1500 requests per second (rps) and 3000 rps. For the 3000 rps case, the web server runs for less than 1ms in 80% of the times it is scheduled whereas in the 1500 rps case the web server runs for less than 1ms only 40% of the time, because the longer run periods reflect fixed-length CPU tasks not correlated with traffic. Because Apache does not saturate its CPU allocation, it retains "boost" priority, which allows it to preempt LLCProbe for every request. Thus, LLCProbe also runs for short periods, causing it to lose the data in its cache.

The rightmost columns in Table 6.4 show the same experiment when the two VMs are allowed to float across all the cores (*floating*). We see a similar trend here, though slightly less severe because for some fraction of time, the victim and beneficiary VMs are scheduled on different packages and do not share an LLC. Thus, we expect in live settings such as EC2 to see less interference than when both VMs are pinned to the same core.

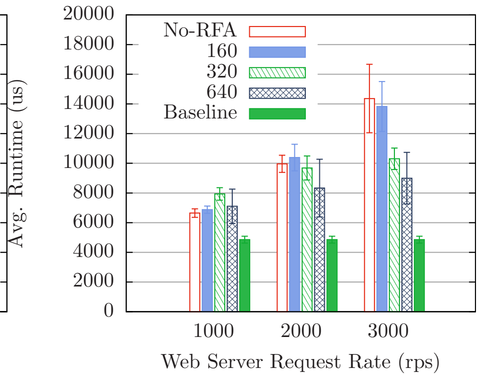
We separately investigate the effect of contention with the Xen driver domain, *Dom0* [1], which handles all device access such as interrupts or requests to send a packet. In the typical setting where *Dom0* is assigned one VCPU per physical CPU, *Dom0* may run on any core and uses the same scheduling mechanism as other guest VMs. As a result, *Dom0* receives boost and can interfere with the beneficiary just like the victim when it handles a network interrupt. *Dom0* and the beneficiary may share a CPU even if the victim is scheduled elsewhere.

The attack. As alluded to in Section 6.2, the beneficiary's performance

[1]The default configuration in Xen is to run device drivers in a single domain with privileged access to I/O hardware.



a: *Pinned*



b: *Floating*

Figure 6.5: **Performance of LLCProbe under Pinned and Floating VM Configurations..** *"Baseline" measures baseline performance when no traffic was sent to the victim; it is shown in each grouping for comparison. "No-RFA" measures performance when no RFA requests were sent. (Left) Performance when* LLCProbe *and web server VMs are pinned to same core. (Right) Performance when they float amongst cores. Error bars indicate one standard deviation.*

degradation is caused by a victim frequently preempting the beneficiary and thereby polluting its cache. The preemptions occur to handle static web page requests due to legitimate traffic to the victim. Our attack aims to exploit the victim's CPU allotment as a bottleneck resource in order to shift, in time, its accesses to the cache, and to reduce the number of requests it serves. Doing so will provide the beneficiary longer periods of uninterrupted access to the cache and less cache pollution from handling requests, resulting in increased cache hit rates and improved performance.

The trigger for this is the introduction of a small number of CGI requests per second from a helper. Even a low rate of requests per second can push the victim up to its CPU cap, forcing it to lose boost and thus consolidating its use of the cache into a smaller time frame. Introducing long-latency dynamic requests means that, instead of interrupting LLCProbe frequently, the web server runs continuously until the Xen scheduler preempts it, which allows LLCProbe to run uninterrupted. The Xen credit scheduler allows a maximum of 30ms of credit per VCPU, with each domain being allotted only one VCPU in our case. Therefore, the helper sends *RFA requests* that invoke the CPU-intensive CGI helper in an effort to use up the victim's CPU allotment. In addition, the CPU-intensive requests displace legitimate traffic and thus reduce the rate of requests that pollute the cache.

Here the helper is any system that can make CGI requests. Given the very low rate required, this could be a free micro instance running on the cloud or —scaling up— a single system that performs the RFA against many victims in parallel (that are each co-resident with a different beneficiary). While for some applications the helper might be put to better use helping with whatever computation the beneficiary is performing, in others this will not be possible (e.g., if it is not easily parallelized) or not as cost effective. We also mention that one might include a lightweight helper on the same VM as the beneficiary, but this would require care to ensure that interference from the client does not outweigh the potential speedup due to the RFA. In our experiments to follow, we run the helper on a system different from the one on which the beneficiary and victim co-reside.

6.3.1 Evaluation on Local Testbed

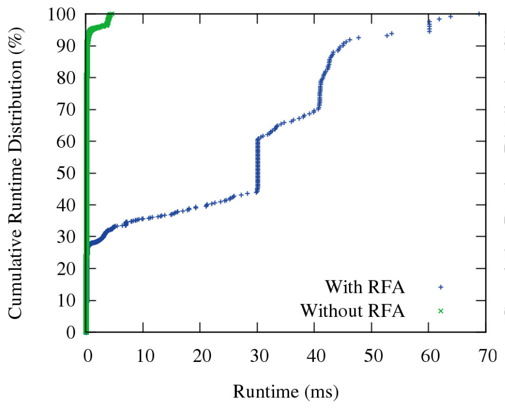
The results above show that LLCProbe experiences a significant performance gap when running on an otherwise idle server as opposed to one that is hosting one or more active web servers. In this section, we show that this performance gap can be narrowed using the RFA outlined above. In the following we look at the effectiveness of the attack under a range of *RFA intensities*, which specifies the their total runtime per second. Unless otherwise noted, we implement the RFA using CGI requests specifying 40ms of computation. We investigate a range of RFA intensities: 160, 320, and 640ms. This allows understanding both the effect of overloading the victim by requesting more computation than its total allotment of 400ms.

We first run LLCProbe fifteen times while the victim VM is idle to get a baseline. Then for each legitimate victim traffic rate and each level of RFA including "No-RFA", we run LLCProbe fifteen times while offering the appropriate legitimate traffic and RFA traffic.

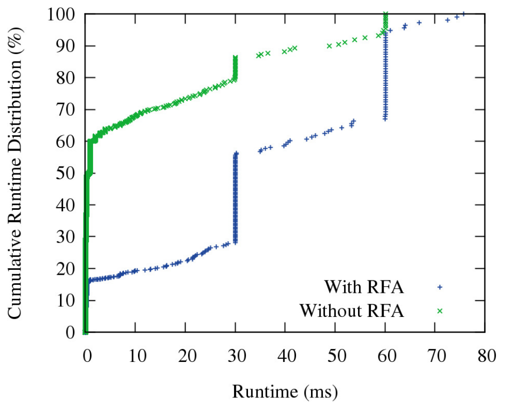
The average runtimes of these tests are shown in Figure 6.5. We observe several interesting trends. Consider the left chart, which reports on a setting with both victim and beneficiary pinned to the same core and all four *Dom0* VCPUs floating across all cores. First, introducing the extra load from the RFA requests helps the beneficiary. Second, the greater the victim's load the higher the payoffs from the RFA.

In order to understand these results, we ran additional experiments trying to identify various sources of interference on the beneficiary. There are three main sources of interference: two effects on request processing by the web server and the effect of network packet processing by *Dom0* . RFA requests help mitigate the effect of web server request handling in two ways. First, introducing sufficiently many CPU-intensive requests will deprive the web server of the boost priority. This is the major reason for the high performance improvement in the pinned case shown in Figure 6.5. Second, introducing long-running CGI requests reduces the amount of CPU time available to serve legitimate traffic and thus, implicitly reduces the capacity of the web server. This is the reason for higher payoffs at higher web-server request rates. Reducing *Dom0* 's impact on the beneficiary can only be indirectly achieved by saturating the web server and hence reducing the rate of incoming request to the web server.

Figure 6.6 shows the CDF of runtime durations of the web server (top chart) and LLCProbe (bottom chart) before being preempted both with and without an RFA for the pinned case. What we see is that LLCProbe runs for more than 1ms 85% of the time in the RFA case but only 40% of the time without the RFA. This accounts for part of its improved performance.



a: *Webserver*



b: *LLCProbe*

Figure 6.6: **Cumulative VM Runtime Distributions of Webserver and LLCProbeVMs..** *Runtime distribution of (top) the web server domain (with load 2,000 rps) and (bottom) the LLCProbe domain under both no RFA and with RFA 320 in pinned core case.*

Similarly, the web server changes from running longer than $1ms$ for only 10% of the time to 60% of the time. Furthermore, we can see that the web server often runs out of scheduling credit from the vertical line at $30ms$, indicating that it uses up some of its scheduling quanta.

Figure 6.7 shows the effect of displacing legitimate traffic at higher RFA intensities for the floating case. At low web-server request rates and low RFA intensities, the offered and the observed load remain similar. However, at 3000 rps and RFA intensity of 320, the observed load reduces to 1995 rps, which leads LLCProbe to have performance similar to No-RFA case at 2000 rps (right graph in Figure 6.5). This is the primary reason for large performance improvement at 3000 rps in both pinned and floating case shown in Figure 6.5.

In the floating case shown on the right in Figure 6.5, we see that RFA requests can sometimes hurt performance. There appear to be two reasons for this. First, some percentage of the time LLCProbe and Apache are running concurrently on two different cores sharing an LLC. Because the

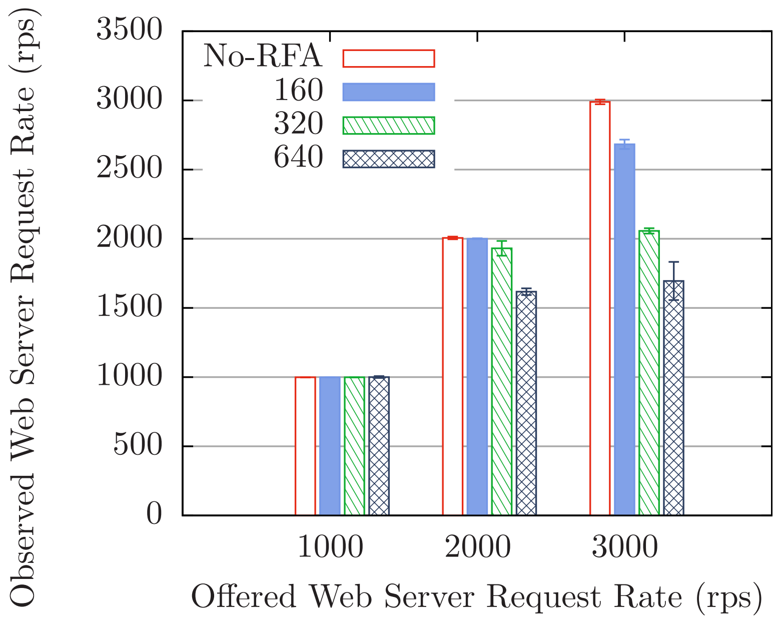


Figure 6.7: **Offered vs. Observed Load on Webserver with Varying RFA Intensities..** *Here all the VMs float across all cores.*

two loads run concurrently, every cache access by the web server hurts the performance of LLCProbe. In such a case, depriving the web server of boost is insufficient and LLCProbe performance increases only when the RFA rate is high enough so that the web server saturates its CPU allotment and so spends more than half the time waiting (40% CPU cap). In a separate experiment, we pinned the web server and the LLCProbe to different cores on the same package, and used a web-server request rate of 2000 rps. In this configuration, a high RFA intensity improved performance by a meager 2.4%. In contrast, when we pin the two to the same core, performance improved by 70%. Thus, improving performance when sharing a core is possible without reducing legitimate foreground traffic, while without sharing a core it requires displacing some legitimate traffic.

Second, in this floating case the beneficiary will for some percentage of

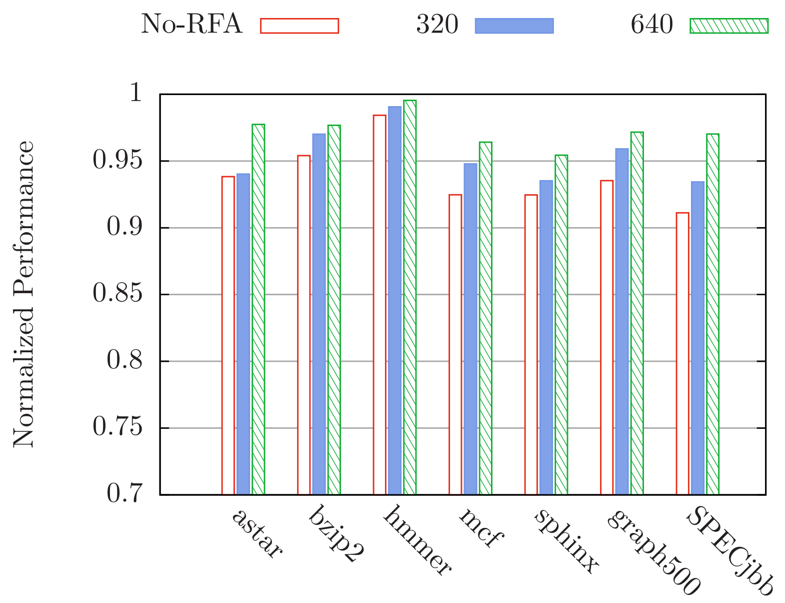


Figure 6.8: **Normalized performance for SPEC workloads on our local testbed..** *Normalized performance is calculated with the baseline runtime over runtime for various RFA intensities. All values are at a web server request rate of 3000 rps.*

the time be scheduled to run on a core or package as *Dom0* . Since *Dom0* handles all incoming and outgoing packets, it may frequently interrupt the beneficiary and pollute its cache state. When we pin LLCProbe and the web server to different packages (no shared cache) but let *Dom0* float, LLCProbe still experiences interference. At a load of 2000 rps on the web server, LLCProbe suffered a 78% degradation in performance just due to *Dom0* 's inference. The RFA we explore can only alleviate contention from *Dom0* by forcing a drop in the web server's foreground traffic rate (by exhausting its VM's CPU allocation as shown in Figure 6.7).

Finally, we analyze a spectrum of SPEC benchmarks. Each SPEC benchmark is run three times with an idle webserver, an active web server, and an active web server with various RFA intensities where all the VMs (including *Dom0*) float across all cores. Figure 6.8 depicts the normalized performance of seven benchmarks under no RFA and intensities of 320 and 640. That is, the reported fractions are computed as t′/t where t is the average runtime (request latency is computed and used for SPECjbb) and t′ is the average baseline performance when no traffic is sent to the victim. All benchmarks benefit from the RFA, with the general trend that cache-sensitive benchmarks (as indicated by a larger drop in performance relative to the baseline) achieve more gains from the RFA. For example, the 640 RFA increases normalized performance of SPECjbb from 0.91 to 0.97, a 6 percentage point improvement in performance and a 66.5% reduction in harm due to contention. The smallest improvement occurs with hmmer, which shows only a 1.1 percentage point improvement because it only suffers a performance loss of 1.6% without the RFA. Across all the benchmarks, the 640 RFA achieves an average performance improvement of 3.4 percentage points and recovers 55.5% of lost performance. These improvements come largely from the ability of the RFA to reduce the request rate of the victim web server.

6.3.2 Evaluation on EC2

The above experiments clearly indicate that RFAs can provide substantial gains in a controlled setting. To verify that the attacks will also work in a noisier, more realistic setting, we turn to Amazon's Elastic Compute Cloud (EC2). There are several reasons it is important to evaluate RFAs in a real cloud setting. First of all, the success of the RFA is highly dependent on the overall load of the physical machine. The instances in question (the beneficiary and the victim) make up only a portion of the total possible load on a single machine. If the other instances on the machine are heavy resource users, they will constantly interfere with the beneficiary and overshadow any performance benefit from slowing the victim. Thus, if most physical machines in EC2 are constantly under heavy load, we are unlikely to see much effect from an RFA on a single victim. Furthermore, EC2's Xen configuration is not publicly available and may prevent RFAs.

Thus, to understand if RFAs actually behave as an attacker would hope, it is necessary to verify their effectiveness in a live setting like EC2.

Ethical considerations. When using EC2 for experiments, we are obligated to consider the ethical, contractual, and legal implications of our work. In our experiments, we use instances running under our accounts in our names as stand-ins for RFA victims and beneficiaries. We abide by the Amazon user agreement, and use only the legitimate Amazon-provided APIs. We only attempt to send reasonable levels of traffic (slightly more than 2000 rps for a small web page) to our own instances (the stand-ins for victims). We do not directly interact with any other customer's instances. Our experiments are therefore within the scope of typical customer behavior on EC2: running a utilized web server and a CPU intensive application. Our experiments can therefore indirectly impact other customer's service only to the same extent as typical use.

Test machines. To test an RFA, we require control of at least two instances running on the same physical machine. As AWS does not provide this capability directly, we used known techniques [152] to achieve sets of co-resident m1.small instances on 12 different physical machines in the EC2 us.east-1c region. Specifically, we launched large numbers of instances of the same type and then used RTT times of network probes to check co-residence. Co-residence was confirmed using a cache-based covert channel. Nine of these were the same architecture: Intel Xeon E5507 with a 4MB LLC. We discarded the other instances to focus on those for which we had a large corpus, which are summarized in Table 6.9.

Each instance ran Ubuntu 11.04 with Linux kernel 2.6.38-11-virtual. For each machine, we choose one of the co-resident instances to play the role of the beneficiary and another one to be the victim. The beneficiary was configured with various benchmarks while the victim had the same Apache installation and configuration as in the local testbed (see Section 6.3.1). Any remaining co-resident instances were left idle.

Machine	#	Machine	#	Machine	#
E5507-1	4	E5507-4	3	E5507-7	2
E5507-2	2	E5507-5	2	E5507-8	3
E5507-3	2	E5507-6	2	E5507-9	3

Table 6.9: **Summary of EC2 machines and Number of Co-resident Instances..** *For m1.small instances running under our EC2 accounts.*

We used separate m1.small instances to run the victim load and the RFA traffic generator. We note that despite offering load of 2000 rps on EC2, the achieved load was only around 1500 on average and sometimes slightly less in the presence of RFAs.

Experimental procedure. We chose a subset of the benchmarks (sphinx, mcf, LLCProbe, and bzip2) used in the local testbed for the experiments on EC2. We ran each benchmark on a beneficiary instance while a co-resident victim received requests made by a client load generator as well as an RFA helper, both located on separate EC2 instances that were not co-resident with the beneficiary and victim. We used an intensity of 512ms and changed the duration of each RFA request to 16ms, as that was most effective in our experiments. For each benchmark we run the benchmark no RFA, followed by running it with the RFA, and we repeat this three times. (For LLCProbe, each single run of the benchmark was in fact five sequential runs to gather more samples.) This gives 4 data points (10 for LLCProbe). The interleaving of no-RFA and RFA helped limit the effects of unexpected intermittent noise (e.g., from other co-resident VMs outside our control) that may effect measurements. Throughout these experiments the client load generator sends web server requests at a configured rate. We also measure the baseline with no background traffic once at the start of measurements for each benchmark.

Aggregate effectiveness. We start by looking at average performance of the RFA's across all nine machines. Figure 6.10 depicts the results as normalized average runtimes (average runtime divided by average baseline runtime). Thus higher is better (less slowdown from interference). What we see is that the RFAs provides slight performance improvements across all the instances and, in particular, never hurts average runtime. While the absolute effects are small, they are not insignificant: the RFA improved LLCProbe performance by 6.04%. For the SPEC benchmarks (not shown), we see that the degradation due to the victim (the No-RFA) is, on average, less than observed on the local testbed. This may be due to the different architectures and software configurations, or it may be due to higher contention in the baseline case due to other co-resident instances (owned by other customers). Given the smaller gap between baseline and No-RFA, there is less absolute performance to recover by mounting an RFA. Nevertheless, as a fraction of lost performance, even here the beneficiary receives back a large fraction of its performance lost to interference.

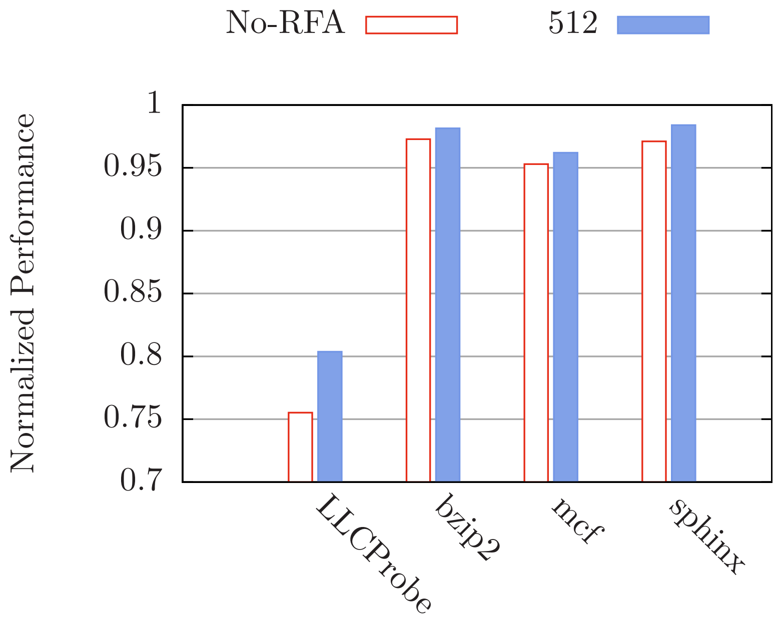
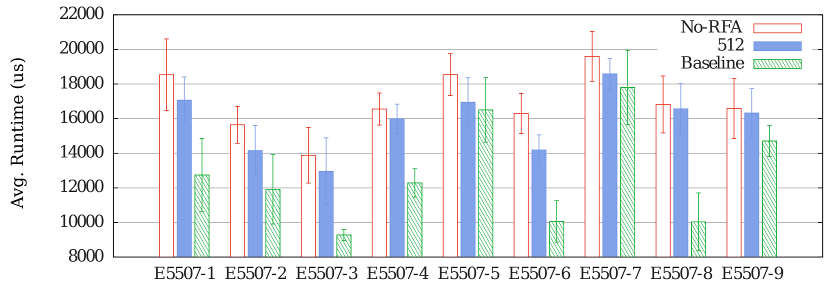


Figure 6.10: **Normalized Performance Across All Machines on EC2..** *Here workload performance is normalized with baseline runtime under zero performance interference.*

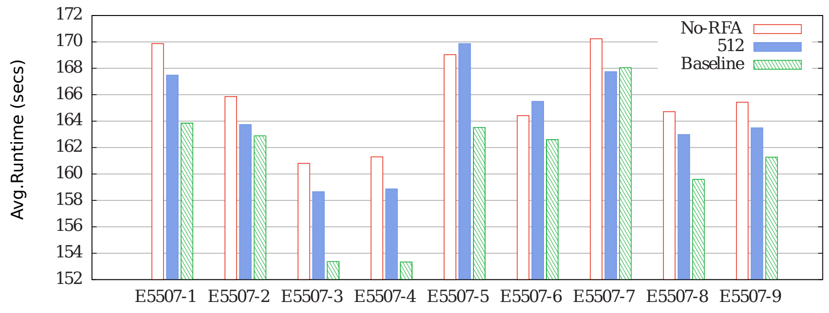
Per-machine breakdown. To understand the effect further and, in particular, to get a better sense of whether other (uncontrolled) co-resident instances are causing contention, we breakdown the results by individual machine. Figure 6.11 depicts average runtimes for each machine and for each of the four benchmarks. (The error bars for LLCProbe denote one standard deviation — for the other benchmarks we omitted these due to having three samples.) As it can be seen, the baseline, No-RFA, and RFA performances all vary significantly across the different machines. While we cannot know the precise reason for this, we speculate that it is mostly due to contention from other customer VMs or, possibly, slight differences in configuration and baseline software performance of the distinct machines.

Likewise the benefit of performing an RFA varies by machine. In the case of LLCProbe, RFAs were always beneficial, but the degree to which they improved performance varied. Machine E5507-6 had the highest speedup of 13% from the RFA, which corresponded to decreasing the cost of contention by about 33%. Interestingly, there seems to be little correlation between benchmarks, for example E5507-6 had negative improvement from RFA for the bzip2 and mcf benchmarks. Other machines faired better for SPEC benchmarks, for example E5507-1 had a 3.2% performance improvement under RFAs.

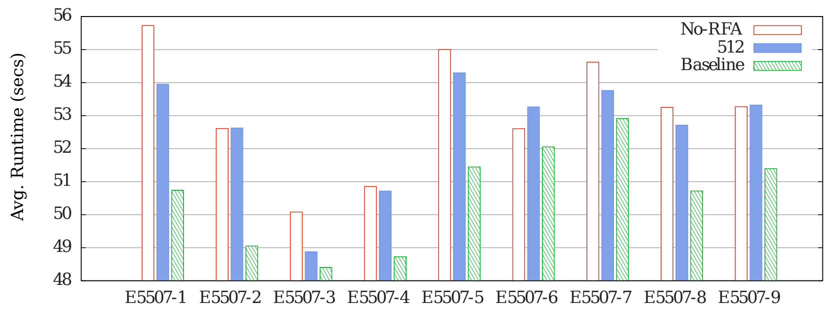
These varied results are not unexpected in the noisy environment of EC2. We draw two general conclusions. First, RFAs can provide significant speedups in the (real-world) environment of EC2, but the benefits will vary depending on a variety of environmental factors. Second, given that the aggregate benefit across all machines is positive, a greedy customer will —on average over the long term— benefit from mounting RFAs.



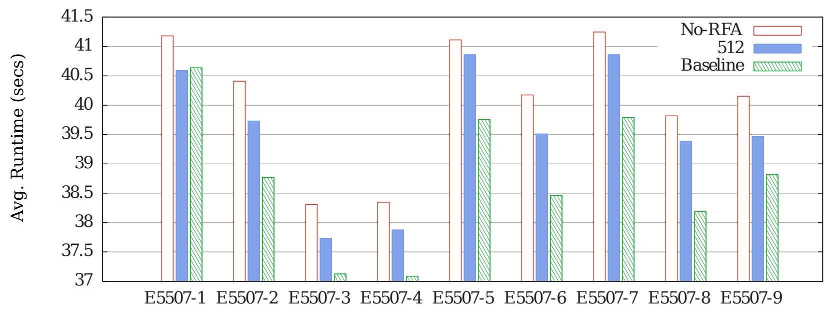
a: *LLCProbe*



b: *bzip2*



c: *mcf*



d: *sphinx*

Figure 6.11: **Average Runtimes of LLCProbe, bzip2, mcf, and sphinx Benchmarks Across 9 EC2 machines..** *Baseline has no traffic to victim, while No-RFA and 512 RFA have foreground request rate of 2000 rps.*

6.4 Discussion

Practical dimensions. Deploying a resource-freeing attack like the one explored in the last few sections would be subject to several complicating issues in practice. First, it may be difficult to predictably modify the victim's workload because the victim's normal (pre-RFA) workload may be unknown to the attacker. As shown in Section 6.3, the amount of extra work required was dependent on the existing workload of the victim. Here, simple adaptive techniques, where workload is continually introduced as long as it improves the beneficiary's performance, may suffice. Moreover, our results suggest an attacker would typically do well to overestimate the RFA intensity required.

Second, it may be that co-resident instances do not have services that are accessible to the RFA helper. As discussed in Section 6.1 a wide swath of, e.g., EC2 instances run public web servers, and such interrupt-driven workloads are likely to be the most damaging to cache-bound workloads. Even public servers may only be indirectly accessible to the helper, for example if they lie behind a load balancer. Future work might target RFAs that can exploit other avenues of generating a bottleneck resource for the victim, for example the attacker might generate extra contention on a disk drive using asynchronous accesses in order to throttle a victim's I/O bound processes. Such an attack would not require any form of logical access to the victim.

Third, the client workload we experimented with does not reflect all victim workloads seen in practice. For example, if thousands of independent clients submit requests concurrently, the RFA may not be able to effect as much displacement of inbound connection requests (though request processing will still be displaced). Future work might clarify the vulnerability of other victim workloads to RFAs.

Economics of RFAs. In the setting of public clouds, performance improvement can translate directly to cost improvement since one pays per unit time. For long running jobs, even modest improvements in performance can significantly lower cost. Of course, one must account for the cost of mounting the RFA itself, which could diminish the cost savings. The RFAs we explored used a helper that sends a small number of web requests to the victim. For example, our helper uses only 15 Kbps of network bandwidth with a CPU utilization of 0.7% (of the E5430 as configured in our local testbed). We located this helper on a separate machine. That the helper is so lightweight means that one might implement it in a variety of ways to ameliorate its cost. For example, by running it in places where spare cycles cannot be used for the main computational task or even on a non-cloud system used to help manage cloud tasks. One could also use a cheap VM instance that runs helpers for a large set of beneficiaries, thereby amortizing the cost of the VM instance.

A related issue is that of VM migration. While contemporary IaaS clouds do not enable dynamic migration, customers may move a VM from one system to (hopefully) another by shutting it down and restarting it. The beneficiary could therefore try to migrate away from a contended host instead of mounting an RFA. We view migration and RFAs as two complementary directions along which a greedy customer will attempt to optimize their efficiency. Which strategy, or a combination thereof, works best will depend on the contention, the workload, the likelihood of ending up on an uncontended host, pricing, etc. Understanding the relative economic and performance benefits of migration and RFAs is an interesting question for future work.

Preventing RFAs. To prevent the kinds of RFAs we consider, one could deploy VMs onto dedicated instances. This was suggested in the cloud setting by Ristenpart et al. [152], and subsequently added as a feature in EC2. However, the significant cost of dedicated instances makes it impractical for a variety of settings.

There are two primary methods for preventing RFAs even in the case of multiplexed physical servers: stronger isolation and smarter scheduling. A hypervisor that provides strong isolation for every shared resource can prevent RFAs. This entails using non-work conserving scheduling, so that idleness of a resource allocated to one VM does not benefit another. In addition, it requires hardware support for allocating access to processor resources, such as the cache and memory bandwidth. With current hardware, the only possibility is cache coloring, which sets virtual-to-physical mappings to ensure that guest virtual machines do not share cache sets [102]. This effectively partitions the cache in hardware, which hurts performance for memory-intensive workloads. Finally, it requires that the hypervisor never overcommit and promise more resources to VMs than are physically available, because concurrent use of overcommitted resources cannot be satisfied. While this approach may work, it sacrifices performance and efficiency by leaving resources idle.

A second approach is to apply smarter scheduling. Based on the contention results in Section 6.2, the hypervisor can monitor the VMs between processes and attempt to schedule those workloads that do not conflict. This approach, often applied to multicore and multithreaded scheduling [48, 63, 159], detects workloads with conflicting resource usages via statistics and processor performance counters, and attempts to schedule them at different times, so they do not concurrently share the contended resource, or on separate cores or packages to reduce contention, as in the case of the LLC.

A final idea would be to prevent RFAs by detecting and blocking them. We suspect that this would be very difficult in most settings. RFAs need not abuse vulnerabilities on a system, rather they can simply take advantage of legitimate functionality (e.g., CGI scripts on a web server). Moreover they are stealthy in the sense that it may only require a few requests per second to drive the victim up against a resource bottleneck. A provider or the victim itself would be hard pressed to detect and block RFA requests without preventing legitimate access to the resource.

7 Soft-isolation: Improving Isolation in Public Clouds

Cross-VM side-channel attacks are one of the most dangerous co-location attacks that have been demonstrated in the public clouds [93, 152]. Perhaps surprisingly, then, is the lack of any investigation of the relationship between hypervisor scheduling policies and side-channel efficacy.

In this chapter, we evaluate the ability of system software to mitigate cache-based side-channel attacks through scheduling. In particular, we focus on the type of mechanism that has schedulers ensure that CPU-bound workloads cannot be preempted before a minimum time quantum, even in the presence of higher priority or interactive workloads. We say that such a scheduler offers a *minimum run time (MRT) guarantee*. Xen version 4.2 features an MRT guarantee mechanism for the stated purpose of improving the performance of batch workloads in the presence of interactive workloads that thrash their cache footprint [59]. A similar mechanism also exists in the Linux CFS scheduler [126].

Cache-based side-channel attacks are an example of such highly interactive workloads that thrash the cache. One might therefore hypothesize that by reducing the frequency of preemptions via an MRT guarantee, one achieves a level of soft isolation suitable for mitigating, or even preventing, a broad class of shared-core side-channel attacks. We investigate this hypothesis, providing the first analysis of MRT guarantees as a defense against cache-based side-channel attacks. With detailed measurements of cache timing, we show that even an MRT below $1ms$can defend against existing attacks.

But an MRT guarantee can have negative affects as well: latency-sensitive workloads may be delayed for the minimum time quantum. To evaluate the performance impact of MRT guarantees, we provide extensive measurements with a corpus of latency-sensitive and batch workloads. We conclude that while worst-case latency can be hindered by large MRTs in some cases, in practice Xen's existing core load-balancing mechanisms mitigate the cost by separating CPU-hungry batch workloads from latency-sensitive interactive workloads. As just one example, memcached, when running alongside batch workloads, suffers only a 7% overhead on 95^{th}-percentile latency for a $5ms$MRT compared to no MRT. Median latency is not affected at all.

The existing MRT mechanism only protects CPU-hungry programs that do not yield the CPU or go idle. While we are aware of no side-channel attacks that exploit such victim workloads, we nevertheless investigate a simple and lightweight use of CPU *state cleansing* to protect programs that quickly yield the CPU by obfuscating predictive state. By implementing this in the hypervisor scheduler, we can exploit knowledge of when a cross-VM preemption occurs and the MRT has not been exceeded. This greatly mitigates the overheads of cleansing, attesting to a further value to soft-isolation style mechanisms. In our performance evaluation of this mechanism, we see only a 10–$50\mu s$worse-case overhead on median latency due to cleansing while providing protection for all guest processes within a VM (and not just select ones, as was the case in Düppel). In contrast, other proposed defenses have similar (or worse) overhead but require new hardware, new guest operating systems, or restrict system functionality.

In the next section, we describe the Xen hypervisor scheduling system, its MRT mechanism, and the principle of soft isolation. In Section 7.2 we measure the effectiveness of MRT as a defense. Section 7.3 shows the performance of Xen's MRT mechanism, and Section 7.4 describes combining MRT with cache cleansing.

7.1 MRT Guarantees and Soft Isolation

We investigate a different strategy for mitigating per-core side-channels: adjusting hypervisor core scheduling to limit the rate of preemptions. This targets the second requirement of attacks such as ZJRR. Such a scheduler would realize a security design principle that we call *soft isolation*[1]: limiting the frequency of potentially dangerous interactions between mutually untrustworthy programs. Unlike hard isolation mechanisms, we will allow shared state but attempt to use scheduling to limit the damage. Ideally, the flexibility of soft isolation will ease the road to deployment, while still significantly mitigating or even preventing side-channel attacks. We expect that soft isolation can be incorporated as a design goal in a variety of resource management contexts. That said, we focus in the rest of this work on CPU core scheduling.

Xen scheduling. Hypervisors schedule virtual machines much like an operating system schedules processes or threads. Just as a process may contain multiple threads that can be scheduled on different processors, a virtual machine may consist of multiple virtual CPUs (VCPUs) that can be scheduled on different physical CPUs (PCPUs). The primary difference between hypervisor and OS scheduling is that the set of VCPUs across all VMs is relatively static, as VM and VCPU creation/deletion is a rare event. In contrast, processes and threads are frequently created and deleted.

Hypervisor schedulers provide low-latency response times to interactive tasks by prioritizing VCPUs that need to respond to an outstanding event. The events are typically physical device or virtual interrupts from

[1]The term "soft" is inherited from soft real-time systems, where one similarly relaxes requirements (in that case, time deadlines, in our case, isolation).

packet arrivals or completed storage requests. Xen's credit scheduler normally lets a VCPU run for 30ms before preempting it so another VCPU can run. However, when a VCPU receives an event, it may receive *boost* priority, which allows it to preempt non-boosted VCPUs and run immediately.

VCPUs are characterized by Xen as either *interactive* (or *latency-sensitive*) if they are mostly idle until an interrupt comes in, at which point they execute for a short period and return to idle. Typical interactive workloads are network servers that execute in response to an incoming packet. We refer to VCPUs that are running longer computations as *batch* or *CPU-hungry*, as they typically execute for longer than the scheduler's time slice (30ms for Xen) without idling.

Schedulers can be *work conserving*, meaning that they will never let a PCPU idle if a VCPU is ready to run, or *non-work conserving*, meaning that they enforce strict limits on how much time a VCPU can run. While work-conserving schedulers can provide higher utilization, they also provide worse performance isolation: if one VCPU goes from idle to CPU-hungry, another VCPU on the same PCPU can see its share of the PCPU drop in half. As a result, many cloud environments use non-work conserving schedulers. For example, Amazon EC2's *m1.small* instances are configured to be non-work conserving, allocating roughly 40% of a PCPU (called cap in Xen) to each VCPU of a VM.

Since version 4.2, Xen has included a mechanism for rate limiting preemptions of a VCPU; we call this mechanism a minimum run-time (MRT) guarantee. The logic underlying this mechanism is shown as a flowchart in Figure 7.1. Xen exposes a hypervisor parameter, `ratelimit_us` (the MRT value) that determines the minimum time any VCPU is guaranteed to run on a PCPU before being available to be context-switched out of the PCPU by another VCPU. One could also rate limit preemptions in other ways, but an MRT guarantee is simple to implement. Note that the MRT is not applicable to VMs that voluntarily give up the CPU, which happens when

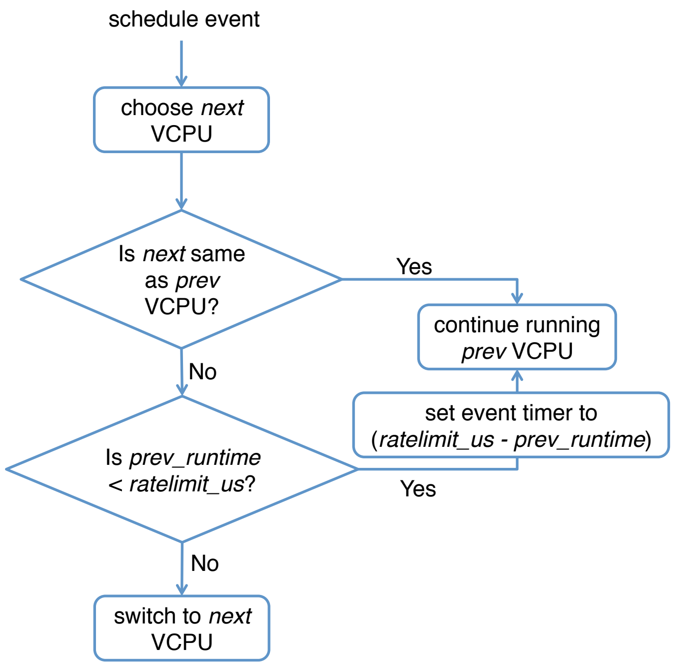

schedule event
choose *next* VCPU
Is *next* same as *prev* VCPU?
Yes
continue running *prev* VCPU
No
set event timer to (*ratelimit_us* - *prev_runtime*)
Is *prev_runtime* < *ratelimit_us*?
Yes
No
switch to *next* VCPU

Figure 7.1: **Logic underlying the Xen MRT mechanism.**

the VM goes idle or waits for an event to occur.

As noted previously, the original intent of Xen's MRT was to improve performance for CPU-hungry workloads run in the presence of latency-sensitive workloads: each preemption pollutes the cache and other microarchitectural state, slowing the CPU-intensive workload

Case study. We experimentally evaluate the Xen MRT mechanism as a defense against side-channel leakage by way of soft isolation. Intuitively, the MRT guarantee rate-limits preemptions and provides an attacker less granularity in his observations of the victim's use of per-CPU-core resources. Thus one expects that increased rate-limits decreases vulnerability. To be deployable, however, we must also evaluate the impact of MRT guarantees on benign workloads. In the next two sections we investigate the following questions:

1. How do per-core side-channel attacks perform under various MRT values? (Section 7.2)

2. How does performance vary with different MRT values? (Section 7.3)

7.2 Side-channels under MRT Guarantees

We experimentally evaluate the MRT mechanism as a defense against side-channel leakage for per-core state. We focus on cache-based leakage.

Experimental setup. Running on the hardware setup shown in Table 7.2, we configure Xen to use two VMs, a victim and attacker. Each has two VCPUs, and we pin one attacker VCPU and one victim VCPU to each of two PCPUs (or cores). We use a non-work-conserving scheduler whose configuration is shown in Table 7.9. This is a conservative version of the ZJRR attack setting, where instead the VCPUs were allowed to float — pinning the victims to the same core only makes it easier for the attacker. The hardware and Xen configurations are similar to the configuration used in EC2 m1.small instances [61]. (Although Amazon does not make their precise hardware configurations public, we can still gain some insight into the hardware on which an instance is running by looking at `sysfs` and the `CPUID` instruction.)

Cache-set timing profile. We start by fixing a simple victim to measure the effects of increasing MRT guarantees. We have two functions that each access a (distinct) quarter of the instruction cache (I-cache)[2]. The victim alternates between these two functions, accessing each quarter 500 times. This experiment models a simple I-cache side-channel where switching from one quarter to another leaks some secret information (we

[2]Our test machine has a 32 KB, 4-way set associative cache with 64-byte lines. There are 128 sets.

Machine Configuration	Intel Xeon E5645, 2.40GHz clock, 6 cores in one package
Memory Hierarchy	Private 32 KB L1 (I- and D-cache), 256 KB unified L2, 12 MB shared L3 and 16 GB main memory.
Xen Version	4.2.1
Xen Scheduler	Credit Scheduler 1
Dom0 OS	Fedora 18, 3.8.8-202.fc18.x86_64
Guest OS	Ubuntu 12.04.3, Linux 3.7.5

Table 7.2: **Hardware configuration in local test bed.**

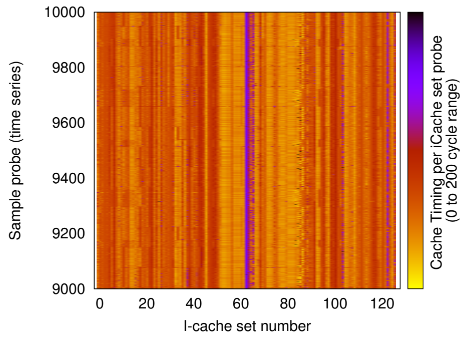
call any such leaky function a *sensitive* operation). Executing the 500 access to a quarter of the I-cache requires approximately 100μμs when run in isolation.

We run this victim workload pinned to a victim VCPU that is pinned to the same PCPU as the attacker VCPU. The attacker uses the IPI-based Prime+Probe technique[3] and measures the time taken to access each I-cache set, similar to ZJRR [191].

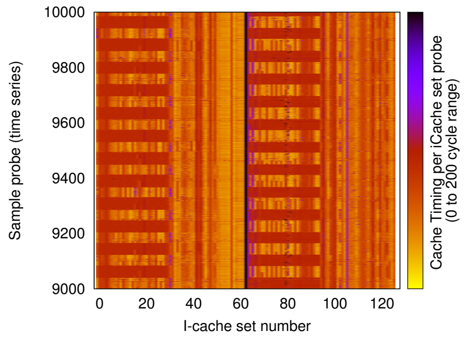
Figure 7.3 shows heat maps of the timings of the various I-cache sets as taken by the Prime+Probe attacker, for various MRT values between 0 (no MRT) and 5ms. Darker colors are longer access times, indicating conflicting access to the cache set by the victim. One can easily see the simple alternating pattern of the victim as we move up the y-axis of time in Figure 7.3b. Also note that this is different from an idle victim under zero-MRT shown in Figure 7.3a. With no MRT, the attacker makes approximately 40 observations of each cache set, allowing a relatively detailed view of victim behavior.

As the MRT value increases we see the loss of resolution by the attacker as its observations become less frequent than the alternations of the victim.

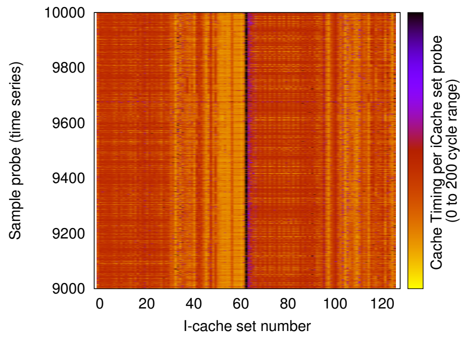
[3]Note that the attacker requires two VCPUs, one measuring the I-cache set timing whenever interrupted and the other issuing the IPIs to wake up the other VCPU. The VCPU issuing IPIs is pinned to a different PCPU.



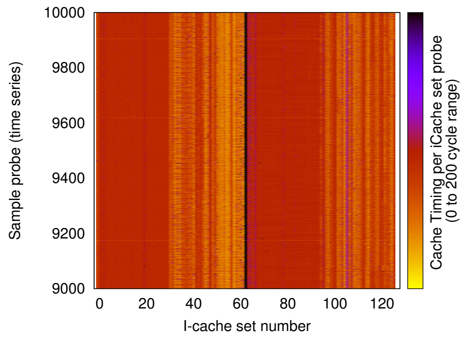
a: *Zero-MRT w/ Idle Victim*



b: *Zero-MRT*



c: 100µs-*MRT*

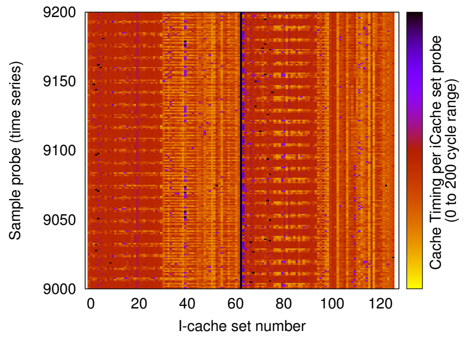


d: 1ms-*MRT*

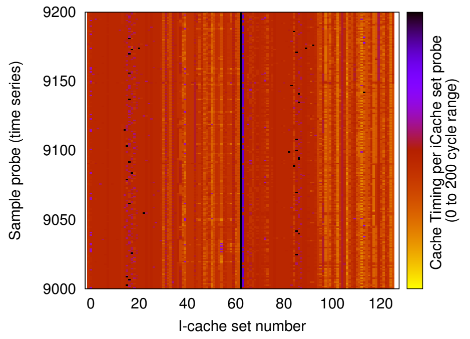
Figure 7.3: **Heatmaps of I-cache set timing as observed by a prime-probe attacker.** *Displayed values are from a larger trace of 10,000 timings. (a) Timings for idle victim and no MRT. (b)–(d) Timings for varying MRT values with the victim running.*

At an MRT of 100µsthe pattern is still visible, but noisier. Although the victim functions run for 100µs, the prime+probe attacker slows downs the victim by approximately a factor of two, allowing the pattern to be visible with a 100µsMRT. When the MRT value is set to 1msthe attacker obtains no discernible information on when the switching between each I-cache set happens.

In general, an attacker can observe victim behavior that occurs at a lower frequency than the attacker's preemptions. We modify the vic-



a: 1ms-*MRT*



b: 5ms-*MRT*

Figure 7.4: **Heatmaps of I-cache set timings as observed by a prime-probe attacker for 10x slower victim computations.** *Displayed values are from a larger trace of 9,200 timings.*

tim program to be 10x slower (where each function takes approximately 1msstandalone). Figure 7.4 shows the result for this experiment. With a 1msMRT, we observe the alternating pattern. When the MRT is raised to 5ms, which is longer than the victim's computation ($\approx$ 2ms), no pattern is apparent. Thus, when the MRT is longer than the execution time of a security-critical function this side-channel fails.

While none of this proves lack of side-channels, it serves to illustrate the dynamics between side-channels, duration of sensitive victim operations, and the MRT: as the MRT increases, the frequency with which an attacker can observe the victim's behavior decreases, and the signal and hence leaked information decreases. All this exposes the relationship between the speed of a sensitive operation, the MRT, and side-channel availability for an attacker. In particular, very long operations (e.g., longer than the MRT) may still be spied upon by side-channel attackers. Also, infrequently accessed but sensitive memory accesses may leak to the attacker. We hypothesis that at least for cryptographic victims, even moderate MRT values on the order of a handful of milliseconds are sufficient to prevent per-core side-channel attacks. We next look, therefore, at how this relationship plays out for cryptographic victims.

```
Procedure SquareMult($x, e, N$):
    Let $e_n, ..., e_1$ be the bits of $e$
    $y \leftarrow 1$
    for $i \leftarrow n$ down to $1$ do
        $y \leftarrow$ Square($y$)
        $y \leftarrow$ ModReduce($y, N$)
        if $e_i = 1$ then
            $y \leftarrow$ Mult($y, x$)
            $y \leftarrow$ ModReduce($y, N$)
    return $y$        :
```

Algorithm 7.5: **Modular exponentiation algorithm used in libgcrypt version 1.5.0.** *Note that the control flow followed when* $e_i = 1$ *is lines* $5 \to 6 \to 7 \to 8 \to 9$ *and when* $e_i = 0$ *is lines* $5 \to 6$*; denoted by the symbols* `1` *and* `0`*, respectively.*

ElGamal victim. We fix a victim similar to that targeted by ZJRR. The victim executes the modular exponentiation implementation from libgcrypt 1.5.0 using a 2048-bit exponent, base and modulus, in a loop. Pseudo-code of the exponentiation algorithm appears in Figure 7.5. One can see that learning the sequence of operations leaks the secret key values: if the code in lines 8 and 9 is executed, the bit is a 1; otherwise it is a zero. We instrument libgcrypt to write the current bit being operated upon to a memory page shared with the attacker, allowing us to determine when preemptions occur relative to operations within the modular exponentiation.

For no MRT guarantee, we observe that the attacker can preempt the victim many times per individual square, multiply, or reduce operation (as was also reported by ZJRR). With MRT guarantees, the rate of preemptions drops so much that the attacker only can interrupt once every several iterations of the inner loop. Table 7.6 gives the number of bits operated on between attacker preemptions for various MRT values. Table 7.7 gives the number of preemptions per entire modular exponentiation computation.

Xen MRT (ms)	Avg. ops/run	Min. ops/run
0	0.096	0
0.1	14.1	4
0.5	49.0	32
1.0	92.6	68
2.0	180.7	155
5.0	441.2	386
10.0	873.1	728

Table 7.6: **The average and minimal number of ElGamal secret key bits operated upon between two attacker preemptions for a range of MRT values.** *Over runs with 40K preemptions.*

Xen MRT	Preemptions per function call		
(ms)	Min	Median	Max
0	3247	19940	20606
0.1	74	155	166
0.5	22	42	47
1.0	16	22	25
2.0	10	11	13
5.0	0	4	6
10.0	1	2	3

Table 7.7: **Rate of preemption with various MRT.** *Here the function called is the Modular-Exponentiation implementation in libgcrypt with a 2048 bit exponent. Note that for zero MRT the rate of preemption is very high that victim computation involving a single bit was preempted multiple times.*

We see that for higher MRT values, the rate of preemption per call to the full modular exponentiation reduces to just a handful. The ZJRR attack depends on multiple observations *per operation* to filter out noise, so even at the lowest MRT value of 100μs, with 4–14 operations per observation, the ZJRR attack fails. In Appendix A.1, we discuss how one might model this leakage scenario formally and evidence a lack of any of a large class of side-channel attacks.

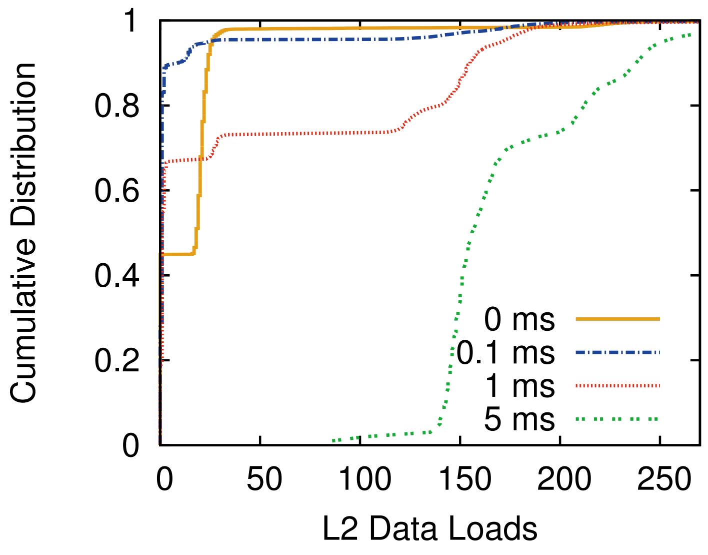


Figure 7.8: **CDF of L2 data-loads per time slice experienced by OpenSSL-AES victim.** *L2 data loads are performance counter events that happen when a program requests for a memory word that is not in both L1 and L2 private caches (effectively a miss). When running along-side a Prime+Probe attacker, these data-cache accesses can be observed by the attacker.*

AES victim. We evaluate another commonly exploited access-driven side-channel victim, AES, which leaks secret information via key-dependent indexing into tables stored in the L1 data cache [78, 141]. The previous attacks, all in the cross-process setting, depend on observing a very small number of cache accesses to obtain a clear signal of what portion of the table was accessed by the victim. Although there has been no known AES attack in the cross-VM setting (at least when deduplication is turned off, otherwise see [97]), we evaluate effectiveness of MRT against the best known IPI Prime+Probe spy process due to ZJRR. In particular, we measured the number of private data-cache misses possibly observable by this Prime+Probe attacker when the victim is running AES encryption in a loop.

To do so, we modified the Xen scheduler to log the count of private-cache misses (in our local testbed both L1 and L2 caches are private) experienced by any VCPU during a scheduled time slice. This corresponds to the number of data-cache misses an attacker could ideally observe. Figure 7.8 shows the cumulative distribution of the number of L2-data cache misses (equivalently, private data-cache loads) during a time slice of the victim running OpenSSL-AES. We can see that under no or lower MRTs the bulk of time slices suffer only a few tens of D-cache misses that happen between two back-to-back preemptions of the attacker. (We note that this is already insufficient to perform prior attacks.) The number of misses increases to close to 200 for an MRT value of $5ms$. This means that the AES process is evicting its own data, further obscuring information from a would-be attacker. Underlying this is the fact that the number of AES encryptions completed between two back-to-back preemptions increases drastically with the MRT: found that thousands to ten thousands AES block-encryptions were completed between two preemptions when MRT was varied from $100\mu s$to $5ms$, respectively.

Summary. While side channels pose a significant threat to the security of cloud computing, our measurements in this section show that, fortunately, the hypervisor scheduler can help. Current attacks depend on frequent preemptions to make detailed measurements of cache contents. Our measurements show that even delaying preemption for a fraction of millisecond prevents known attacks. While this is not proof that future attacks won't be found that circumvent the MRT guarantee, it does strongly suggest that deploying such a soft-isolation mechanism will raise the bar for attackers. This leaves the question of whether this mechanism is cheap to deploy, which we answer in the next section.

Note that we have focused on using the MRT mechanism for CPU and, indirectly, per-core hardware resources that are shared between multiple VMs. But rate-limiting-type mechanisms may be useful for other shared devices like memory, disk/SSD, network, and any system-level shared devices which suffer from a similar access-driven side-channels. For instance, a timed disk read could reveal user's disk usage statistics like relative disk head positions [100]. Fine-grained sharing of the disk across multiple users could leak sensitive information via such a timing side-channel. Reducing the granularity of sharing by using MRT-like guarantees in the disk scheduler (e.g., servicing requests from user for at least T_{min}, minimum service time, before serving requests from another user) would result in a system with similar security guarantees as above, eventually making such side-channels harder to exploit. Further research is required to analyze the end-to-end performance impact of such a mechanism for various shared devices and schedulers that manage them.

7.3 Performance of MRT Mechanism

The analysis in the preceding section demonstrates that MRT guarantees can meaningfully mitigate a large class of cache-based side-channel attacks. The mitigation becomes better as MRT increases. We therefore turn to determining the maximal MRT guarantee one can fix while not hindering performance.

7.3.1 Methodology

We designed experiments to quantify the negative and positive effects of MRT guarantees as compared to a baseline configuration with no MRT (or zero MRT). Our testbed configuration uses the same hardware as in the last section and the Xen configurations are summarized in Table 7.9. We run two DomU VMs each with a single VCPU. The two VCPUs are pinned to the same PCPU. Pinning to the same PCPU serves to isolate the effect of the MRT mechanism. The management VM, Dom0, has 6 VCPUs, one for each PCPU (a standard configuration option). The remaining PCPUs in the system are otherwise left idle.

Work-conserving configuration

Dom0	6 VCPU / no cap / weight 256
DomU	1 VCPU / 2 GB memory / no cap / weight 256

Non-work-conserving configuration

Dom0	6 VCPU / no cap / weight 512
DomU	1 VCPU / 2 GB memory / 40% cap / weight 256

Table 7.9: **Xen configurations used for performance experiments.**

We use a mix of real-world applications and microbenchmarks in our experiments (shown in Table 7.10). The microbenchmark *CProbe* simulates a perfectly cache-sensitive workload that continuously overwrites data to the (unified) L2 private cache, and *Chatty-CProbe* is its interactive counterpart that overwrites the cache every 10µsand then sleeps. We also run the benchmarks with an idle VCPU (labeled *Idle* below).

7.3.2 Latency Sensitivity

The most obvious potential performance downside of a MRT guarantee is increased latency: interactive workloads may have to wait before gaining access to a PCPU. We measure the negative effects of MRT guarantees by running latency-sensitive workloads against *Nqueens* (a CPU-bound program with little memory access). Figure 7.11 shows the 95^{th} percentile latency for the interactive workloads. The baseline results are shown as a MRT of 0 on the X-axis. As expected, the latency is approximately equal to the MRT for almost all workloads (*Apache* has higher latency because it requires multiple packets to respond, so it must run multiple times to complete a request). Thus, in the presence of a CPU-intensive workload and when pinned to the same PCPU, the MRT can have a large negative impact on interactive latency.

As the workloads behave essentially similarly, we now focus on just the *Data-Caching* workload. Figure 7.12 shows the response latency when run

CPU-hungry Workloads

Workload	Description
SPECjbb	Java-based application server [160]
graph500	Graph analytics workload [76] with scale of 18 and edge factor of 20.
mcf, sphinx, bzip2	SpecCPU2006 cache sensitive benchmarks [85]
Nqueens	Microbenchmark solving n-queens problem
CProbe	Microbenchmark that continuously trashes L2 private cache.

Latency-sensitive Workloads

Workload	Description
Data-Caching	Memcached from Cloud Suite-2 with twitter data set scaled by factor of 5 run for 3 minutes with rate of 500 requests per second [64].
Data-Serving	Cassandra KV-store from Cloud Suite-2 with total of 100K records[4] [64]
Apache	Apache webserver, HTTPing client [90], single 4 KB file at $1ms$ interval.
Ping	Ping command at $1ms$ interval.
Chatty-CProbe	One iteration of *CProbe* every $10\mu s$.

Table 7.10: **Workloads used in performance experiments.**

against other workloads. For the two CPU-intensive workloads, *CProbe* and *Nqueens*, latency increases linearly with the MRT. However, when run against either an idle VCPU or *Chatty-CProbe*, which runs for only a short period, latency is identical across all MRT values. Thus, the MRT has little impact when an interactive workload runs alone or it shares the PCPU with another interactive workload.

We next evaluate the extent of latency increase. Figure 7.13 shows the 25th, 50th, 75th, 90th, 95th and 99th percentile latency for *Data-Caching*. At the 50th percentile and below, latency is the same as with an idle

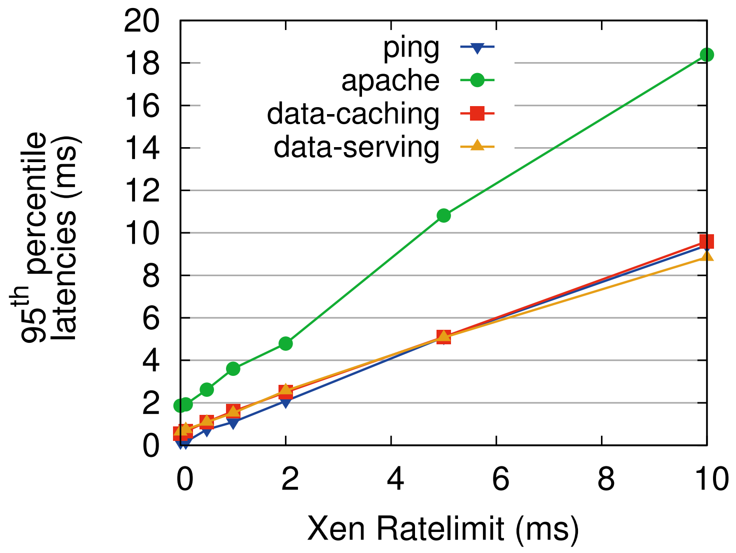


Figure 7.11: **95th Percentile Latency of Various Latency Sensitive Workloads.** *Under non-work-conserving scheduling.*

VCPU. However, at the 75th latency rises to half the MRT, indicating that a substantial fraction of requests are delayed.

We repeated the above experiments for the work-conserving setting, and the results were essentially the same. We omit them for brevity. Overall, we find that enforcing an MRT guarantee can severely increase latency when interactive VCPUs share a PCPU with CPU-intensive workloads. However, they have limited impact when multiple interactive VCPUs share a PCPU.

7.3.3 Batch Efficiency

In addition to measuring the impact on latency-sensitive workloads, we also measure the impact of MRT guarantees on CPU-hungry workloads. The original goal of the MRT mechanism was to reduce frequent VCPU context-switches and improve performance of batch workloads. We pin a

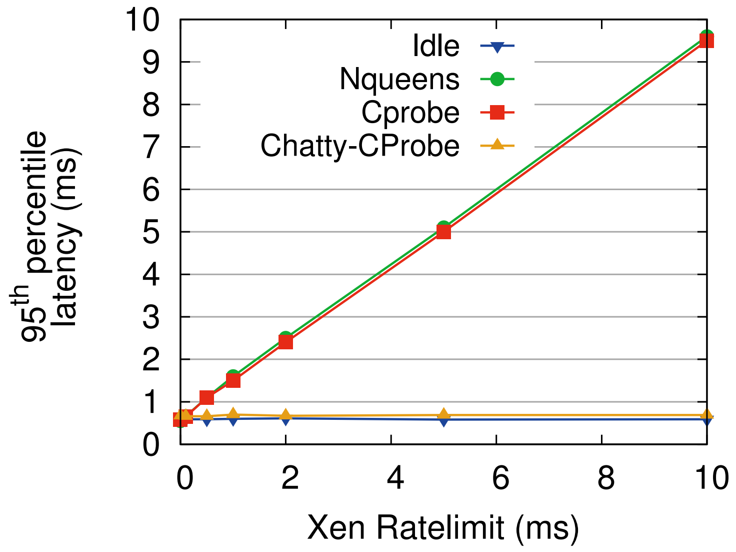


Figure 7.12: **95th Percentile Request Latency of *Data-Caching* Workload with Various Competing Micro-benchmarks.** *Under non-work-conserving scheduling.*

CPU-hungry workload to a PCPU against competing microbenchmarks.

Figure 7.14 shows the effect of MRT values on the *graph500* workload when run alongside various competing workloads. Because this is work-conserving scheduling, the runtime of *graph500* workload increases by roughly a factor of two when run alongside *Nqueens* and *CProbe* as compared to *Idle*, because the share of the PCPU given to the VCPU running *graph500* drops by one half. The affect of MRT is more pronounced when looking running alongside *Chatty-CProbe*, the workload which tries to frequently interrupt *graph500* and trash its cache. With no MRT guarantee, this can double the runtime of a program. But with a limit of only 0.5ms, performance is virtually the same as with an idle VCPU, both because *Chatty-CProbe* uses much less CPU and because it trashes the cache less often.

With a non-work-conserving scheduler, the picture is significantly dif-

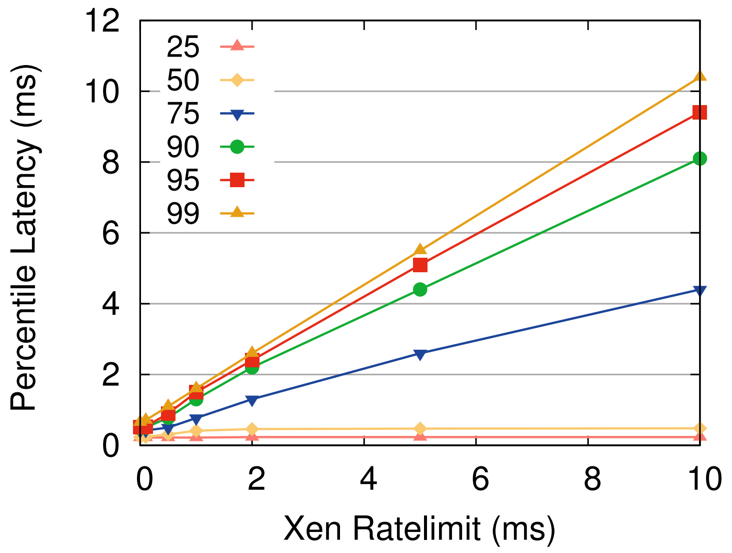


Figure 7.13: **25th, 50th, 75th, 90th, 95th and 99th Percentile Latency of *Data-Caching* Workload when run alongside *Nqueens*.** *Under non-work-conserving scheduling.*

ferent. Figure 7.15 shows the performance of three batch workloads when run alongside a variety of other workloads, for various MRT values. First, we observe that competing CPU-bound workloads such as *Nqueens* and *CProbe* do not significantly affect the performance of CPU-bound applications, even in the case of *CProbe* that trashes the cache. This occurs because the workloads share the PCPU at coarse intervals (30ms), so the cache is only trashed once per 30msperiod. In contrast, when run with the interactive workload *Chatty-CProbe*, applications suffer up to 4% performance loss, which *increases* with longer MRT guarantees. Investigating the scheduler traces showed that under zero MRT the batch workload enjoyed longer scheduler time slices of 30mscompared to the non-zero MRT cases. This was because under zero MRT highly interactive *Chatty-CProbe* quickly exhausted Xen's boost priority. After this, *Chatty-CProbe* could not preempt and waited until the running VCPU's 30mstime slice expires.

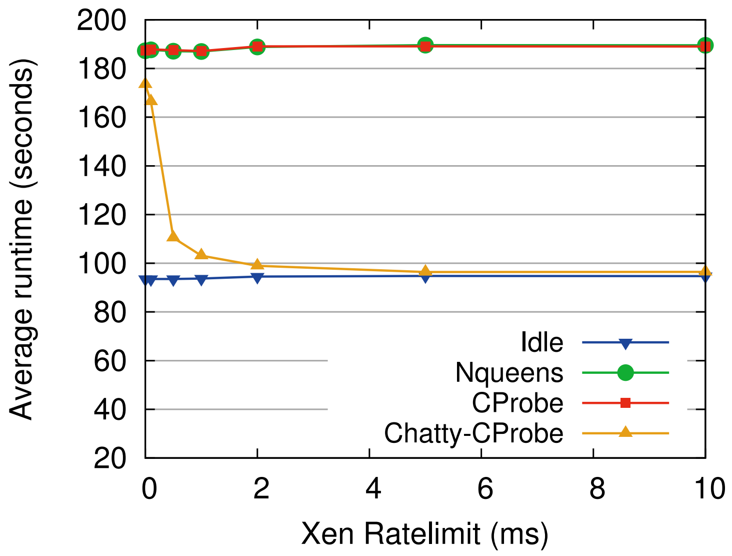


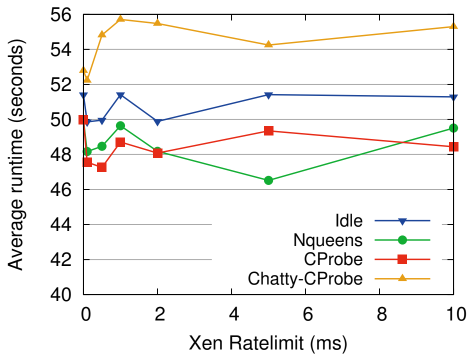
Figure 7.14: **Average runtime of *graph500* workload when run alongside various competing workloads and under work-conserving scheduling.** *Averaged over 5 runs.*

With longer MRT values, though, *Chatty-CProbe* continues to preempt and degrade performance more consistently.

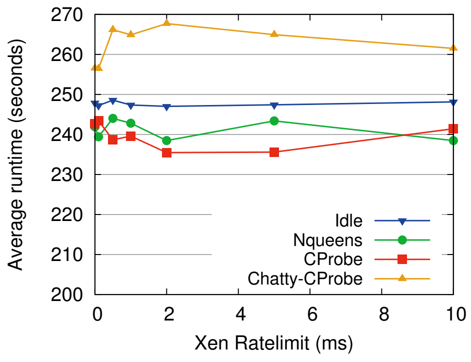
Another interesting observation in Figure 7.15 is that when the batch workloads share a PCPU with an idle VCPU, they perform worse than when paired with *Nqueens* or *CProbe*. Further investigation revealed that an idle VCPU is not completely idle but wakes up at regular intervals for guest timekeeping reasons. Overall, under non-work-conserving settings, running a batch VCPU with any interactive VCPU (even an idle one) is worse than running with another batch VCPU (even one like *CProbe* that trashes the cache).

7.3.4 System Performance

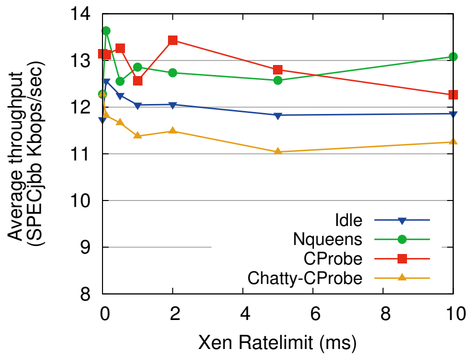
The preceding sections showed the impact of MRT guarantees when both applications are pinned to a single core. We next analyze the impact of



a: *mcf*



b: *graph500*



c: *SPECjbb*

Figure 7.15: **Average runtime of various batch workloads under non-work conserving setting.** *Note that for SPECjbb higher is better (since the graph plots the throughput instead of runtime). All data points are averaged across 5 runs.*

the Xen scheduler's VCPU placement policies, which choose the PCPU on which to schedule a runnable VCPU. We configure the system with 4 VMs each with 2 VCPUs to run on 4 PCPUs under a non-work conserving scheduler. We run three different sets of workload mixes, which together capture a broad spectrum of competing workload combinations. Together with a target workload running on both VCPUs of a single VM, we run: (1) *All-Batch* — consisting of worst-case competing CPU-hungry workload (*CProbe*); (2) *All-Interactive* — consisting of worst-case competing interactive workload (*Chatty-CProbe*); and (3) *Batch & Interactive* — where half of other VCPUs run *Chatty-CProbe* and half *CProbe*. We compare the performance of Xen without MRT to running with the default 5ms limit. The result of the experiment is shown in Figure 7.16. For interactive workloads, the figure shows the relative 95th percentile latency, while for CPU-hungry workloads it shows relative execution time.

On average across the three programs and three competing workloads, latency-sensitive workloads suffered on average of only 4% increase in latency with the MRT guarantee enabled. This contrasts sharply with the 5-fold latency increase in the pinned experiment discussed earlier. CPU-hungry workloads saw their performance improve by 0.3%. This makes sense given the results in the preceding section, which showed that an MRT guarantee offers little value to batch jobs in a non-work-conserving setting.

To understand why the latency performance is so much better than our earlier results would suggest, we analyzed a trace of the scheduler's decisions. With the non-work-conserving setting, Xen naturally segregates batch and interactive workloads. When an interactive VCPU receives a request, it will migrate to an idle PCPU rather than preempt a PCPU running a batch VCPU. As the PCPU running interactive VCPUs is often idle, this leads to coalescing the interactive VCPUs on one or more PCPUs while the batch VCPUs share the remaining PCPUs.

7.3.5 Summary

Overall, our performance evaluation shows that the strong security benefits described the in Section 7.2 can be achieved at low cost in virtualized settings. Prior research suggests more complex defense mechanisms [103, 115, 120, 179, 180, 193] that achieve similar low performance overheads but at a higher cost of adoption, such as substantial hardware changes or modifications to security-critical programs. In comparison,

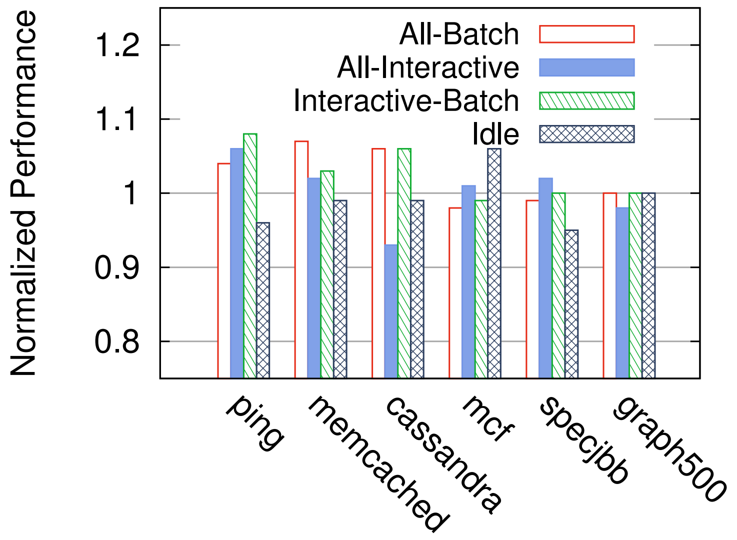


Figure 7.16: **Normalized Performance in a non-work-conserving configuration with 5msMRT.** *Normalized to performance under zero-MRT case. The left three workloads report* 95^{th} *percentile latency and the right three report runtime, averaged across 5 runs. In both cases lower is better.*

the MRT guarantee mechanism is simple and monotonically improves the security against many existing side-channel attacks with zero cost of adoption and low overhead.

We note that differences between the hypervisor and the OS scheduling mean that the MRT mechanism cannot be as easily applied by an operating system to defend against malicious processes. As mentioned above, a hypervisor schedules a small and relatively static number of VCPUS onto PCPUs. Thus, it is feasible to coalesce VCPUs with interactive behavior onto PCPUs separate from those running batch VCPUs. Furthermore, virtualized settings generally run with *share-based* scheduling, where each VM or VCPU is assigned a fixed share of CPU resources. In contrast, the OS scheduler must schedule an unbounded number of threads, often without assigned shares. Thus, there may be more oversubscription of PCPUs, which removes the idle time that allows interactive VCPUs to coalesce separately from batch VCPUs. As a result, other proposed defenses may still be applicable for non-virtualized systems, such as PaaS platforms that multiplex code from several customers within a single VM [86].

7.4 Integrating Core-State Cleansing

While the MRT mechanism was shown to be a cheap mitigation for protecting CPU-hungry workloads, it may not be effective at protecting interactive ones. If a (victim) VCPU yields the PCPU quickly, the MRT guarantee does not apply and an attacker may observe its residual state in the cache, branch predictor, or other hardware structures. We are unaware of any attacks targeting such interactive workloads, but that is no guarantee future attacks won't.

We investigate incorporating per-core state-cleansing into hypervisor scheduling. Here we are inspired in large part by the Düppel system [193], which was proposed as a method for guest operating systems to protect themselves by periodically cleansing a fraction of the L1 caches. We will see that by integrating a selective state-cleansing (SC) mechanism for I-cache, D-cache and branch predictor states into a scheduler that already enforces an MRT guarantee incurs much less overhead than one might expect. When used, our cleansing approach provides protection for all processes within a guest VM (unlike Düppel, which targeted particular processes).

7.4.1 Design and Implementation

We first discuss the cleansing process, and below discuss when to apply it. The cleanser works by executing a specially crafted sequence of instructions that together overwrite the I-cache, D-cache, and branch predictor states of a CPU core. A sample of these instructions is shown in Figure 7.17; these instructions are 27 bytes long and fit in a single I-cache line.

In order to overwrite the branch predictor or the Branch Target Buffer (BTB) state, a branch instruction conditioned over a random predicate in memory is used. There are memory move instructions that add noise to the D-cache state as well. The last instruction in the set jumps to an address that corresponds to the next way in the same I-cache set. This jump sequence is repeated until the last way in the I-cache set is accessed, at which point it is terminated with a `ret` instruction. These instructions and the random predicates are laid out in memory buffers that are equal to the size of the I-cache and D-cache, respectively. Each invocation of the cleansing mechanism randomly walks through these instructions to touch all I-cache sets, D-cache sets, and flush the BTB.

We now turn to how we have the scheduler decide when to schedule cleansing. There are several possibilities. The simplest strategy would be to check, when a VCPU wakes up, if the prior running VCPU was from another VM and did not use up its MRT. If so, then run the cleansing procedure before the incoming VCPU. We refer to this strategy as *Delayed-SC* because we defer cleansing until a VCPU wants to execute. This strategy guarantees to cleanse only when needed, but has the downside of potentially hurting latency-sensitive applications (since the cleanse has to run between receiving an interrupt and executing the VCPU). Another strategy is to check, when a VCPU relinquishes the PCPU before its MRT guarantee expires, whether the next VCPU to run is from another domain or if the PCPU will go idle. In either case, a cleansing occurs before the next VCPU or idle task runs. Note that we may do unnecessary cleansing here, because the VCPU that runs after idle may be from the same domain. We therefore refer to this strategy as *Optimistic-SC*, given its optimism that a cross-VM switch will occur after idle. This optimism may pay off because idle time can be used for cleansing.

```
000 <L13-0xd>:
  0: 8b 08           mov    (%rax),%ecx
  2: 85 c9           test   %ecx,%ecx
  4: 74 07           je     d <L13>
  6: 8b 08           mov    (%rax),%ecx
  8: 88 4d ff        mov    %cl,-0x1(%rbp)
  b: eb 05           jmp    12 <L14>
00d <L13>:
  d: 8b 08           mov    (%rax),%ecx
  f: 88 4d ff        mov    %cl,-0x1(%rbp)
012 <L14>:
 12: 48 8b 40 08     mov    0x8(%rax),%rax
 17: e9 e5 1f 00 00  jmpq   <next way in set>
```

Figure 7.17: **Instructions used to add noise.** *The assembly code is shown using X86 GAS Syntax. `%rax` holds the address of the random predicate used in the `test` instruction at the relative address `0x2`. The moves in the basic blocks `<L13>` and `<L14>` reads the data in the buffer, which uses up the corresponding D-cache set.*

Note that the CPU time spent in cleansing in Delayed-SC is accounted to the incoming VCPU but it is often free with Optimistic-SC as it uses idle time for cleansing when possible.

7.4.2 Evaluation

We focus our evaluation on latency-sensitive tasks: because we only cleanse when an MRT guarantee is not hit, CPU-hungry workloads will only be affected minimally by cleansing. Quantitatively the impact is similar to the results of Section 7.3 that show only slight degradation due to *Chatty-CProbe* on CPU-hungry workloads.

We use the hardware configuration shown in Table 7.2. We measured the standalone, steady state execution time of the cleansing routine as 8.4µs; all overhead beyond that is either due to additional cache misses that the workload experiences or slow down of the execution of the cleansing routine which might itself experience additional cache misses. To measure the overhead of the cleansing scheduler, we pinned two VCPUs of two different VMs to a single PCPU. We measured the performance of one of several latency-sensitive workloads running within one of these VMs, while the other VM ran a competing workload similar to *Chatty-CProbe* (but it did not access memory buffers when awoken). This ensured frequent cross-VM VCPU-switches simulating a worst case scenario for the cleansing scheduler.

We ran this experiment in four settings: no MRT guarantee (0ms-MRT), a $5ms$MRT guarantee (5ms-MRT), a $5ms$MRT with Delayed-SC, and finally a $5ms$MRT with Optimistic-SC. Figure 7.18 shows the median and 95^{th} percentile latencies under this experiment. The median latency increases between 10–50µscompared to the 5ms-MRT baseline, while the 95^{th} percentile results are more variable, and show at worst a 100μsincrease in tail latency. For very fast workloads, like *Ping*, this results in a 17% latency increase despite the absolute overhead being small. Most of the overhead comes from reloading data into the cache, as only 1/3rd of the overhead is from executing the cleansing code.

To measure overhead for non-adversarial workloads, we replaced the synthetic worst-case interactive workload with a moderately loaded Apache webserver (at 500 requests per second). The result of this experiment is not shown here as it looks almost identical to Figure 7.18, suggesting the choice of competing workload has relatively little impact on overheads. In this average-case scenario, we observed an overhead of 20–30µsacross all workloads for the *Delayed-SC* and 10–20 $\mu s$for *Optimistic-SC*, which is 10 μsfaster. Note that in all the above cases, the cleansing mechanism perform better than the baseline of no MRT guarantee with no cleansing.

To further understand the trade-off between the two variations of

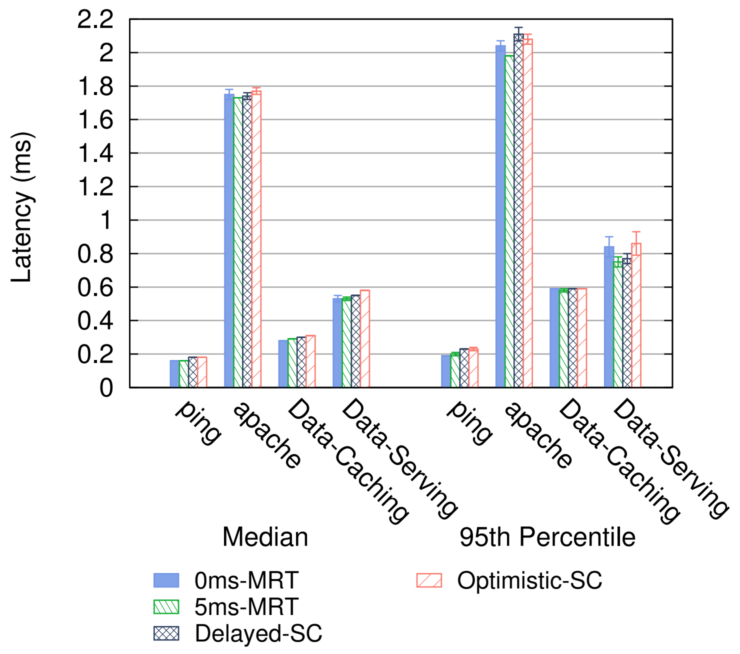
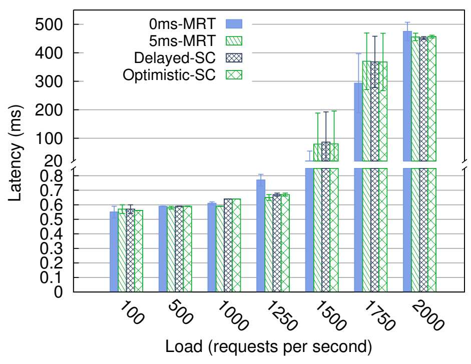


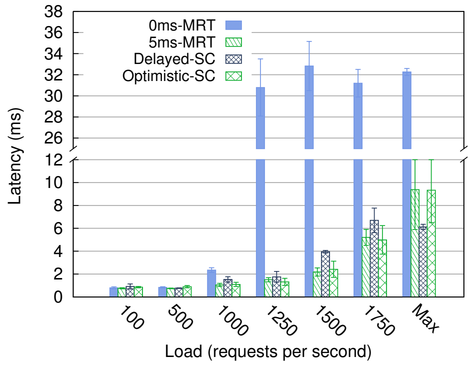
Figure 7.18: **Median and 95th percentile latency impact of the cleansing scheduler under *worst-case* scenario.** *Here all the measured workloads are feed by a client at 500 requests per second. The error bars show the standard deviation across 3 runs.*

state-cleansing, we repeated the first (worst-case) experiment above with varying load on the two latency-sensitive workloads, *Data-Caching* and *Data-Serving*. The 95th percentile and median latencies of these workloads under varying loads are shown in Figure 7.19 and Figure 7.20, respectively. The offered load shown on the x-axis is equivalent to the load perceived at the server in all cases except for *Data-Serving* workload whose server throughput saturates at 1870rps (this is denoted as *Max* in the graph).

The results show that the two strategies perform similarly in most situations, with optimization benefiting in a few cases. In particular, we see that the 95% latency for heavier loads on *Data-Serving* (1250, 1500, and 1750) is significantly reduced for Optimistic-SC over Delayed-SC. It

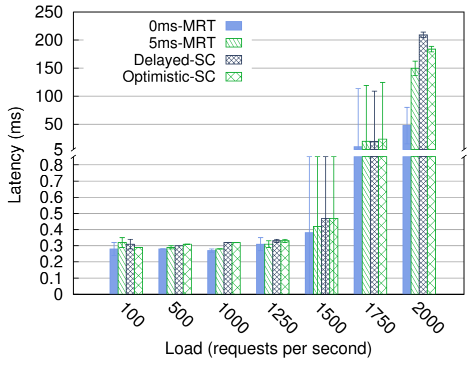


a: *Data-Caching*

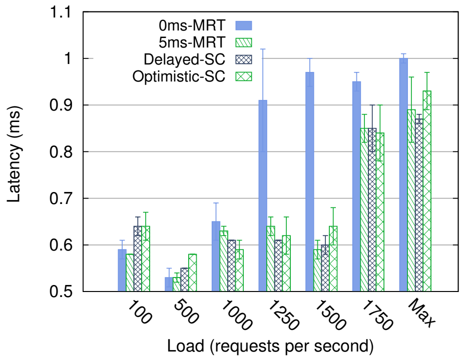


b: *Data-Serving*

Figure 7.19: 95th **percentile latency impact of the cleansing scheduler with varying load on the server.** *(a) Data-Caching and (b) Data-Serving. The error bars show the standard deviation across 3 runs.*



a: *Data-Caching*



b: *Data-Serving*

Figure 7.20: **Median latency impact of the cleansing scheduler with varying load on the servers.** *(a) Data-Caching and (b) Data-Serving. The error bars show the standard deviation across 3 runs.*

turned out that the use of idle-time for cleansing in Optimistic-SC was crucial for *Data-Serving* workload as the tens to hundreds of microsecond overhead of cleansing mechanism under the Delayed-SC scheme was enough to exhaust boost priority at higher loads. From scheduler traces of the runs with *Data-Serving* at 1500rps, we found that the VM running the *Data-Serving* workload spent 1.9s without boost priority under Delayed-SC compared to 0.8s and 1.1s spent under 5ms-MRT and Optimistic-SC, respectively (over a 120 long second run). The *Data-Serving* VM also experienced 37% fewer wakeups under Delayed-SC relative to 5ms-MRT baseline, implying less interactivity.

We conclude that both strategies provide a high-performance mechanism for selectively cleansing, but that Optimistic-SC handles certain cases slightly better due to taking advantage of idle time.

8 Related Work

In this chapter, we survey several prior research related to the three problems that are tackled in this dissertation, in their respective sections: 1. placement vulnerability (§ 8.1), 2. lack of performance isolation (§ 8.2), and 3. cross-VM side-channel attacks (§ 8.3).

8.1 Co-location in Public Clouds

Our work on understanding placement vulnerabilities in modern security-hardened public clouds (Chapter 5) derive inspiration from many related works. There are also subsequent works in this area that signifies the importance of this problem. We survey all those related works in this section.

8.1.1 VM Placement Vulnerability Studies

Ristenpart et al. [152] first studied the placement vulnerability in public clouds, which showed that a malicious cloud tenant could place one of his VMs on the same machine as a target VM with high probability. Placement vulnerabilities exploited in their study include publicly available mapping of VM's public/internal IP addresses, disclosure of Dom0 IP addresses, and a shortcut communication path between co-resident VMs. Their study was followed by Xu et al. [184] and further extended by Herzberg et al. [88]. However, the results of these studies have been outdated by the recent development of cloud technologies, which is the main motivation of our work.

Concurrent with our work, Xu et al. [185] conducted a systematic measurement study of co-resident threats in Amazon EC2. Their focus, however, is in-depth evaluation of co-residency detection using network route traces and quantification of co-residence threats on older generation instances with EC2's classic networking [15] (prior to Amazon VPC). In contrast, we study placement vulnerabilities in the context of VPC on EC2, as well as on Azure and GCE. That said, the two studies are complementary and collectively strengthen the arguments made by each other.

8.1.2 Defenses Against Co-location

New VM placement policies to defend against placement attacks have been studied by Han et al. [80, 81] and Azar et al. [36]. Han et al. propose two different policies for reducing the chances of co-location of an adversary: 1. *game-theoretic defense*: use a pool of policies and select a policy at random or 2. place VMs that belong to a user in a smaller subset of machines instead of stripping each VM on different machines, thereby reducing the probability of co-location. Although, both these mechanisms increase the cost of a simple attacker, it is unclear whether their proposed policies work against the performance and reliability goals of public cloud providers. In fact, we consider a random placement policy as a more secure and apt reference policy for comparing different placement policies. Further, Han et al. themselves admit that the second placement policy could be circumvented by a sophisticated attacker who could judiciously use multiple accounts to increase his chance of co-location. On the other hand, Azar et al. take a formal approach to designing co-location resistance placement algorithm [36]. In their formal model, they consider both efficiency (as total amount of resources used) and security, which they define by introducing a formal notion of co-location resistance. This work establishes foundation to reason and design placement algorithms with strict notion of security against co-location attacks, although there is a long gap that needs to be filled to employ them in practice.

Recently, Moon et al. propose a cloud-provider-assisted migration mechanism, where they use a moving target philosophy to bound information leakage via co-location of any single tenant [127]. First, they start by formalizing information leakage due to co-residency between different user VMs and propose four models, one model for each type of victim (Replicated vs. Non-replicated) and adversary (Collaborative vs. Non-collaborative). Second, they design an efficient and scalable algorithm that decides when and where to migrate a VM, taking into account the history of migration and resource needs for migration. But, as mentioned earlier VM migration is often expensive and involve non-negligible downtime for a commercial application. Further, as the Moon et al. point out, a faster attack may still succeed under this defense. Nevertheless, this is a significant step in the right direction and will greatly improve security of the cloud infrastructure against malicious users.

8.1.3 Co-residency Detection Techniques

Techniques for co-residency detection have been studied in various contexts. We categorize these techniques into one of the two major classes: side-channel approaches to detecting co-residency with *uncooperative* VMs and covert-channel approaches to detecting co-residency with *cooperative* VMs.

Side-channel based detection. Side-channels allow one party to exfiltrate secret information from another; hence these approaches may be adapted in practical placement attack scenarios with targets not controlled by the attackers. Network round-trip timing side-channel was used by Ristenpart et al. [152] and subsequently others [61, 168] to detect co-residency.

Zhang et al. [190] developed a system called *HomeAlone* to enable VMs to detect third-party VMs using timing side-channels in the last level caches. Bates et al. [43] proposed a side-channel for co-residency detection by causing network traffic congestion in the host NICs from attacker-controlled VMs; the interference of target VM's performance, if the two VMs are co-resident, should be detectable by remote clients. However, none of these approaches works effectively in modern cloud infrastructures for reasons mentioned earlier (in Section 4.1).

Covert-channel based detection. Covert-channels, a topic studied since the 1970s [112], are secretive communication channels that involve two colluding parties. Covert-channels on shared hardware components can be used for co-residency detection when both VMs under test are cooperative. Coarse-grained covert-channels in CPU caches and hard disk drives were used in Ristenpart et al. [152] for co-residency confirmation. Xu et al. [184] established covert-channels in shared last level caches between two colluding VMs in the public clouds. Wu el al. [182] exploited memory bus as a covert-channel on modern x86 processors, in which the sender issues atomic operations on memory blocks spanning multiple cache lines to cause memory bus locking or similar effects on recent processors. However, covert-channels proposed in the latter two studies were not designed for co-residency detection, while those developed in our work are tuned for this purpose. Concurrent to our work, Zhang et al. also used the same memory-covert channel in their recent work on placement vulnerability study [185].

Recently (in 2015), Inci et al. explored several techniques to detect co-residency including techniques suggested in many prior works and concluded that an LLC based covert-channel was the most reliable technique among the lot [93]. Note that an LLC-based covert-channel may not be effective in detecting all VM co-locations running on a multi-socket multi-core machine as LLCs are a per-socket resource. In contrast, we discovered an uncooperative co-residency detection mechanism that depends on creating and detecting contention on a shared memory bus, which is a system-wide shared resource.

Other techniques. Kohno et al. [108] demonstrated how clock skew in previous generation machines could be used to *fingerprint machines*. They utilize timestamps in TCP (or ICMP) network pings to remotely identify a unique machine (or differentiate two machines). Such a technique would also be applicable to virtualized platform where hypervisors give native access to the system clock as virtualizing system clock adds non-trivial overheads. An attacker could cleverly adapt this mechanism to detect co-residency by comparing the machine fingerprints of the two VMs under test. Although, analysis of clock-skew is non-trivial, the advantage of this approach over the others is that the attacker only require a network address and a publicly accesible TCP endpoint (e.g. ssh) on VMs under test. Despite these advantages, because of the complexity of this technique, we use a simpler alternative technique in our study.

8.2 Lack of Performance Isolation in Public Clouds

Our work on Resource-Freeing Attack (RFA) (Chapter 6) builds on past work surveying the performance interference of virtual machines, hardware and software techniques for improving performance isolation, side-channel attacks, and scheduler vulnerabilities. Many of these works are also related to our work on scheduler-based defenses against cross-VM attacks that aim to improve performance isolation (Chapter 7).

8.2.1 Performance Interference

Numerous works have found severe performance interference in cloud computing platforms [113, 144, 155, 177]. Our study of performance interference focuses more on the worst-case interference in a controlled setting than on the actual interference in cloud platforms. In addition, we measure the interference from pairs of different workloads rather than two instances of the same workload. Finally, our work looks at the impact of multicore scheduling by pinning VMs to a specific core.

8.2.2 Performance Isolation

There are numerous prior works that have attempted to reduce the effect of performance interference between competing workloads. Majority of these works have focused on performance- and efficiency-centric goals with limited or no focus on security implications of the system. Particularly, they either focus on Quality-of-Service (QoS) of a subset of applications (foreground or interactive) or aim to improve the datacenter utilization and sometimes both. Nevertheless, such works help reduce contention and hence the need for RFAs.

Contention for cache and processor resources is a major cause of performance loss, and many projects have studied resource-aware CPU schedulers that avoid contention [48, 98, 122, 196]. In cache/network contention, these schedulers may place the cache and network workloads on different packages to avoid affecting the cache. Similar work has been done at the cluster level to place jobs [39, 57, 114, 156, 162, 186, 189]. These systems attempt to place workloads that use non-interfering resources together or even to leave a processor idle if interference is bad. These cluster schedulers either do task placement apriori at the start of the task or detect interference and migrate tasks to a different machine. Some of them also try to throttle interfering low priority (background) tasks to reduce performance interference with a high priority (foreground) tasks. Although none of these systems are designed for public clouds, several of these techniques could be adopted to improve isolation in public clouds.

Beyond scheduling, software mechanisms can ensure performance isolation for many other hardware resources, including cache [148], disk [77], memory bandwidth [174] and network [158]. In addition to the software techniques, changes to low-level hardware have been proposed to better share memory bandwidth and processor caches [133, 147]. Many of the techniques that defend against side-channels (described in a later section, § 8.3.2) could also improve isolation.

Overall, all the above mechanisms would reduce the amount of contention and hence reduce the need and the benefit of RFAs.

8.2.3 Gaming Schedulers

The network/cache RFA works by forcing the scheduler to context switch at much coarser granularities than normal. Similar techniques have been used in the past to game schedulers in Linux [165] and Xen [194] in order to extend the timeslice of a thread. These techniques exploit the difference between the granularity of CPU allocation (cycles) and the granularity of accounting (timer ticks). There are also other attacks that enable Denial of Service on Linux Completely Fair Scheduler (CFS) [78]. Unlike these attacks, RFAs influence the scheduler to view an originally interactive workload as a CPU-bound workload, in the process influencing resource allocation that may benefit another task.

8.2.4 Side-channel Attacks

RFAs exploit the lack of isolation to boost performance. Several works demonstrated side-channel attacks through the shared LLC that can be used to extract information about co-resident virtual machines (e.g., [152, 184, 190]). More details in the following section.

8.3 Side-channel Attacks and Defenses

The work on soft-isolation to address information leakage across multi-tenant VM boundaries and improve performance isolation, also derives motivation from many related works. We survey works on side-channel attacks that motivate the problem we address (§ 8.3.1) and various proposed defenses which shows that our work takes an approach that has not been explored in the recent past (§ 8.3.2).

8.3.1 Attacks

Side-channel attacks can be classified into three types: time-, trace-, and access-driven. Time-driven attacks arise when an attacker can glean useful information via repeated observations of the (total) duration of a victim operation, such as the time to compute an encryption (e.g., [2, 47, 52, 83, 107]). Trace-driven attacks work by having an attacker continuously monitor a cryptographic operation, for example via electromagnetic emanations or power usage leaked to the attacker (e.g., [66, 106, 145].

In this dissertation, we focus on access-driven side-channel attacks, in which the attacker is able to run a program on the same physical server as the victim. These abuse stateful components of the system shared between attacker and victim program, and have proved damaging in a wide variety of settings, including [3, 78, 141, 143, 152, 187].

In cloud setting. In the cross-VM setting, the attacker and victim are two separate VMs running co-resident (or co-tenant) on the same server. The cross-VM setting is of particular concern for public IaaS clouds, where it has been shown that an attacker can obtain co-residence of a malicious VM on the same server as a target [152].

Zhang, Juels, Reiter, and Ristenpart (ZJRR) [191] demonstrated the first cross-VM attack with sufficient granularity to extract ElGamal secret keys from the victim. They use a version of the classic Prime+Probe technique [141]: the attacker first *primes* the cache (instruction or data) by accessing a fixed set of addresses that fill the entire cache. He then yields the CPU, causing the hypervisor to run the victim, which begins to evict the attacker's data or instructions from various cache. As quickly as possible, the attacker preempts the victim, and then *probes* the cache by again accessing a set of addresses that cover the entire cache. By measuring the speed of each cache access, the attacker can determine which cache lines were displaced by the victim, and hence learn some information about which addresses the victim accessed. The ZJRR attack builds off a long line of cross-process attacks (c.f., [3, 4, 78, 141, 143]) all of which target per-core microarchitectural state. When simultaneous multi-threading (SMT) is disabled (as is typical in cloud settings), such per-core attacks require that the attacker time-shares a CPU core with the victim. Similar to the cross-process attack demonstrated by Bangerter et al. that abuse the Linux process scheduler [78], ZJRR uses inter-processor interrupts to make frequent observation of shared state.

Fewer attacks thus far have abused (what we call) *off-core state*, such as last-level caches used by multiple cores. Some off-core attacks are coarse-grained, allowing attackers to learn only a few bits of information (e.g., whether the victim is using the cache or not [152]). An example of a fine-grained off-core attack is the recent Flush+Reload attack of Yarom and Falkner [187]. Their attack extends the Bangerter et al. attack to instead target last-level caches on some modern Intel processors and has been shown to enable very efficient theft of cryptographic keys in both cross-process and cross-VM settings. However, like the Bangerter et al. attack, it relies on the attacker and victim having shared memory pages. This is a common situation for cross-process settings, but also arises in cross-VM settings should the hypervisor perform memory page deduplication. While several hypervisors implement deduplication, thus far no IaaS clouds are known to use the feature and so are not vulnerable.

Attack on live public clouds. Zhang et al. demonstrated first of its kind successful side-channel attack across tenant instances in a container-based PaaS cloud [192]. They showed how a malicious user could steal potentially sensitive application data like number of items in the shopping cart of an e-commerce application, hijack user accounts and break single sign-on user authentication applications across multi-tenant container boundary.

Recently, Inci et al. demonstrated cross-VM side-channel on modern implementation of RSA in a popular public clouds (EC2) [93], which apart from the actual side-channel attack included figuring out how to detect co-location, and reverse engineering the hardware LLC algorithm that maps an address to memory location. The latter is essential for speeding up the actual attack making it practical in live clouds. This further demonstrates the practicality of side-channel attacks in modern security-hardened cloud infrastructure.

8.3.2 Defenses

Hard isolation. An obvious solution is to prevent a successful side-channel attack is to avoid sharing hardware between attacker and victim tasks, which we call *hard isolation*. Partitioning the cache in hardware or software prevents its contents from being shared [103, 149, 157, 179, 180]. This requires special-purpose hardware or loss of various useful features (e.g., large pages) and thus limits the adoption in a public cloud environment. Similarly, one can allocate exclusive memory resources for a sensitive process [103] in the software. Such a mechanism requires identification of the sensitive application and hence is not a general-purpose solution. Assigning VMs to run on different cores avoids sharing of per-core hardware [101, 118, 164], and assigning them to different servers avoids sharing of any system hardware [152]. A key challenge here is identifying an attacker and victim in order to separate them; otherwise this approach reduces to using dedicated hardware for each customer, reducing utilization and thus raising the price of computing.

Another form of hard isolation is to reset hardware state when switching from one VM to another. For example, flushing the caches on every context switch prevents the cache state from being shared between VMs [193]. However, this can decrease performance of cache-sensitive workloads both because of the time taken to do the flush and the loss in cache efficiency.

Adding noise. Beyond hard isolation are approaches that modify hardware to add noise, either in the timing or by obfuscating the specific side-channel information. The former can be accomplished by removing or modifying timers [115, 120, 172] to prevent attackers from accurately distinguishing between microarchitectural events, such as a cache hit and a miss. For example, StopWatch [115] removes all timing side-channels and incurs a worse-case overhead of 2.8x for network intensive workloads. Specialized hardware-support could also be used to obfuscate and randomize processor cache usage [109, 180]. All of these defenses either result in loss of high-precision timer or require hardware changes.

An alternative to adding noise to the timing information, prior works have also attempted to add noise to the shared state. Düppel adds noise to the local processor cache state by flushing caches when sensitive applications run inside a VM [193]. Similarly, programs can be changed to obfuscate access patterns [49, 50]. These approaches are not general-purpose, as they rely on identifying and fixing all security-relevant programs. Worst-case overheads for these mechanisms vary from 6–7%.

Recently, Xiao et al. used differential privacy mechanisms to plug information leakage in shared pseudo filesystems like `procfs` [183], which are often used in attacks on a multi-user environment. This fixes a specific vulnerability in a system shared resource in a novel way but again is not a general purpose solution.

Smart scheduling. This dissertation is not the first to explore smart scheduler designs to improve security which includes defending against side-channel attacks. In 1992, Wei-Ming Hu proposed a new process scheduler called lattice scheduler [91] that plugs cache covert-channels. Processes under the lattice scheduler are tagged into access classes of varying secrecy and the scheduler avoids scheduling processes that resulted in a downward transition in secrecy, i.e., a regular process is scheduled after a highly sensitive (high secrecy) process. Our work is similar to this simple scheduler mechanism in the fact that any interaction other than intra-tenant (multiple VCPU of a single user VM) as sensitive and ratelimits such interaction.

Several past efforts attempt to minimize performance interference between workloads (e.g., Q-clouds [131], mClock [77] and Bubble-Up [118]), but do not consider adversarial workloads such as side-channel attacks.

9 Conclusion and Lessons Learned

Many enterprise applications [25–30, 34, 68, 124, 135] that are in day-to-day use by customers over the Internet are hosted in the public clouds. These applications run inside virtual machines that share the same host to run other VMs that belong to arbitrary users (multi-tenancy). The onus is on the cloud infrastructure to make sure that this sharing is only profitable and not insecure. Unfortunately, we show that the state-of-the-art cloud infrastructure do not provide sufficient isolation proving the thesis that: *"the practice of multi-tenancy in public clouds demands stronger isolation guarantees between VMs in the presence of malicious users."* In this chapter, we will revisit this thesis and show how the conclusion of the works (Chapter 5, 6 & 7) presented in this dissertation support this thesis.

We specifically focused on one of the major security threats in public clouds, co-location or cross-VM attacks and demonstrated how lack of isolation enable a malicious user to exploit such attacks in live clouds. In the first part of the dissertation we evaluated the placement policies used in public clouds for vulnerabilities that enable any user to influence co-location with a set of target victim VMs. Apart from evaluating the cluster scheduler, we explored how performance interference between co-located VMs could incentivize new attacks that help steal resources from neighboring VMs.

In the second part, we investigated a new design principle called soft-isolation that improve VM isolation without compromising on the efficiency of resource sharing between VMs. We demonstrated this soft-isolation principle with a simple CPU scheduler primitive, Minimum RunTime (MRT) guarantee, which limits dangerous cross-VM interactions between multi-tenant VMs. We showed that this simple modification to the hypervisor's CPU scheduler was sufficient to prevent many known cross-VM side-channel attacks.

In this chapter, we will summarize the contribution of these works (§ 9.1) and share some of the lessons learned in the process (§ 9.3) that might benefit any budding researcher and the community.

9.1 Summary

In this section, we will recall the important questions/problems that we motivated in Chapter 4 and see how the pieces of the dissertation answer them, consequently tying it back to the thesis of this dissertation.

(Q1) Are co-location attacks impractical in modern clouds?

We started by systematically analyzing the placement policy of three popular public clouds (Amazon EC2, Microsoft Azure and Google Compute Engine) for placement vulnerability [170]. We did this by observing placement behavior of the clouds' VM placement policy by simulating VM launch scenarios of both the victims and an attacker. In order to observe placement behavior we also required a mechanism to detect whether two VMs (that belong to different users) are co-located on the same host, as cloud providers do not expose this information. We showed that all prior published techniques no longer worked, and hence investigated new ways to reliably detect co-location with any victim VM. Overall, this extensive study on three clouds lead to two important conclusions. First, there exists adversarial launch strategy for all three public clouds that resulted in targeted co-location with the victim VMs. Although the launch strategy varied based on the cloud provider and their specific placement algorithm, the resulting launch strategies were extremely cheap and sometimes incurred as low as 14 cents in some clouds. Second, we found that many prior techniques to cheaply detect co-location no longer worked. We found a new technique that uses a shared-memory bus performance side-channel to detect co-location with victim VMs. We do this by just using victim VM's public interfaces (e.g. HTTP) even when they are part of a large multi-tiered cloud application.

In summary, the answer to the question is no, achieving co-location with a set of target victim VMs is practical as we demonstrated it on three public clouds. Our results demonstrate that even though cloud providers have massive datacenters with numerous physical servers, the chances of co-location are far higher than expected. PaaS clouds are no exception to these attacks. Counter-intuitive to conventional wisdom, it is also extremely cheap to do the attack in live clouds.

(Q2) Are there unique opportunities for malicious users to exploit the lack of performance isolation for monetary or performance gains?

After analyzing the cluster scheduler and its placement policy, we turned to analyze the (lack of) performance isolation in Hypervisors that multiplex multi-tenant VMs on the same machine. With a set of carefully designed microbenchmarks that stresses different per-host resources like processor cache, memory, network and disks, we showed that worst-case resource contention between two VMs for any two resources can degrade performance of one of the VMs by as high as 5×-6×. Understanding the extent of resource contention, taking the perspective of a malicious and greedy user, we set out to find ways to reduce this contention. We demonstrated a new class of attacks called Resource-Freeing Attacks (RFAs) [168], where the goal is to free up contention on a target resource that the attacker cares about. RFAs achieve this by interfering with the victim's performance by using their public interface to create a bottleneck on another resource that victim relies on and thereby freeing up the target resource. This was based on a simple observation that real-world workloads depend on more than one hardware resource for its performance. We demonstrated this attack on a realistic setting where a highly-loaded webserver (network-intensive) degrades the performance of a cache-sensitive workload by approximately 5×. Using RFAs and our understanding on how CPU schedulers handle an interactive and batch type workloads in VMs, we were able to reduce contention and improve performance by 60% in this scenario. For moderately cache-sensitive SPEC workloads, RFAs relieved contention by 66.5%, which translated into 6% performance improvement in a live and busy public cloud (Amazon EC2).

In summary, the answer to the question is yes, RFAs exploit lack of isolation for performance gain, which may in turn be translated into monetary gains as well. This work leads to two important conclusions: 1. in the presence of performance heterogeneity in public clouds because of interference (and hardware heterogeneity [61]), a simple and static pay-per-hour pricing model hugely incentivizes attacks like RFAs, 2. RFAs provide insight into design principle for resource schedulers – a purely work-conserving schedulers make RFAs hugely profitable. Hence a hybrid scheduler, where majority of the resources are reserved to VMs and only a fraction of the free resources are distributed between active VMs, gets the best of both worlds – efficiency via sharing and improved isolation for security.

(Q3) Can isolation be improved for security without compromising the fruits of efficiency via sharing?

In the second part of the dissertation, we sought to solve the problem of improving isolation, particularly, to thwart Cross-VM side-channel attacks that exploit lack of isolation for stealing secrets across the VM boundary. We observed that all the prior works either had high-overhead or required specialized hardware and hence were hard to deploy in the wild. We also observed that all the most effective Prime+Probe side-channels relied on Hypervisor's CPU scheduler allowing frequent cross-VM preemptions at an interval as low as $16\mu s$. This lead us to investigated the relationship between CPU scheduling policies and side-channel efficacy. We showed that a simple scheduler primitive called minimum runtime guarantee of $5ms$ ratelimited the dangerous cross-VM preemptions long enough such that it was sufficient to thwart all known cross-VM attacks [166]. With extensive performance evaluation we also showed that counter-intuitive to conventional wisdom this change did not affect any latency sensitive workloads. Complementary to this protection mechanism for batch victims, we also proactively designed a protective mechanism to avoid information leakage from interactive victim VMs using a low-overhead state-cleansing mechanism. Both of these mechanisms resulted in no overhead for the average-case and negligible overhead of 7% in the worst-case. This was possibly partly because of a smarter multi-core scheduler that avoided scheduling interactive and batch style VMs on the same core.

In summary, the answer is yes, with soft-isolation we were able to get best of both sharing and isolation by carefully scheduling VM interactions with the hypervisor's CPU scheduler.

9.2 Conclusion

Cross-VM co-location attacks still remain a problem in modern clouds amidst deployment of advanced virtualization techniques that improve isolation. This dissertation shows that public clouds need to be more aggressive when it comes to isolation as arbitrary users have several tools to achieve co-location and game schedulers because of lack of proper isolation. It is also important to note that, it is not only the software infrastructure that needs to be fixed to improve isolation, the design of machine architecture is also an equally significant player in providing isolation across VMs. Apart from playing devil's advocate against the state of security in modern clouds, we also demonstrate that striving to improve isolation without compromising on efficiency is also possible. At this point, it should come as no surprise that, *"the practice of multi-tenancy in public clouds demands stronger isolation guarantees between multi-tenant VMs in the presence of malicious users."*

9.3 Lessons Learned

Until this point of the dissertation, we presented a research endeavor that was smooth, coherent with no failures similar to a bed of roses (with thorny stems cut out). This is the section where we present the thorny stems that we encountered. As often famously quoted Thomas Alva Edison's saying goes:

> *"I have not failed. I've just found 10,000 ways that won't work."*
>
> – Thomas A. Edison

; we also record our failures here. Although it would be hard for Edison to publish a paper with that attitude as it would be received by a whole-hearted rejection from a modern conference program committee!

9.3.1 CPU Scheduling is Delicate

In the work on soft-isolation [166] we presented a simple modification to the CPU scheduler for preventing some classes of side-channels. In fact, the final proposal was a simple change in scheduler configuration as the Minimum RunTime (MRT) scheduler primitive was already present in both KVM (Linux CFS [126]) and Xen hypervisors [59]. Although it was introduced as a performance improvement of batch workloads, we showed that it also had good security properties that prevented information leakage via per-core shared state. Recall, under this MRT mechanism a vCPU or a VM cannot be preempted until a minimum scheduled runtime (denoted by the MRT value) is elapsed. Arriving at this relatively simple proposal was not a easy as it was preceded by numerous failures in trying to design a reasonably efficient CPU scheduler with the soft-isolation principle.

In this work, the main challenge is to find the right balance between preventing malicious side-channel workloads and affecting the performance of legitimate interactive workloads. Here side-channel workloads are highly preemptive (several orders higher, $10\mu s vs. 1ms$) than legitimate workloads.

We will describe the approach that eventually failed to performance as expected, below. We designed a mechanism which involved adaptively ratelimiting dangerous cross-VM preemptions at a fine-granularity and a VM migration scheduler to balance preemptions across different cores in the same host. We defined a preemption interval threshold called Leakage Vulnerability Window (LVW) below which a cross-VM preemption is vulnerable to information leakage. We also defined a threshold number of cross-VM preemptions (migration threshold) above which a migration is forced. The above two fine-grained and coarse-grained components of the preemption scheduler are guided by these two thresholds. These two mechanisms are described below:

- *Balancing preemption across available cores.* The idea here is instead of limiting preemption, moving highly preemptive vCPUs (which are scheduling units in a hypervisor's CPU scheduler) across multiple cores after the number of cross-VM preemptions exceed a threshold (migration threshold). Here we assume that each VM belongs to a different user and a preemption is potentially dangerous if it results in a context switch between two vCPUs of two disparate VMs (referred to as cross-VM preemption). This would limit the effect of any targeted side-channel attack on a victim VM. Considering moderns servers that at least have 8 to 16 cores with large shared LLCs, intuitively this approach would only marginally affect any legitimate latency sensitive workloads because of cache misses due to vCPU migration. Surprisingly, macro benchmarks faired poorly against the baseline with a nominal $1ms$ MRT. This is because of the fact that VCPU migrations are cheap only when done rarely and the cost of migration quickly became expensive for a moderately interactive legitimate workload.

- *Adaptive preemption ratelimiting.* The above failure suggested us to avoid frequent VM migration and we set out to design a fine-granular mechanism to provide soft-isolation by ratelimiting cross-VM preemptions. A straight-forward method would be use existing static MRT mechanism, but we believed an adaptively ratelimiting mechanism would further reduce overhead of a static MRT mechanism on legitimate interactive workloads. The goal of such a mechanism is to detect and penalize a highly preemptive VMs with higher MRT values and thereby scheduling cross-VM preemptions as a resource. There are several ways we can increase MRT value. One example is an exponential back-off style increasing of MRT by doubling the current MRT value if the VM indulges in frequent preemptions (i.e. preemption interval $< LVW$).

We experimented with several variations of the above scheduler with varying migration threshold and policies for learning the MRT value. Here, a higher migration threshold trades off isolation for performance by reducing the rate of vCPU migration. Similarly different policies for learning dynamically learning MRT value decide how quickly or accurately the system arrives at an ideal MRT value for a VM. After extensive experiments with microbenchmarks and real-world benchmarks we were surprised to learn that the payoff from using the two level preemption scheduler did not fair well against both the security and performance of the simple static MRT mechanism.

We learned two crucial lessons in this experiment:

1. Designing schedulers or even adding minor modifications to them without any regression is an art that is hard to get correct the first time.

2. Absence of standard and extensive tests to identify any regression made this even more difficult.

Apart from these lessons, we made two main observations on why this research endeavor failed.

Current CPU schedulers are extremely complex. It would not be inappropriate to equate CPU scheduler design to dark magic. One of the main lesson we learned from working with CPU schedulers is that intuition often fails in predicting the actual outcome in a scheduler. When schedulers work at very short time scales (tens of μs to tens of milliseconds), it is almost impossible to simulate the outcome with just one's intuition. Only handful of people with extensive experience in designing schedulers are sufficiently equipped to do this daunting task. This task of modifying existing schedulers is further exacerbated by the complexity of the state-of-the-art mature CPU schedulers. I took almost two months to get a sufficient understanding of the Linux Completely Fair Scheduler [126]. There are numerous corner cases that specializes scheduling designs for certain specific scenarios. To quantify the complexity of the scheduler subsystem, we translated the logic into plain English by sifting through the source code of the scheduler[1]. It took 7 two column pages in plain English to summarize the core scheduler logic [167]. This is not considering the interactions of the scheduler with other subsystems. Frustrated by this complexity, we switched to working with Xen hypervisor's CPU scheduler, which was relatively very simple. Nevertheless, the above adaptive cross-VM preemption scheduler did not perform well when implemented in Xen relative to the existing static MRT mechanism.

Batch vs. interactive – a never ending tension. Systems community for decades have been working on the problem of satisfying the complimentary needs of both batch and interactive workloads [54, 142, 153, 176]. Batch workloads benefit from longer scheduler time slices and are highly cache-sensitive. On the other hand, interactive workloads benefit from quicker wakeups and in general are not highly sensitive cache-hotness requiring shorter time slices.

Schedulers have tried to find the right balance between scheduling these two classes of workloads. Sliding this delicate balance a little further in either side adversely affects the performance of the other. Further, this coarse classification of workloads is not accurate, making this balance even more difficult. In this dissertation, we found that the security aspects of these schedulers make this balance trickier. *The balancing batch vs. interactive workload still remains as a hard problem when considering adversarial workloads like side-channels*. For example, Prime+Probe attackers abuse properties of interactive workloads to preempt batch like victim workloads frequently.

Identifying and separating these workload classes on separate cores in a multi-core system might alleviate this problem. In fact, one of the surprising revelations from the soft-isolation project [166] is that the existing multi-core scheduler in Xen did a good job in separating batch and interac-

[1]Currently there is no single document that completely explains the Linux CFS schedulers.

tive workloads on different cores [2]. This hugely reduced the negative effect of longer MRT values on interactive workload's wakeup latency. On the other hand, consequently this separation has a negative effect on utilization [42, 118]. Hence, balancing the batch and the interactive workloads remain a never ending problem.

9.3.2 Security Starts at the Hardware

There were also several failed attempts that did not satisfy publication and hence did not make it into this dissertation. We had instances were lack of proper hardware support made it hard to design a secure and efficient system on top of it.

For managing processor caches in software. One of the most difficult resources to manage in the Operating System are memory bandwidth and processor caches. An ardent reader might have noticed that this is the same resource that is also a major source of performance interference in virtualized environments (§ 6.2). The main reason for this is because there are no hardware support to account and control usage of both these resources. All prior work have worked around this problem either using paging to limit memory usage and page coloring to control cache usage [103, 148, 157] or hardware performance counters to account and CPU scheduling to control memory and cache usage [62, 63]. Both these workarounds only give coarse control over the resources and limits the extent to which software can effectively and efficiently manage them. This demonstrates the requirement for two main recommendations for hardware vendors: 1. accounting per task cache usage that enable monitoring task responsible for particular misses (self or other), 2. mechanisms to monitor, control memory bandwidth per task (and/or per core). It turns out that recently (at the time of writing this dissertation), Intel's newer

[2]It is not clear if this side-effect of the scheduler is intentional as we did not find any discussion or comment stating any such goal in the scheduler source code

generation processors (since Broadwell, 2015) provides hardware support and interfaces to manage and allocate last-level caches [94, 95]. Although there are still some limitations, it is a good first step in the right direction. To the best of our knowledge, there is no known support for allocation and management of the memory bandwidth.

Need for testing of hardware for security. When investigating various mechanisms to detect co-location, we stumbled up on a peculiar microarchitecture design bug in x86 processors that enabled a malicious user to degrade whole system performance by running a single instruction (refer to § 5.2.3). This was because of naïve implementation of an atomic instruction, where in order to provide atomicity over an unaligned memory address the hardware was designed to lock a globally shared memory bus. Note that an atomic operation over an unaligned address is unconventional and almost never occurs in modern applications. Although that may not stop an adversary from abusing these features for a malicious intent. Unfortunately, even after this discovery of this vulnerability it is surprisingly hard to detect these abuses efficiently in software.

The lesson we learn here is that security starts at the hardware and it is hard to expect designers at every layer to be security conscious. But, it is important to test the security aspects of these mechanisms (esp. hardware vendors) as well as it is not unreasonable to suspect the presence of other related side-channels that are waiting to be discovered and abused. In addition, it is high time we also incorporate security testing in every layer of the application stack.

Bibliography

[1] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop*, pages 110–124, August 2010.

[2] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286, February 2007.

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

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[5] Amazon cloudfront cdn, 2015. `http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html`.

[6] Amazon cloudwatch, 2015. `http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatch.html`.

[7] Amazon dynamodb, 2015. `https://aws.amazon.com/dynamodb/`.

[8] Amazon elastic block store (amazon ebs), 2015. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html`.

[9] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`.

[10] Amazon elastic compute cloud api reference, 2015. `http://docs.aws.amazon.com/AWSEC2/latest/APIReference`.

[11] Amazon elastic compute cloud api reference – runinstances, 2015. `http://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_RunInstances.html`.

[12] Amazon ec2 instance store. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html`.

[13] Amazon ec2 instances, 2015. `https://aws.amazon.com/ec2/instance-types/`.

[14] Amazon ec2 pricing, 2015. `http://aws.amazon.com/ec2/pricing/`.

[15] Amazon ec2 and amazon virtual private cloud: Differences between ec2-classic and ec2-vpc. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html#differences-ec2-classic-vpc`.

[16] Amazon elastic load balancer, 2015. `https://aws.amazon.com/documentation/elastic-load-balancing/`.

[17] Amazon glacier, 2015. `https://aws.amazon.com/glacier/`.

[18] Amazon found every 100ms of latency cost them 1% in sales, 2008. `http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/`

[19] Amazon Ltd. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`.

[20] Amazon relational database service (amazon rds), 2015. `http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`.

[21] Amazon simple storage service documentation, 2015. `https://aws.amazon.com/documentation/s3/`.

[22] Amazon Web Services. Extend your it infrastructure with amazon virtual private cloud. Technical report, Amazon, 2013.

[23] Aws innovation at scale, re:invent 2014, slide 9-10. `http://www.slideshare.net/AmazonWebServices/spot301-aws-innovation-at-scale-aws-reinvent-2014`.

[24] Apache libcloud. `http://libcloud.apache.org/`.

[25] Aws case study: Airbnb, 2015. `https://aws.amazon.com/solutions/case-studies/airbnb/`.

[26] Aws case study: Coinbase, 2015. `https://aws.amazon.com/solutions/case-studies/coinbase/`.

[27] Aws case study: Dow jones, 2015. `https://aws.amazon.com/solutions/case-studies/dow-jones/`.

[28] Aws case study: Expedia, 2015. `https://aws.amazon.com/solutions/case-studies/expedia/`.

[29] Aws case study: Intuit, 2015. `https://aws.amazon.com/solutions/case-studies/intuit-cloud-migration/`.

[30] Aws case study: Pfizer, 2015. `https://aws.amazon.com/solutions/case-studies/pfizer/`.

[31] Aws documentation: Dedicated instances, 2015. `http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/dedicated-instance.html`.

[32] Aws elastic beanstalk. `http://aws.amazon.com/elasticbeanstalk/`.

[33] Aws case study: Us food and drug administration (fda), 2015. `https://aws.amazon.com/solutions/case-studies/us-food-and-drug-administration/`.

[34] Aws case study: Healthcare.gov, 2015. `https://aws.amazon.com/solutions/case-studies/healthcare-gov/`.

[35] Aws case study: Nasa/jpl's mars curiosity mission, 2015. https://aws.amazon.com/solutions/case-studies/nasa-jpl-curiosity/.

[36] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. Co-location-resistant clouds. In *In Proceedings of the ACM Workshop on Cloud Computing Security*, pages 9–20, 2014.

[37] Windows azure. http://www.windowsazure.com/.

[38] Virtual machine and cloud service sizes for azure. https://msdn.microsoft.com/en-us/library/azure/dn197896.aspx.

[39] Mohammad Banikazemi, Dan Poff, and Bulent Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[40] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[41] Sean K. Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys*, 2010.

[42] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[43] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. Detecting co-residency with active traffic analysis techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, pages 1–12. ACM, 2012.

[44] Brian Beach. Virtual private cloud. In *Pro Powershell for Amazon Web Services*, pages 67–88. Springer, 2014.

[45] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[46] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[47] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[48] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *ICS*, 2010.

[49] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. In *Selected Areas in Cryptography*, pages 69–83. Springer, 2005.

[50] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[51] Neil Brown. Control groups series, 2014. `https://lwn.net/Articles/604609/`.

[52] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[53] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[54] L. Cherkasova, D.Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETERICS Performance Evaluation Review*, 25(2), September 2007.

[55] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[56] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, 2003.

[57] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[58] What is docker? `https://www.docker.com/what-docker`.

[59] George Dunlap. Xen 4.2: New scheduler parameters. http://blog.xen.org/index.php/2012/04/10/xen-4-2-new-scheduler-parameters-2/.

[60] Jake Edge. Denial of service via hash collisions. `http://lwn.net/Articles/474912/`, January 2012.

[61] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012.

[62] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[63] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[64] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[65] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[66] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 251–261, May 2001.

[67] Google app engine: Platform as a service. https://cloud.google.com/appengine/docs.

[68] Best buy slashes app development time and resources with google app engine, 2015. https://cloud.google.com/customers/best-buy/.

[69] The jre class white list, 2015. https://cloud.google.com/appengine/docs/java/jrewhitelist.

[70] Google cloud bigtable, 2015. https://cloud.google.com/bigtable/.

[71] Google compute engine docs: Transparent maintenance, 2015. https://cloud.google.com/compute/docs/zones#maintenance.

[72] Google cloud storage documentation, 2015. https://cloud.google.com/storage/docs/overview.

[73] Google compute engine. https://cloud.google.com/compute/.

[74] Google compute enginer autoscaler. http://cloud.google.com/compute/docs/autoscaler/.

[75] Google compute engine – disks. https://cloud.google.com/compute/docs/disks/.

[76] Graph 500 benchmark 1. http://www.graph500.org/.

[77] Ajay Gulati, Arif Merchant, and Peter J. Varma. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, 2010.

[78] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Security and Privacy, 2011 IEEE Symposium on*, 2011.

[79] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[80] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher Leckie. Security games for virtual machine allocation in cloud computing. In *Decision and Game Theory for Security*. Springer International Publishing, 2013.

[81] Yi Han, J. Chan, T. Alpcan, and C. Leckie. Virtual machine allocation policies against co-resident attacks in cloud computing. In *IEEE International Conference on Communications,*, 2014.

[82] Haproxy: The reliable, high performance tcp/http load balancer. `http://www.haproxy.org/`.

[83] M. Weißand B. Heinz and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, February 2012.

[84] J. L. Henning. Spec cpu2006 benchmark descriptions. In *SIGARCH Computer Architecture News*, 2006.

[85] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[86] Heroku PaaS system. `https://www.heroku.com/`.

[87] Heroku devcenter: Dynos and the dyno manager, ip addresses. `https://devcenter.heroku.com/articles/dynos#ip-addresses`.

[88] Amir Herzberg, Haya Shulman, Johanna Ullrich, and Edgar Weippl. Cloudoscopy: Services discovery and topology mapping. In *2013 ACM Workshop on Cloud Computing Security Workshop*, pages 113–122, 2013.

[89] Hpe helion. `http://www8.hp.com/us/en/cloud/helion-overview.html`.

[90] HTTPing. Httping client. `http://www.vanheusden.com/httping/`.

[91] Wei-Ming Hu. Lattice scheduling and covert channels. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, pages 52–61. IEEE, 1992.

[92] Hyperv architecture, 2015. https://msdn.microsoft.com/en-US/library/cc768520(v=bts.10).aspx.

[93] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/.

[94] Cache monitoring technology and cache allocation technology. http://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html.

[95] Intel Corporation. White paper: Improving real-time performance by utilizing cache allocation technology. Technical report, Intel, 2015.

[96] Intel Ivy Bridge cache replacement policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[97] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. Cryptology ePrint Archive, Report 2014/435, 2014. http://eprint.iacr.org/.

[98] Aamer Jaleel, Hashem H Najaf-Abadi, Samantika Subramaniam, Simon C Steely, and Joel Emer. Cruise: cache replacement and utility-aware scheduling. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 249–260. ACM, 2012.

[99] David Kanter. L3 cache and ring interconnect. http://www.realworldtech.com/sandy-bridge/8/.

[100] Paul A Karger and John C Wray. Storage channels in disk arm optimization. In *2012 IEEE Symposium on Security and Privacy*, pages 52–52. IEEE Computer Society, 1991.

[101] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. No-hype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[102] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM TOCS*, 10(4):338–359, November 1992.

[103] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12. USENIX Association, 2012.

[104] Keith Kirkpatrick. Software-defined networking. *Commun. ACM*, 56(9):16–19, September 2013.

[105] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[106] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397, August 1999.

[107] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – Crypto'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[108] T Kohno, A Broido, and K Claffy. Remote physical device fingerprinting. In *Security and Privacy, 2005 IEEE Symposium on*, pages 211–225. IEEE, 2005.

[109] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, pages 393–404. IEEE Computer Society, 2009.

[110] What is kubernetes? `http://kubernetes.io/v1.0/docs/whatisk8s.html`.

[111] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[112] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

[113] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *IMC*, 2010.

[114] J. Li, M. Qiu, J. Niu, W. Gao, Z. Zong, and X. Qin. Feedback dynamic algorithms for preemptable job scheduling in cloud systems. In *WI-IAT*, 2010.

[115] Peng Li, Debin Gao, and Michael K. Reiter. Mitigating access-driven timing channels in clouds using stopwatch. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:1–12, 2013.

[116] Linux man page. xentrace(8). `http://linux.die.net/man/8/xentrace`.

[117] What is lxc?, 2015. `https://linuxcontainers.org/lxc/introduction/`.

[118] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11. ACM, 2011.

[119] David Marshall. Understanding full virtualization, paravirtualization, and hardware assist, 2007.

[120] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12. IEEE Computer Society, 2012.

[121] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[122] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys*, 2010.

[123] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[124] Ge healthcare delivers core customer solutions on the microsoft cloud, 2015. `https://customers.microsoft.com/Pages/CustomerStory.aspx?recid=12166`.

[125] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Avi Patel, and Muttukrishnan Rajarajan. A survey on security issues and solutions at different layers of cloud computing. *The Journal of Supercomputing*, 63(2):561–592, 2013.

[126] Ingo Molnar. Linux Kernel Documentation: CFS Scheduler Design. `http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[127] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1595–1606. ACM, 2015.

[128] David Mosberger and Tai Jin. httperfâŁ"a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[129] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Usenix Security Symposium*, 2007.

[130] Al Muller and Seburn Wilson. Virtualization with vmware esx server. 2005.

[131] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, 2010.

[132] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 391–394, 2005.

[133] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA*, 2007.

[134] Netcraft Ltd. August 2011 web server survey. `http://news.netcraft.com/archives/2011/08/05/august-2011-web-server-survey-3.html`, August 2011.

[135] Four reasons we choose amazonâŁ™s cloud as our computing platform, 2010. `http://techblog.netflix.com/2010/12/four-reasons-we-choose-amazons-cloud-as.html`.

[136] Olio workload. `https://cwiki.apache.org/confluence/display/OLIO/The+Workload`.

[137] Openshift red hat's platform as a service. `https://www.openshift.com/`.

[138] The openstack project, 2015. `https://wiki.openstack.org/wiki/Main_Page`.

[139] Openstack scheduling in kilo release, 2015. `http://docs.openstack.org/kilo/config-reference/content/section_compute-scheduler.html`.

[140] Openvz virtuozzo containers, 2015. `http://openvz.org/Main_Page`.

[141] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[142] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[143] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[144] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *CLOUD*, 2010.

[145] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *LNCS*, pages 200–210, September 2001.

[146] Rackspace. `http://www.rackspacecloud.com/`.

[147] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.

[148] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[149] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[150] Rightscale. `http://www.rightscale.com`.

[151] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off my cloud: exploring information leakage in third party compute clouds. In *CCS*, 2009.

[152] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[153] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[154] Eric C Rosen and Yakov Rekhter. Bgp/mpls ip virtual private networks (vpns). 2006.

[155] J. Schad, J. Dittrich, and J. Quiane-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *PVLDB*, 2010.

[156] B. Sharma, R. Prabhakar, S. Lim, M. T. Kandemir, and C. R. Das. Mrorchestrator: A fine-grained resource orchestration framework for hadoop mapreduce. Technical Report CSE-12-001, Pennsylvania State University, January 2012.

[157] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199. IEEE, 2011.

[158] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[159] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, 2002.

[160] SPEC. SPECjbb2005 - Industry-standard server-side Java benchmark (J2SE 5.0). Standard Performance Evaluation Corporation, June 2005.

[161] Specjbb2005. `http://www.spec.org/jbb2005/`.

[162] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proc. HotPowerWorkshop Power Aware Comput. Syst*, 2008.

[163] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[164] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[165] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *16th USENIX Security Symposium*, pages 1–18, 2007.

[166] V Varadarajan, T Ristenpart, and M Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium*. USENIX Association, 2014.

[167] Venkatanathan Varadarajan. Understanding linux's completely fair scheduler. `http://pages.cs.wisc.edu/~venkatv/`.

[168] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292. ACM, 2012.

[169] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, Washington, D.C., August 2015. USENIX Association.

[170] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. A placement vulnerability study in multi-tenant public clouds. *CoRR*, abs/1507.03114, 2015.

[171] G. Varghese, J. Bassett, R.E. Thomas, P. Higginson, G. Cobb, B.A. Spinney, and R. Simcoe. Virtual lans, May 6 2003. US Patent 6,560,236.

[172] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen (short paper). In Thomas Ristenpart and Christian Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, October 2011.

[173] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[174] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS*, pages 181–192, 1998.

[175] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI*, 2002.

[176] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[177] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *IEEE INFOCOM*, 2010.

[178] Liang Wang, Antonio Nappa, Juan Caballero, Thomas Ristenpart, and Aditya Akella. Whowas: A platform for measuring web deployments on IaaS clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 101–114. ACM, 2014.

[179] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[180] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[181] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[182] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[183] Qiuyu Xiao, Michael K Reiter, and Yinqian Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1582–1594. ACM, 2015.

[184] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[185] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium*, 2015.

[186] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 607–618. ACM, 2013.

[187] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.

[188] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.

[189] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 379–391, Prague, Czech Republic, 2013.

[190] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[191] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

[192] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[193] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13. ACM, 2013.

[194] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. arXiv:1103.0759v1 [cs.DC], March 2011.

[195] Fangfei Zhou, Manish Goel, Peter Desnoyers, and Ravi Sundaram. Scheduler vulnerabilities and attacks in cloud computing. *CoRR*, abs/1103.0759, 2011.

[196] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.

A

Appendix A

A.1 Modeling the Effects of MRTs

In this section we investigate a model that, while abstract, enables more formal reasoning about the security offered by the MRT guarantee for some classes of victims.

The model. Recall that in a preemption-driven side-channel attack, the attacker must interleave execution on a CPU core with the victim in order to frequently measure some shared resource such as CPU caches, branch predictors, or the like. Consider the example pseudo-code procedures in Figure A.1. The first includes a side-channel vulnerability because an attacker can infer the value of the secret using based on inferring which portions of a CPU data cache are accessed by a victim.

For either of the procedures, we can represent the side-channel leakage as a sequence of actions by the victim and which the attacker seeks to learn. Fix a finite alphabet of symbols $\Sigma = \{A, B, C, \dots\}$, each representing a microarchitectual operation on state that is time-shared between the attacker and victim. For example in $\textsc{Procedure}_1$ the possible actions are accesses to the table elements, e.g., A accessing the first element of F, B the second element, and so on. In $\textsc{Procedure}_2$, action A represents bringing the memory footprint of function F_a into the I-cache and B represents function F_b. An execution of the victim leads to a sequence of actions $S_1, S_2, \dots, S_n$ for some number n and with $S_i \in \Sigma$. The attacker's goal is to

```
Procedure PROCEDURE1($secret, ...$):
    Let $secret_1, ..., secret_n$ be $\subset$ $secret$
    Let $table[size]$ be any data array
    for $i \leftarrow 1$ to $n$ do
        $F(table[secret_i])$

Procedure PROCEDURE2($secret, ...$):
    Let $secret_1, ..., secret_n$ be $\subset$ $secret$
    for $i \leftarrow 1$ to $n$ do
        if $secret_i = x$ then
            $F_a(...)$
        if $secret_i = y$ then
            $F_b(...)$
```

Algorithm A.1: **Pseudo-code of two sample procedures that leaks control-flow and data-access information.** PROCEDURE1 *leaks data-access information and* PROCEDURE2 *leaks control-flow information.*

reconstruct (a portion of) the secret by learning (a subset of) the sequence of victim's actions.

To model preemption-based side-channel attacks, we fix a schedule of preemptions as well as a leakage model. For scheduling, we assume the attacker can arrange for itself to make an observation before the first state, at one of the first m states, at intervals of m states thereafter, and after the final state. Letting $\mathcal{O}$ denote an observation, we can represent a scheduling trace as a sequence such as $S_1, \mathcal{O}, S_2, S_3, \mathcal{O}, S_4, S_5, \mathcal{O}, \ldots$ (here $m = 2$). The value m is determined by the duration of victim operations (how long each one takes before the next state is initiated) as well as the scheduler's MRT mechanism. We assume the attacker knows where in the victim computation the preemption occurred (i.e., the index i of the state just set by the victim before the preemption). The attacker can force the victim to re-run (on the same input) repeatedly up to t times.

We consider the following model of leakage, which models shared state such as caches as *sticky*. Each preemption by the attacker leaks to it all types of operations executed (states visited) by the victim since the last observation, but not the order of the states visited or their number of times each state is visited. So if states A, C, A, B are visited between two observations the attacker learns $\{A, B, C\}$. This models many side-channels due to shared state, such as caches, in which an operation affects the shared state in a fixed way independent of the order of operations. Note that this is generous to the attacker, in that known attacks do not reveal a noise-free set of all states visited: operations or data may have overlapping effects obfuscating which of the kinds of operations occurred.

Now consider the victim of the ZJRR attack, shown in Figure 7.5 from Section 7.2. The attacker wants to extract code paths, distinguishing between $5 \to 6 \to 7 \to 8 \to 9$ and $5 \to 6$. The former reflects a one bit in the secret key, and involves square and reduce and multiply plus reduce operations. For simplicity, we treat the two code paths as the types of operations/states, so an execution is simply a sequence of values $b_1, \ldots, b_n$ over a binary alphabet.[1] In ElGamal the secret key, and so this sequence of states, is uniformly random.

Note that because we (conservatively) assume the attacker can schedule preemptions subject only to the rate limit, and that the attacker can make an observation before the first and after the last operation, which implies the attacker can trivially learn the first and last m bits of the key. The question is what can be learned about the remaining bits of the key, and here we can take advantage of the sticky model. In particular, consider for illustrative purposes a sequence of key bits b_1, b_2, b_3, b_4, b_5. Suppose that $m = 2$, then the adversary can learn all of the bits via a sequence by learning b_1 via an observation $\mathcal{O}, b_1, \mathcal{O}, b_2 \cdots, b_7$. From there the adversary can perform a sliding window of size $m = 2$, learning each subsequent

[1]This is not quite without loss, since in fact the two code paths require differing amounts of time, reflecting the timing side channel. Our focus is on access-driven side channels, so we ignore this subtlety. Future refinements to the model to include different timing of operations could address it.

bit by knowledge of the previous bit and the side-channel which gives whether both $0, 1$ are observed in a window, just 0, or just 1.

However, as soon as $m = 3$ the adversary cannot always learn the entire bit string. Consider observation schedule $\mathcal{O}, b_1, b_2, b_3, \mathcal{O}, b_4$. As before, the adversary can use previous observations to learn b_1 and b_2, but this observation will only leak b_3 in the case that $b_1 = b_2$. Otherwise, the second observation returns $\{0, 1\}$ and the bit could be either value. Likewise for observation schedule $b_1, b_2, \mathcal{O}, b_3, b_4, b_5, \mathcal{O}$ the value of b_3 is only learned if $b_4 = b_5$. Since bits are uniform, the probability that b_3 is not learnable is $1/4$.

We can provide an upper bound for arbitrary $2 < m \leqslant n$, as follows. The attacker can always learn bits b_1, b_2 and b_{n-1}, b_n by way of the reasoning above. For simplicity we allow the attacker to in fact learn the first m bits and the last m bits. If no m-bit substring is all zeros or all ones, then the attacker can learn no further bits with certainty. By a (loose) union bound, the probability that there exists any m-bit substring that is equal to all zeros or all ones is at most $2(n - m)/2^m$. If we set $m = 68$, which corresponds to the minimal value seen in experiments for $n = 2048$-bit exponents when run in the attack setting with the $1ms$ MRT (see Table 7.6), then this probability is at most 2^{-56}.

This argument is admittedly informal, but it provides intuition regarding how one might show how the MRT guarantee can prevent attacks (under some well-stated assumptions on the side-channel).