Bolt: I Know What You Did Last Summer... In the Cloud

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

Christos Kozyrakis
Stanford University
kozyraki@stanford.edu

Abstract

Cloud providers routinely schedule multiple applications per physical host to increase efficiency. The resulting interference on shared resources often leads to performance degradation and, more importantly, security vulnerabilities. Interference can leak important information ranging from a service's placement to confidential data, like private keys.

We present Bolt, a practical system that accurately detects the type and characteristics of applications sharing a cloud platform based on the interference an adversary sees on shared resources. Bolt leverages online data mining techniques that only require 2-5 seconds for detection. In a multi-user study on EC2, Bolt correctly identifies the characteristics of 385 out of 436 diverse workloads. Extracting this information enables a wide spectrum of previously-impractical cloud attacks, including denial of service attacks (DoS) that increase tail latency by 140x, as well as resource freeing (RFA) and co-residency attacks. Finally, we show that while advanced isolation mechanisms, such as cache partitioning lower detection accuracy, they are insufficient to eliminate these vulnerabilities altogether. To do so, one must either disallow core sharing, or only allow it between threads of the same application, leading to significant inefficiencies and performance penalties.

***CCS Concepts:** • **Security and privacy** → **Systems security**; • **Computer systems organization** → **Cloud computing**

***Keywords:** cloud computing; security; interference; isolation; datacenter; latency; denial of service attack; data mining

1. Introduction

Cloud computing has reached proliferation by offering *resource flexibility* and *cost efficiency* [3, 1, 28]. Cost efficiency is achieved through multi-tenancy, i.e., by co-scheduling multiple jobs from multiple users on the same physical hosts to increase utilization. However, multi-tenancy leads to interference in shared resources, such as last level caches or network switches, causing unpredictable performance [48, 16, 53]. More importantly, unmanaged contention leads to security and privacy vulnerabilities [39]. This has prompted significant work on side-channel [43, 70] and distributed denial of service attacks (DDoS) [30, 2, 70], data leakage exploitations [80, 43], and attacks that pinpoint target VMs in a cloud system [60, 78, 75, 32]. Most of these schemes leverage the lack of strictly enforced resource isolation between co-scheduled instances and the naming conventions cloud frameworks use for machines to extract confidential information from victim applications, such as encryption keys.

This work presents *Bolt*, a practical system that can extract detailed information about the type, functionality, and characteristics of applications sharing resources in a cloud system. Bolt uses online data mining techniques to quickly determine the pressure an application puts on each of several shared resources. We show that this information is sufficient to determine the framework type (e.g., Hadoop), functionality (e.g., DNN), and dataset characteristics of a co-scheduled application, as well as the resources it is most sensitive to. Bolt periodically collects statistics on the resource pressure applications incur and projects this signal against datasets from previously seen workloads. Since detection repeats periodically, Bolt accounts for changes in application behavior and can distinguish between multiple co-residents on the same physical host. We validate Bolt's detection accuracy with a controlled experiment in a 40-server cluster using virtualized instances. Out of 108 co-scheduled victim applications, including batch and real-time analytics, and latency-critical services, such as key-value stores and databases, Bolt correctly identifies the type and characteristics of 87% of workloads. We also used Bolt in a user study on Amazon EC2. We asked 20 users to launch applications of their preference on EC2 instances and not disclose any information on their type and characteristics to us. Bolt correctly labeled 277 out of 436 launched jobs and determined the resource characteristics of 385, even when more than 5 jobs share a physical host.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037703

The information obtained by Bolt makes several cloud attacks *practical* and *difficult to detect*. For example, we use Bolt to launch host-based DoS attacks that use information on the victim's resource sensitivity to inject carefully-crafted contentious programs that degrade the victim's performance. In the 40-server cluster, Bolt's DoS attack translates to a tail latency increase of up to 140× for interactive workloads. Unlike traditional DoS attacks that saturate compute and memory resources, Bolt maintains low CPU utilization, by only introducing interference in the most critical resources, making it resilient to DoS mitigation techniques, such as load-triggered VM migration. We have also used Bolt to launch resource freeing attacks (RFA) that force the victim to yield its resources to the adversary [67], and VM co-residency detection attacks that pinpoint where in a shared cluster a specific application resides [69]. We show that the information obtained through data mining is critical to escalate the impact of the attack, reduce its time and cost, and to make it difficult to detect.

Finally, we examine to what degree current isolation techniques can alleviate these security vulnerabilities. We analyze baremetal systems, containers, and virtual machines with techniques like thread pinning, memory bandwidth isolation, and network and cache partitioning. These are the main isolation techniques available today, and while they progressively reduce application detection accuracy from 81% to 50%, they are not sufficient to eliminate these vulnerabilities. We show that the only method currently available to reduce accuracy down to 14% is core isolation, where an application is only allowed to share cores with itself to avoid malicious co-residents. However, since application threads contend for shared on-chip resources performance degrades by 34% on average. Alternatively, if we allocate more cores per application to mitigate performance unpredictability, we end up with resource underutilization and cloud inefficiency. We hope that this study will motivate public cloud providers to introduce stricter isolation solutions in their platforms and systems architects to develop fine-grain isolation techniques that provide strong isolation and performance predictability at high utilization.

2. Related Work

Performance unpredictability is a well-studied problem in public clouds that stems from platform heterogeneity, resource interference, software bugs and load variation [12, 47, 62, 18, 17, 20, 54, 35, 21, 62, 59, 37]. A lot of recent work has proposed isolation techniques to reduce unpredictability by eliminating interference [58, 61, 45, 65, 64]. With Bolt, we show that unpredictability due to interference also hides security vulnerabilities, since it enables an adversary to extract information about an application's type and characteristics. Below we discuss related work with respect to cloud vulnerabilities, such as VM placement detection, DDoS, and side-channel attacks.

VM co-residency detection: Cloud multi-tenancy has motivated a line of work on locating a target VM in a public cloud. Ristenpart et al. [60] showed that the IP machine naming conventions of cloud providers allowed adversarial users to narrow down where a victim VM resided in a cluster. Xu et al. [74] and Herzberg et al. [32] extended this study, resulting, in part, in cloud providers changing their naming conventions, reducing the effectiveness of network topology-based co-residency attacks. Following this development Varadarajan et al. [69] evaluated the susceptibility of three cloud providers to VM placement attacks, and showed that techniques like virtual private clouds (VPC) render some of them ineffective. Similarly, Zhang et al. [78] designed HomeAlone, a system that detects VM placement by issuing side-channels in the L2 cache during periods of low traffic. Finally, Han et al. [31] proposed VM placement strategies that defend against placement attacks, although they are not specifically geared towards public clouds. With Bolt, we show that leveraging simple data mining techniques on the pressure applications introduce on shared resources increases the accuracy of VM co-residency detection significantly. Bolt does not rely on knowing the cloud's network topology or host IPs, making it resilient against mitigation techniques, such as VPCs.

Performance attacks: Once a victim application is located, an adversary can can negatively affect its performance. Distributed Denial of Service attacks [51, 23, 67, 34] in the cloud have increased in number and impact over the past years. This has generated a lot of interest in detection and prevention techniques [55, 14, 30]. Bakshi et al. [2], for example, developed a system that detects abnormally high network traffic that could signal an upcoming DDoS attack, while Crosby et al. [13] and Edge [22] proposed a new DoS attack relying on algorithmic complexity that drives CPU usage up. Finally, resource-freeing attacks (RFAs) also hurt a victim's performance, while additionally forcing it to yield its resources to the adversary [67]. While RFAs are effective, they require significant compute and network resources, and are prone to defenses, such as live VM migration, that cloud providers introduce to mitigate performance unpredictability due to resource saturation. In contrast, Bolt launches host-based attacks on the same machine as the victim that take advantage of the victim's resource sensitivity, and keep resource utilization moderate, therefore evading defense mechanisms.

Side-channel attacks: There are also attacks that attempt to extract confidential information from co-scheduled applications, such as private keys [81, 42, 76, 4]. Zhang et al. [80] proposed a system that launches side-channel attacks in a virtualized environment, and cross-tenant side-channel attacks in PaaS clouds [79]. Wu et al. [73], on the other hand, used the memory bus of an x86 processor to launch a covert-channel attack and degrade the victim's performance. On the defense side, Perez-Botero et al. [56] analyzed the vul-

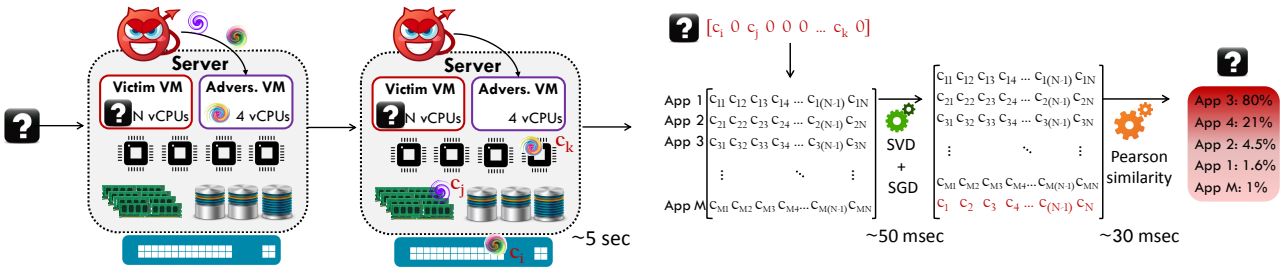


Figure 1: Overview of the application detection process. Bolt first measures the pressure co-residents place in shared resources, and then uses data mining to determine the type and characteristics of co-scheduled applications.

nerabilities of common hypervisors, and Wang et al. [70] proposed a system for intrusion detection in cloud settings, while Liu et al. [43] and Varadajaran et al. [68] designed scheduler-based defenses against covert- and side-channel attacks in the memory bus. The latter system controls the overlapping execution of different VMs and injects noise in the memory bus to prevent an adversary from extracting confidential information. Bolt does not rely on accurate microarchitectural event measurements through performance counters to detect application placement, and is therefore resilient to techniques that limit the fidelity of time-keeping and performance monitoring to thwart information leakage in side-channel attacks [49].

3. Bolt

3.1 Threat Model

Bolt targets IaaS providers that operate public clouds for mutually untrusting users. Multiple VMs can be co-scheduled on the same server. Each VM has no control over where it is placed, and no a priori information on other VMs on the same physical host. For now, we assume that the cloud provider is neutral with respect to detection by adversarial VMs, i.e., it does not assist such attacks or employ additional resource isolation techniques than what is available by default to hinder attacks by adversarial users. In Section 6 we explore how additional isolation techniques affect Bolt's detection accuracy.

Adversarial VM: An adversarial VM has the goal of determining the nature and characteristics of any applications co-scheduled on the same physical host, and negatively impacting their performance. Adversarial VMs start with zero knowledge of co-scheduled workloads.

Friendly VM: This is a normal VM scheduled on a physical host that runs one or more applications. Friendly VMs do not attempt to determine the existence and characteristics of other co-scheduled VMs. They also do not employ any schemes to prevent detection, such as memory pattern obfuscation [27].

3.2 Application Detection

Detection relies on inferring workload characteristics from the contention the adversary experiences in shared resources. For simplicity, we assume one co-scheduled victim job for now and generalize in Section 3.3. Figure 1 shows an overview of the system's operation.

Bolt instantiates an adversarial VM on the same physical host as the victim, friendly VM. Bolt uses its VM to run a few microbenchmarks of tunable intensity that each put progressively more pressure on a specific shared resource. The resources stressed by the microbenchmarks include on-chip resources, such as functional units, and different levels of the cache hierarchy, and off-chip resources, such as the memory, network and storage subsystems [15]. Each microbenchmark progressively increases its intensity from 0 to 100% until it detects pressure from the co-scheduled workload, i.e., until the microbenchmark's performance is worse than its expected value when running in isolation. The intensity of the microbenchmark at that point captures the pressure the victim incurs in shared resource i and is denoted c_i, where $i \in [1,N]$, $N=10$ and $c_i \in [0,100]$. The 10 resources are: L1 instruction and L1 data cache, L2 and last level cache, memory capacity and memory bandwidth, CPU, network bandwidth, disk capacity and disk bandwidth. Large values of c_i imply high pressure in resource i. For unconstrained resources, e.g., last level cache, 100% pressure means that the benchmark takes over the entire resource capacity. For resources constrained through partitioning mechanisms, e.g., memory capacity in VMs, 100% corresponds to taking over the entire memory partition allocated to the VM.

Bolt uses 2-3 microbenchmarks for profiling, requiring approximately 2-5 seconds in total. It randomly selects one core and one uncore benchmark to get a more representative snapshot of the co-resident's resource profile. If the core is not shared between the adversarial and friendly VMs, Bolt measures zero pressure in the shared core resource. It then adds a third microbenchmark for an additional uncore resource (shared caches, memory, network or storage).

We use this sparse signal to determine the application's pressure in all other resources, its type and characteristics. Online data mining techniques have recently been shown to effectively solve similar problems in cluster management by finding similarities between new unknown applications and previously seen workloads [19]. The Quasar cluster manager, for example, used collaborative filtering to find similar-

ities in terms of heterogeneity, interference sensitivity, and provisioning. The advantage of collaborative filtering is that it can identify application similarities without a priori knowledge of application types and critical features. Unfortunately this means that it is also unable to label the victim workloads and classify their functionality, making it insufficient for the adversary's purpose.

Practical data mining: Bolt instead feeds the profiling signal to a *hybrid* recommender using feature augmentation [9, 25] that determines the type and resource characteristics of victim workloads. The recommender combines *collaborative filtering* and *content-based similarity detection* [9, 29]. The former has good scaling properties, relaxed sparsity constraints and offers conceptual insight on similarities, while the latter exploits contextual information for accurate resource profile matching.

First, a collaborative filtering system recovers the pressure the victim places in non-profiled resources [16, 83]. The system relies on matrix factorization with singular value decomposition (SVD) [72] and PQ-reconstruction with stochastic gradient descent (SGD) [6, 38] to find similarities between the new victim application and previously-seen workloads. SVD produces three matrices, U, Σ and V. The singular values σ_i in Σ correspond to similarity concepts, such as the intensity of compute operations or the correlation between high network and disk traffic. Similarity concepts are in the order of the number of examined resources, and are ordered by decreasing magnitude in Σ. Large singular values reveal stronger similarity concepts (higher confidence in workload correlation), while smaller values correspond to weaker similarity concepts and are typically discarded during the dimensionality reduction by SVD. Matrix $U_{(m,r)}$ captures the correlation between each application and similarity concept, and $V_{(n,r)}$ the correlation between each resource and similarity concept.

Once we have identified critical similarity concepts and discarded inconsequential information, a content-based system that uses *weighted Pearson correlation* coefficients to compute the similarity between the resource profile u_j of a new application j and the applications the system has previously seen. Weights correspond to the values of the r more critical similarity concepts [1], and resource profiles to the rows of the matrix of left singular vectors U. Using traditional Pearson correlation would discard the application-specific information that certain resources are more critical for a given workload. Similarity between applications A and B is given by:

$$Weighted\ Pearson(A,B) = \frac{cov(u_A, u_B; \sigma)}{\sqrt{cov(u_A, u_A; \sigma) \cdot cov(u_B, u_B; \sigma)}} \quad (1)$$

where u_A is the correlation of application A with each similarity concept σ_i, $cov(u_A, u_B; \sigma) = \frac{\sum_i \sigma_i (u_{Ai} - m(u_A;\sigma))(u_{Bi} - m(u_B;\sigma))}{\sum_i \sigma_i}$

[1] We keep the r largest singular values, such that we preserve 90% of the total energy: $\sum_{i=1}^{r} \sigma_i^2 = 90\% \sum_{i=1}^{n} \sigma_i^2$.

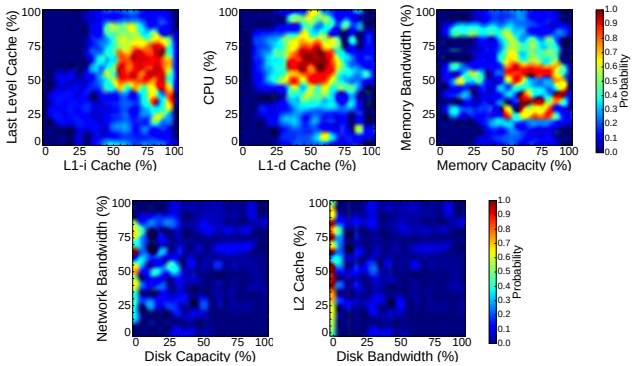


Figure 2: We show the correlation between the pressure in various resources and the probability of a co-scheduled application being memcached. For example, workloads with high LLC pressure and very high L1-i pressure have a high probability of being memcached.

the covariance of A and B under weights σ, and $m(u_A;\sigma) = \frac{\sum_i \sigma_i \cdot u_{Ai}}{\sum_i \sigma_i}$ the weighted mean for A. The output of the hybrid recommender is a distribution of similarity scores of how closely a victim resembles different previously-seen applications. For example, a victim may be 65% similar to a memcached workload, 18% similar to a Spark job running PageRank, 10% similar to a Hadoop job running an SVM classifier, and 3% to a Hadoop job running k-means. The 95^{th} percentile of the recommender's end-to-end latency is 80msec. Apart from application labels, this analysis yields information on the resources the victim is sensitive to, enabling several practical performance attacks (Section 5).

System insights from data mining: Before dimensionality reduction, each similarity concept corresponds to a shared resource. Different resources convey different amounts of information about a workload, and thus have different value to Bolt for detection. The magnitude of each similarity concept reflects how strongly it captures application similarities. For the controlled experiment of Section 3.4 that involves batch and interactive jobs, the resources with most value are the LL and L1-i caches, followed by compute intensity and memory bandwidth. While the exact order depends on the application mix, correlating similarity concepts to resources shows that certain resources are more prone to leaking information about a workload, and their isolation should be prioritized.

Finally, we show how resource pressure correlates to the probability that an application is of a specific type. Figure 2 show the probability that an unknown workload is a memcached instance with a read-mostly load and KB-range values, as a function of its measured resource pressure (details on methodology in Section 3.4). We decouple the 10 resources in 2D plots for clarity. From the heatmaps it becomes clear that cache activity is a very strong indicator of workload type, with applications with very high L1-i and high LLC pressure corresponding to memcached with a high probability. Disk traffic also conveys a lot of information,

with zero disk usage signaling a memcached workload with very high likelihood. Similar graphs can be created to fingerprint other application types. The high heat areas around the red regions of each graph correspond to memcached jobs with different rd:wr ratios and value sizes, and memory-bound workloads like Spark.

3.3 Challenges

Multiple co-residents: When a single application shares a host with Bolt, detection is straightforward. However, cloud operators colocate VMs on the same physical host to maximize the infrastructure's utility. Disentangling the contention caused by several jobs is challenging. By default Bolt uses two benchmarks - one core and one uncore - to measure resource pressure. If the recommender cannot determine the type of co-scheduled workload based on them (all Pearson correlation coefficients below 0.1), either the application type has not been seen before, or the resource pressure is the result of multiple co-residents. If at least one of the co-residents shares a core with Bolt, i.e., the core benchmark returns non-zero pressure, we profile with an additional core benchmark and use it to determine the type of co-runner. Because hyperthreads are not shared between multiple *active* instances, this allows accurately measuring pressure in core resources. The remainder of the uncore interference is used to determine the other co-scheduled workloads. This assumes a linear relationship between jobs for the resources of memory bandwidth and I/O bandwidth which can introduce some inaccuracies, but in practice does not significantly impact Bolt's detection ability (see Section 3.4).

Finally, there are occurrences where none of the co-scheduled applications share a core with Bolt. In this case, the system must rely solely on uncore resource pressure to detect applications. To disentangle the resource characteristics across co-residents, Bolt employs a *shutter profiling mode*, which involves frequent, brief profiling phases (10-50msec each) in uncore resources. The goal of this mode is to capture at least one of the co-scheduled workloads during a low-pressure phase, which would reveal the resource usage of only one of the co-residents, much like a high speed camera shutter attempts to capture a fast-moving object still. Figure 3 shows how shutter profiling works. When the shutter is open (upper left figure), thus profiling is off, Bolt cannot differentiate between the two victim VMs. When the shutter is closed during profiling, Bolt checks for changes in uncore resource pressure. If both victims are at high load differentiating between them is difficult (lower left). Similarly when both VMs are at low load (lower right). However, when only VM2 is at low load the pressure, Bolt captures is primarily from VM1, allowing the system to detect its type, and based on the remaining pressure also detect VM2. This technique is particularly effective for user-interactive applications that go through intermittent phases of low load; it is less effective for mixes of services with constant steady-state load, such as long-running analytics, or logging services. We will consider whether additional input signals, such as per-job cache miss rate curves, can improve detection accuracy for the latter workloads.

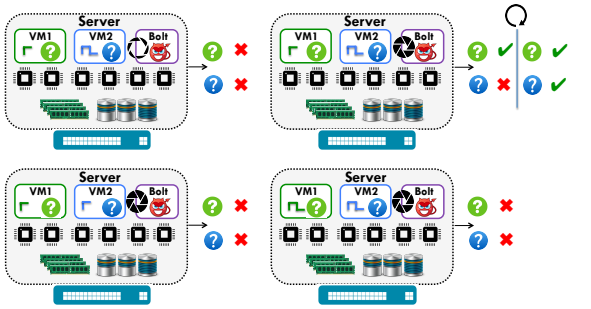


Figure 3: Shutter profiling mode.

Application phases: Datacenter applications are notorious for going through multiple phases during their execution. Online services in particular follow diurnal patterns with high load during the day and low load in the night [3, 50, 44]. Moreover, an application may not be immediately detected, especially during its initialization phase. Cloud users may purchase instances to run a service and then maintain them to execute other applications, e.g., analytics over different datasets. In these cases, the results of Bolt's detection can become obsolete over time. We address this by periodically repeating the profiling and detection steps for co-scheduled workloads. Each iteration takes 2-5 seconds for profiling and a few milliseconds for the recommender's analysis. Section 3.4 shows a sensitivity study on how frequently detection needs to happen.

3.4 Detection Accuracy

We first evaluate Bolt's detection accuracy in a controlled environment, where all applications are known. We use a 40-machine cluster with 8 core (2-way hyperthreaded) Xeon-class servers, and schedule a total of 108 workloads, including batch analytics in Hadoop [46] and Spark [77] and latency-critical services, such as webservers, memcached [26] and Cassandra [10]. For each application type, there are several different workloads with respect to algorithms, framework versions, datasets and input load patterns. Friendly applications are scheduled using a least-loaded (LL) scheduler that allocates resources on the machines with the most available compute, memory and storage. All workloads are provisioned for peak requirements to reduce interference. Even so, interference between co-residents exists, since the scheduler does not account for the sensitivity applications have to contention. In the end of this section, we evaluate how a scheduler that accounts for cross-application interference affects detection accuracy. The training set consists of 120 diverse applications that include webservers, various analytics algorithms and datasets, and several key-value stores and databases. The training set is selected to provide sufficient coverage of the space of resource characteristics. Figure 4 shows the pressure jobs in the training set put in compute, memory, storage, and network bandwidth.

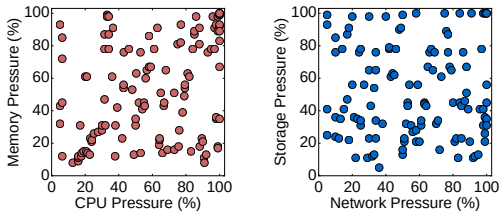


Figure 4: Coverage of resource characteristics for applications in the training set.

Applications	Detection accuracy (%)	
	Least Load scheduler	Quasar scheduler
Aggregate	87%	89%
memcached	78%	80%
Hadoop	92%	92%
Spark	85%	86%
Cassandra	90%	89%
speccpu2006	84%	85%

Table 1: Bolt's detection accuracy in the controlled experiment with the least loaded scheduler and Quasar.

While there are clusters of applications that saturate several resources, the selected workloads cover the majority of the resource usage space. This enables Bolt to match any resource profile to information from the training set, even if the new application has not been previously seen. Increasing the training set size further did not improve detection accuracy. Finally, note that there is no overlap between training and testing sets in terms of algorithms, datasets, and input loads.

In each server we instantiate an adversarial VM running Ubuntu 14.04. By default the VM has 4vCPUs (2 physical cores) to generate enough contention (Fig. 10 shows a sensitivity analysis on the VM size). The remainder of each machine is allocated to one or more friendly VMs. The max VM number, 5 in our setting, depends on the available servers. Friendly VMs run on Ubuntu 14.04 or Debian 8.0. Adversarial VMs have no a priori information on the number and type of their co-residents. Applications are allowed to share a physical core, but must be on different hyperthreads (vCPUs). While sharing a single hyperthread is common in private deployments hosting batch jobs in small containers [7], it is uncommon in public clouds, where 1 vCPU is the minimum guaranteed size of dedicated resources for non-idling instances.

Table 1 shows Bolt's detection accuracy, per application class, and aggregate. We signal a detection as correct if Bolt identifies correctly the framework (e.g., Hadoop) or service (e.g., memcached) the application uses, and the algorithm, e.g., SVM on Hadoop, or user load characteristics, e.g., read- vs. write-heavy load. We do not currently examine more detailed features, such as the distribution of individual query types. Bolt correctly identifies 87% of jobs, and for certain application classes like databases and analytics, the accuracy

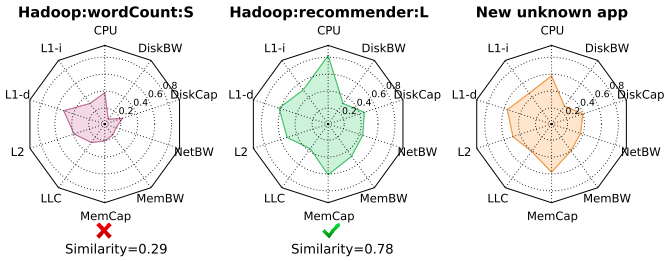


Figure 5: Star charts showing the resource profiles of two Hadoop and one unknown job. The closer the shaded area is to a vertex the higher the pressure in the specific resource.

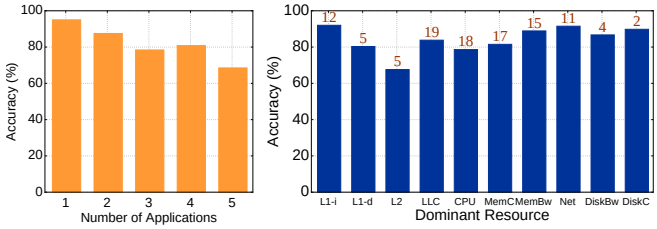


Figure 6: Detection accuracy as a function of the number of co-runners (left) and the apps' dominant resources (right).

exceeds 90%. Misclassified jobs are typically identified as workloads with the same or similar critical resources.

Per-application profiles: Note that, although in Table 1 we group applications by programming framework or online service, each framework *does not correspond to a single resource profile*. Profiles of applications within a framework can vary greatly depending on functionality, complexity, and dataset features. Bolt's recommender system matches resource profiles to specific algorithmic classes and dataset characteristics within each framework. Figure 5 shows an example where two Hadoop jobs, one running word count on a small dataset, and one running a recommender system on a very large dataset exhibit very different resource profiles. While the third application is also a Hadoop job it is identified as very similar to the recommender as opposed to word count.

Number of co-residents: The number of victim applications affects detection accuracy. Figure 6a shows accuracy as a function of the number of victims per machine. When the number of co-residents is less than 2, accuracy exceeds 95%. As the number increases accuracy drops, since it becomes progressively more difficult to differentiate between the contention caused by each workload. When there are 5 co-scheduled applications, accuracy is 67%. Interestingly, accuracy is higher for 4 than for 3 applications, since with 4 workloads the probability of sharing a core, and thus getting an accurate measurement of core pressure is higher. While aggressive multi-tenancy affects accuracy, large numbers of co-residents in public clouds are unlikely in practice [41].

Dominant resource: We now examine the correlation between the resource an application puts the most pressure on, and Bolt's ability to correctly detect it (Figure 6). The numbers on each bar show how many applications have each resource as dominant. In general, applications that are most easily detected are ones with high instruction cache (L1-i), memory bandwidth, network bandwidth and disk capacity pressure. These are workloads that fall in one of the following categories: latency-critical services with large codebases such as webservers, in-memory analytics like Spark, and disk-bound analytics like Hadoop. Interestingly, L2 activity is a poor indicator of application type, in contrast to L1 and LLC, since it does not capture a significant change in the working set size (from 32KB to 256KB for our platforms).

Number of iterations: In this controlled experiment, we stop the detection process upon correct identification of a workload. For several jobs, this requires more than one iterations of profiling and data mining. Figure 7a shows the fraction of workloads that were correctly-identified after N iterations. 71% of victims only require a single iteration, while an additional 15% required a second. A small fraction of applications needed more than two iterations, while jobs that are not identified correctly until the sixth iteration did not benefit from additional iterations. The number of applications per machine also affects the iterations required for detection (Figure 7b). When a single job is scheduled on a machine, one iteration is sufficient for almost all workloads. As the number of victims per machine increases additional iterations are needed to disentangle the contention signals of each co-scheduled job.

Figure 8 shows a case of detection over several iterations. The victim is a 4vCPU instance executing different consecutive jobs. Detection occurs every 20sec by default (see Fig. 10 for a sensitivity study on profiling frequency). After the initial iteration, Bolt detects the job as `mcf` from SPECCPU2006. In the next two iterations, the interference profile of the victim continues to fit `mcf`'s characteristics. At t=60sec, Bolt detects that the job profile has changed and now resembles an SVM classifier running on Mahout [46]. Similarly at t=180sec, the victim changes again to a Spark data mining workload. Changes are typically captured within a few seconds.

Resource pressure: Figure 9 shows the correlation between the pressure of a victim in a given resource and Bolt's ability to correctly detect it. We plot detection accuracy for three core and three uncore resources; the results are similar for the remaining four. In almost all cases, very low or very high pressure carries the most value for detection. For disk bandwidth, accuracy remains high except for the 20-50% region which contains many application classes with moderate disk activity, including analytics in Hadoop, databases and graph applications. Resource pressure also affects the number of iterations Bolt needs to correctly identify a workload. For resources with a lot of value for detection, such as

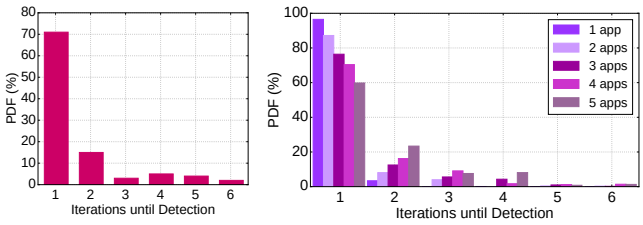


Figure 7: PDFs of iterations required for detection in total (left), and as a function of the co-residents number (right).

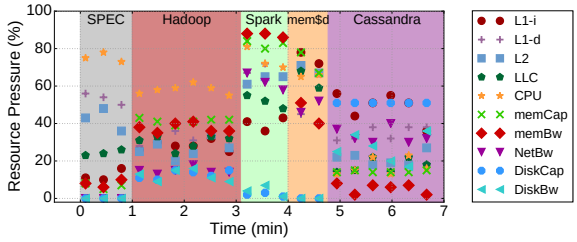


Figure 8: Example of workload phase detection by Bolt.

the L1-i and LLCs, when pressure is moderate several iterations are needed for accuracy to increase. In contrast, for off-chip resources like network bandwidth this effect is less pronounced. In general, when moderate pressure hurts detection accuracy (Figure 9), more iterations are needed for correct identification.

Scheduler: So far we have used a least loaded scheduler, which is commonly-used in datacenters [33, 63]. This scheduler does not account for resource contention between applications, leading to suboptimal performance [16, 24, 5, 84, 82]. Recent work has shown that if interference is accounted for at the scheduler level both performance and utilization improve [48, 19, 53]. We now evaluate the impact of such a scheduler on Bolt's detection accuracy. Quasar [19] leverages machine learning to quickly determine which applications can be co-scheduled on the same machine without destructive interference. We use Quasar to schedule all victim applications and then inject Bolt in each physical host for detection. In a real setting Bolt can trick an interference-aware scheduler by incurring very low levels of interference initially, until it is co-scheduled with a victim application and can determine its type and characteristics.

Table 1 shows Bolt's accuracy with the least loaded scheduler (LL) and Quasar. Interestingly, accuracy increases slightly with Quasar, 2% on average, but in general the impact of the scheduler is small. There are two reasons for the increase; first, both LL and Quasar do not share a single hyperthread between jobs, so core pressure measurements remain accurate. Second, because Quasar only co-schedules jobs with different critical resources, it provides Bolt with a less "noisy" interference signal for uncore resources, making distinguishing co-residents easier. Therefore reducing interference in software alone is not sufficient to mitigate these security threats.

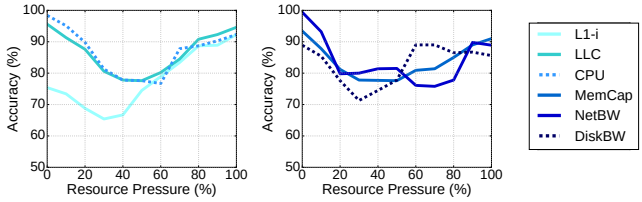


Figure 9: Detection accuracy as a function of the pressure victims place in various shared resources.

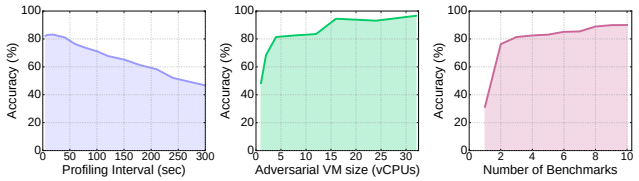


Figure 10: Sensitivity to: (a) profiling frequency, (b) adversarial VM size, and (c) profiling benchmarks.

Sensitivity analysis: Finally, we examine how design decisions in Bolt affect detection. Figure 10a shows how accuracy changes as profiling frequency decreases. For profiling intervals beyond 30 sec accuracy drops rapidly. If profiling only occurs every 5 minutes almost half the victims are incorrectly identified. Note that if workloads are long-running and mostly-stable, profiling can be less frequent without such an impact on accuracy. Figure 10b shows accuracy as a function of the adversarial VM's size. We examine sizes offered as on-demand instances by EC2 [1]. If the adversary has fewer than 4 vCPUs, its resources are insufficient to create enough contention to capture the co-residents' pressure. Accuracy continues to grow for larger sizes, however, the larger the adversarial instance is the less likely it will be co-scheduled with other VMs, nullifying the value of Bolt. Finally, Figure 10c shows accuracy as a function of the number of microbenchmarks used for profiling. A single benchmark is not sufficient to fingerprint the characteristics of a workload, however, using more than 3 benchmarks has diminishing returns in accuracy. Unless otherwise specified, we use 20sec profiling intervals, 4 vCPU adversarial VMs and 2 benchmarks for initial profiling.

3.5 Limitations

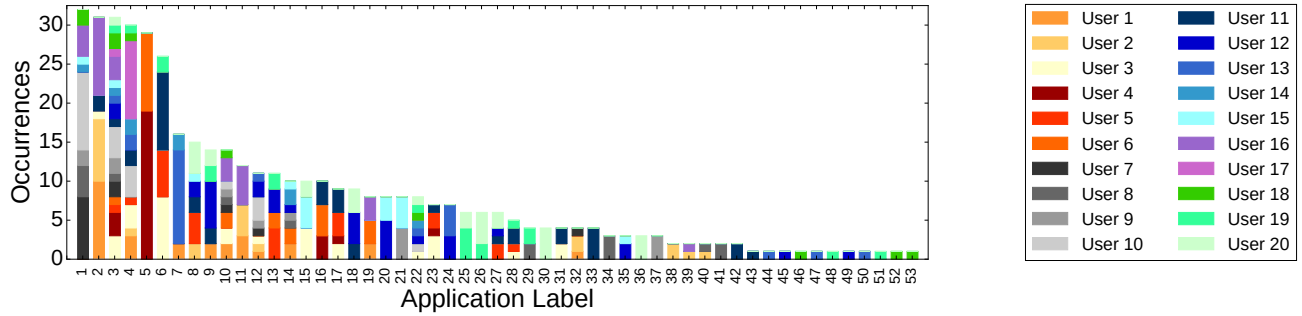
While Bolt can accurately detect most frequently-run workloads, it has certain limitations. First, when no job shares a core with the adversary, the system assumes linear relation between the co-residents' resource pressure in uncore resources. While this is true in some cases, it may not be generally accurate. We will explore alternative ways of distinguishing co-residents in future work. Second, the system cannot differentiate between multiple jobs running in a single instance and multiple instances running one job each. While this does not affect detection, if a covert-channel attack is aimed at a particular user, more accurate information would increase the adversary's leverage. Similarly, Bolt cannot currently distinguish between two copies of the same job sharing a platform at low load and one copy of the job that runs at higher load.

4. Cloud User Study

We now evaluate Bolt in a real-world environment with multiple users and unknown applications. We conducted a multi-user study on Amazon EC2 subject to the following rules:

- We have engaged 20 independent users from two academic institutions (Cornell University and Stanford University), provided them with access to a shared cluster of 200 c3.8xlarge instances on EC2, with 32 vCPUs and 60GB of memory each, and asked them to submit several applications of their choosing.
- On each instance we maintain a 4-vCPU VM for Bolt. All other resources are made available to the users as VMs.
- Each user submits one or more applications and all users have equal priority, i.e., no application will be preempted to accommodate the job of another user.
- Users were instructed to behave amenably, i.e., to avoid consuming disproportionate resources, so that the system can observe a representative job sample from all users.
- Users were also advised to pin their applications to specific cores to eliminate interference from the OS scheduler. Physical cores can be shared across jobs, however, each vCPU (hardware thread) is dedicated to a single application.
- Users can optionally select the instance(s) they want to launch their applications on. We provide a spreadsheet with a list of all available instances and their utilization at each point in time. This information, as well as any a priori information on the type, number, and characteristics of submitted applications is concealed from Bolt.
- If a user does not select an instance, a least-loaded scheduler selects the VM with the most available cores and memory.

Once users start launching jobs in the cluster, we instantiate Bolt on each machine and activate it periodically to detect the type, characteristics, and number of co-scheduled applications. The entire experiment lasts approximately 4 hours, beyond which point all instances are terminated. Figure 11 shows the PDF of application types the different users launched, provided by the users after the completion of the experiment. Different colors correspond to different users. A total of 436 jobs were submitted to the cluster. The application mix includes analytics jobs in Hadoop and Spark, scientific benchmarks (Parsec, Bioparallel), hardware synthesis tools (Cadence, VivadoHLS), multicore (zsim) and n-body simulations, email clients, web browsers, web servers (http), music and video streaming services, and databases, among others. Each job may take one or more vCPUs. Even though



1:hadoop	*5:cadence*	*9:MLPython*	*13:spec*	*17:parsec*	*21:postgres*	*25:ppt*	*29:du -h*	*33:cpu_burn*	*37:html*	*41:cp/mv*	*45:rsync*	*49:rm*	*53:ix*
2:spark	*6:zsim*	*10:make*	*14:matlab*	*18:vim*	*22:musicStream*	*26:OS img*	*30:cr/del cgroup*	*34:audacity*	*38:cassandra*	*42:sirius*	*46:ping*	*50:skype*	
3:email	*7:video*	*11:mem$d*	*15:mysql*	*19:scala*	*23:minebench*	*27:pdfview*	*31:bioparallel*	*35:javascript*	*39:mongoDB*	*43:oProfile*	*47:photoshop*	*51:zipkin*	
4:browser	*8:latex*	*12:http server*	*16:vivado*	*20:php*	*24:n-body sim*	*28:scons*	*32:storm*	*36:create VMs*	*40:mkdir*	*44:dwnld LF*	*48:ssh*	*52:graphX*	

Figure 11: Probability distribution function of applications launched in the cloud study, per user.

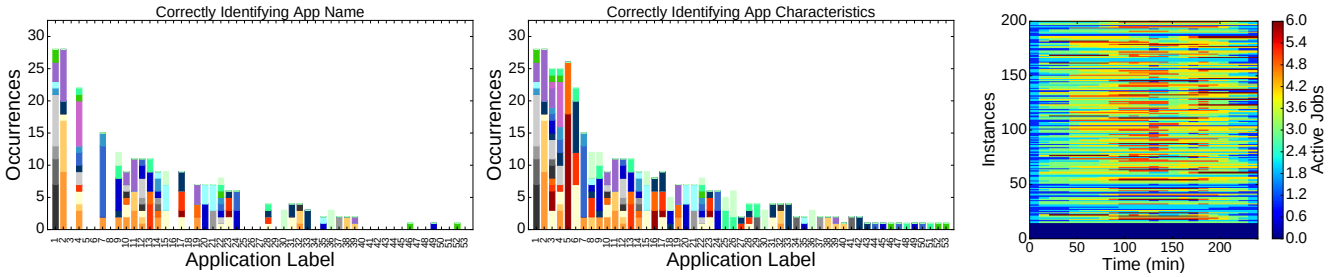


Figure 12: Detection accuracy of (a) application labels and (b) resource characteristics. Fig 12(c): number of jobs per instance.

figure 11 groups applications by framework the individual logic and datasets differ between jobs in the same framework. Note that we have not updated the training set for this study; it consists of the same 120 applications described in Section 3.4.

Figure 12a shows the number of applications that were correctly identified by name across the different categories. In total 277 jobs are correctly labeled. As expected, Bolt could not assign a label to application types it has not seen before, such as email clients, or image editing applications. However, even when Bolt cannot label an application, it can still identify the resources it is sensitive to, as shown in figure 12b. For 385 out of 436 jobs, Bolt correctly identifies the job's resource characteristics. For performance attacks, such as the ones described in Section 5, this information is sufficient to dramatically degrade the performance of a victim application. Finally, figure 12c shows the number of co-scheduled applications per instance throughout the duration of the experiment. The bottom 14 instances remained unused; for the remaining 186 instances the number of active applications ranges from 1 to 6, with each job taking one or more vCPUs. The majority of misclassified applications correspond to instances with 5 or more concurrently active jobs.

5. Security Attacks

Bolt makes several cloud attacks practical and difficult to detect. We discuss three possible attacks that leverage the information obtained through detection.

5.1 Internal Denial of Service Attack

Attack setting: Denial of service attacks hurt the performance of a victim service by overloading its resources [14, 2, 51]. In cloud settings specifically, they can be categorized in two types; *external* and *internal* (or host-based) attacks. External attacks are the most conventional form of DoS [51, 13, 22]. These attacks utilize external servers to direct excessive traffic to the victims, flooding their resources, and hurting their availability. External DoS attacks affect mostly PaaS and SaaS systems, and include IP spoofing, synchronization (SYN) flooding, smurf, buffer overflow, land, and teardrop attacks.

In contrast, internal DoS attacks take advantage of IaaS and PaaS cloud multi-tenancy to launch adversarial programs on the same host as the victim and degrade its performance [14, 60, 34, 52]. For example, Ristenpart et al. [60] showed how an adversarial user can leverage the IP naming conventions of IaaS clouds to locate a victim VM and degrade its performance. Cloud providers are starting to build defenses against such attacks. For example, EC2 offers autoscaling that will increase the instances of a service under heavy resource usage. Similarly, there is related work on mechanisms that detect saturation in memory bandwidth

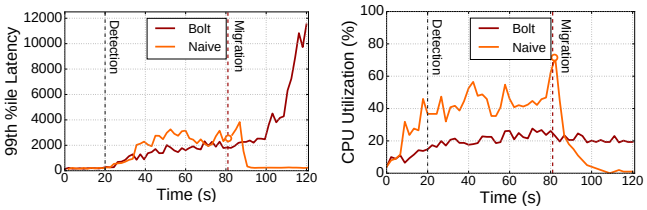


Figure 13: Latency and utilization with Bolt and a naïve DoS attack that saturates CPU resources.

and can trigger VM migration [71]. This means that DoS attacks that simply overload a physical host are ineffective when such defenses are in place. We focus on host-based DoS attacks and make them resilient against such defenses by avoiding resource saturation. We leverage the information obtained by Bolt to construct custom interference programs that stress the resources the victim is most sensitive to without overloading the host.

Bolt creates custom contentious workloads by combining the microbenchmarks used to measure the resource pressure of the victim. Since these microbenchmarks are tunable, Bolt configures their intensity to a higher point than their measured pressure c_i during detection. For example, if a victim is detected as memcached, and its most critical resources are the L1-i cache (81% pressure) and the last level cache (LLC) (78% pressure), Bolt uses the two microbenchmarks for L1-i and LLC at a higher intensity than what memcached can tolerate.

Impact: We use the DoS attack setting above against the 108 applications of the controlled experiment (Section 3.4). The DoS attack results in degraded performance by 2.2x on average and up to 9.8x in terms of execution time. Degradation is much more pronounced for interactive workloads like key-value stores, with tail latency increasing by 8-140x, a dramatic impact for applications with strict tail latency SLAs.

We now examine the impact of the DoS attack on utilization. If it translates to resource saturation, there is a high probability that the DoS will be detected and the victim migrated to a new machine. The experimental cluster supports live migration. If CPU utilization (sampled every 1 sec) exceeds 70% the victim is migrated to an unloaded host. Figure 13 compares the tail latency and CPU utilization with Bolt to that of a naïve DoS that saturates the CPU through a compute-intensive kernel. We focus on a single victim VM running memcached. The overhead of migration (time between initiating migration and latency returning to normal levels) for this VM is 8sec. Performance degradation is similar for both systems until t=80sec, at which point the victim is migrated to a new host due to the compute-intensive kernel causing utilization to exceed 70%. While during migration performance continues to degrade, once the VM resumes in the new server, latency returns to nominal levels. In contrast, Bolt keeps utilization low, and impacts the victim's performance beyond t=80sec.

5.2 Resource Freeing Attack

Attack setting: The goal of a resource freeing attack (RFA) is to modify the workload of a victim such that it frees up resources for the adversary, improving its performance [67, 66]. The adversarial VM consists of two components, a *beneficiary* and a *helper*. The beneficiary is the program whose performance the attacker wants to improve, and the helper is the program that forces the victim to yield its resources. The RFA works by adding load in the victim's critical resources, causing other resources to free up. For example, if the victim is a memory-bound Spark job, introducing additional memory traffic will result in Spark stalling in memory, and lessening its pressure in the other resources, until it can reclaim its required memory bandwidth. When launched in a public cloud, the victim of an RFA ends up paying more and achieving worse performance compared to running in isolation.

We now create a proof of concept RFA for a webserver, a network-bound Hadoop job, and a memory-bound Spark job. Launching RFAs requires in depth knowledge of the victim's resource requirements [67], which Bolt provides by identifying the victim's dominant resource. Once the dominant resource is detected, the runtime applies a custom helper program that saturates this critical resource. For the webserver, the helper is a CPU-intensive benchmark launching CGI requests, causing the victim to saturate its CPU usage, servicing fewer real user requests, and freeing up cache resources. For the Hadoop job, we use a network-intensive benchmark similar to `iperf`, which saturates network bandwidth, and frees up CPU and memory resources for the beneficiary. Similarly, for Spark we use a streaming memory benchmark that slows down k-means and frees network and compute resources for the adversary. The selection of the beneficiary is of lesser importance; without loss of generality we select `mcf`, a CPU-intensive benchmark from SPECCPU2006. The same methodology can be used with other beneficiaries, conditioned to their critical resource not overlapping with the victim's.

Impact: Table 2 shows the performance degradation for the three victim applications, and the improvement in execution time for the beneficiary. The webserver suffers the most in terms of queries per second, as the helper's CGI requests pollute its cache, preventing it from servicing legitimate user requests. Hadoop and Spark experience significant degradations in execution time, due to network and memory bandwidth saturation respectively. Execution time for `mcf` improves by 16-38%, benefiting from the victim stalling in its critical resource to improve its cache and CPU usage.

5.3 VM Co-residency Detection

Attack setting: Sections 3.4 and 4 showed that we can identify the applications sharing a cloud infrastructure. However, a malicious user is rarely interested in a random service running on a public cloud. More often, they target a specific

Victim		Beneficiary		Target
App	Perf	App	Perf	Resource
Apache Webserver	-64% (QPS)	`mcf`	+24%	CPU
Hadoop (*SVM*)	-36% (Exec.)	`mcf`	+16%	Network BW
Spark (*k-means*)	-52% (Exec.)	`mcf`	+38%	Memory BW

Table 2: RFA impact on the victims and beneficiaries.

workload, e.g., a competitive e-commerce site. Therefore they need to pinpoint where the target resides in a practical manner. This requires a *launch strategy* and a mechanism for *co-residency detection* [69]. The attack is practical if the target is located with high confidence, in reasonable time and with modest resource costs. Bolt enables this attack to be carried out in a practical way and remains resilient against defense mechanisms, such as virtual private clouds (VPCs) which make internal IP addresses private to a single tenant. Once a target VM is located, the adversary can launch RFA or DoS attacks as previously described. Co-residency detection attacks rely on leakage of logical information, e.g., IP address, or on observing the impact of a side-channel attack due to resource contention. Bolt relies on a variation of the latter approach.

Assume a system of N servers. A target victim user launches k VMs. The adversary launches n malicious VMs. Instances are launched simultaneously to avoid malicious-malicious instance co-residency [69]. The probability that at least one adversarial VM will be co-scheduled with a victim instance is: $P(f) = 1-(1-k/N)^n$. The adversary then uses a two-process detection scheme. The *sender* is a process that creates contention to degrade the victim's performance. The *receiver* is the process that detects this degradation, running either on the same host as the victim (*cooperative detection*) or externally (*uncooperative detection*). We assume an uncooperative victim, which is the more general case. Bolt works as the sender to determine the type of co-residents on each sampled host and identify any of the desired type. Assume m VMs of the desired type have been detected in the sample set. Bolt now needs to prune down this space to VM(s) from the specific target victim. It launches an external receiver aiming at the victim over a public channel, e.g., HTTP, or key-value protocol, to ping the victim service. This is the only point of communication with the victim. At the same time the sender incurs contention in the resources the victim is most sensitive to. If the sender and victim are co-residents the receiver's requests will be slower due to contention. Bolt quickly prunes down the sampled VMs by determining the type of co-residents. In a large cluster this reduces the time to pinpoint a victim, and the resource costs for the adversary. It also increases the confidence in the detection, in the presence of noise from other co-residents.

Impact: We evaluate co-residency detection on the same 40-node cluster. The victim is a SQL server, instantiating a single VM. The cluster also hosts 7 other VMs running SQL. Other active VMs host key-value stores, Hadoop and Spark jobs. Bolt launches 10 senders simultaneously on randomly-selected machines ($P(f) = 0.66$), and detects 3 VMs running SQL in the sample set. It then introduces memory interference, as the receiver initiates a large number of SQL queries. While these queries have 8.16msec mean latency without interference, latency now becomes 26.14msec. Given a $\sim 3\times$ latency increase, we can conclude that one of the co-residents is the victim SQL server. Detection required 6sec from instantiation to receiver detection, and 11 adversary VMs.

6. Improving Security via Resource Isolation

Since interference is at the heart of Bolt's detection methodology, isolation mechanisms that attenuate contention should reduce its accuracy. We first evaluate to what extent existing isolation techniques mitigate security vulnerabilities, and then highlight trade-offs between security, performance, and utilization. We use the same setting as in the controlled experiment of Section 3.4. We evaluate 4 resource-specific isolation techniques and 3 settings for OS-level isolation mechanisms: *baremetal*, *containerized*, and *virtualized*. The baremetal system does not employ any OS level isolation. The containerized setup uses Linux containers (`lxc`), and assigns cores to applications via `cpuset cgroups`. Both containers and VMs constrain memory capacity. Baremetal experiments do not enforce memory capacity allocations, and the Linux scheduler is free to float applications across cores.

The first resource-specific isolation techniques is *thread pinning* to physical cores, to constrain interference from scheduling actions, like context switching. The number of cores an application is allocated can change dynamically, and is limited by how fast Linux can migrate tasks, typically in the tens of milliseconds.

For *network isolation*, we use the outbound network bandwidth partitioning capabilities of Linux's traffic control. Specifically, we use the `qdisc` [8] scheduler with hierarchical token bucket queueing discipline (HTB) to enforce egress bandwidth limits. The limits are set to the maximum traffic burst rate for each application to avoid contention (`ceil` parameter in HTB interface). Ingress network bandwidth isolation has been extensively studied in previous work [36]; these approaches can be applied here as well.

For *DRAM bandwidth isolation*, there is no commercially available partitioning mechanism. To enforce bandwidth isolation we use the following approach: we monitor the DRAM bandwidth usage of each application through performance counters [45] and modify the scheduler to only colocate jobs on machines that can accommodate their aggregate peak memory bandwidth requirements. This requires extensive application knowledge, and is used simply to highlight the benefits of DRAM bandwidth isolation.

Finally, for *last level cache (LLC) isolation*, we use the Cache Allocation Technology (CAT) available in recent Intel chips [11]. CAT partitions the LLC in ways, which in

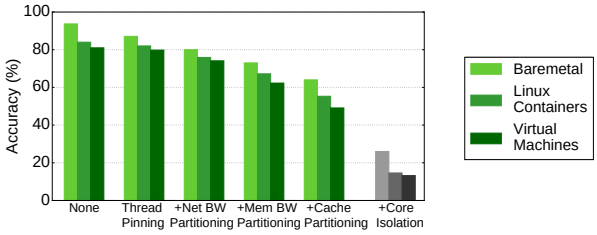


Figure 14: Detection accuracy with isolation techniques.

a highly-associative cache allows for non-overlapping partitions at the granularity of a few percent of the LLC capacity. Each co-resident is allocated one partition configured to its current capacity requirements [57]. Partitions can be resized at runtime by reprogramming specific low-level registers (MSRs); changes take effect after a few milliseconds.

We add one isolation mechanism at a time in the three system settings (baremetal, containers, VMs). Figure 14 shows the impact of isolation techniques on Bolt's detection accuracy. As expected, when no isolation is used, baremetal allows for a significantly higher detection accuracy than container- and VM-based systems, mostly due to the latter constraining core and memory capacity usage. As a result, introducing thread pinning benefits baremetal the most, since it reduces core contention. It also benefits container- and VM-based setups to a lesser degree, by eliminating unpredictability introduced by the Linux scheduler (e.g., context switching) [40]. The dominant resource of each application determines which isolation technique benefits it the most. Thread pinning mostly benefits workloads bound by on-chip resources, such as L1/L2 caches and cores. Adding network bandwidth partitioning lowers detection accuracy for all three settings almost equally. It primarily benefits network-bound workloads, for which network interference conveys the most information for detection. Memory bandwidth isolation further reduces accuracy by 10% on average, benefiting jobs dominated by DRAM traffic. Finally, cache partitioning has the most dramatic reduction in accuracy, especially for LLC-bound workloads. We attribute this to the importance cache pressure has as a detection signal. The number of co-residents also affects the extent to which isolation helps. The more co-scheduled applications exist per machine, the more isolation techniques degrade accuracy, as they make distinguishing between co-residents harder.

Unfortunately, even when all the techniques are on, accuracy is still 50% due to two reasons: current techniques are not fine-grain enough to allow strict and scalable isolation and core resources (L1/L2 caches, CPU) are prone to interference due to contending hyperthreads [61]. To evaluate the latter hypothesis, we modify the scheduler, such that hyperthreads of different jobs are never scheduled on the same physical core, e.g., if a job needs 7vCPUs it is allocated 4 dedicated physical cores. The grey bars of Figure 14 show the detection accuracy. Baremetal instances still allow certain applications to be detected, but for containerized and virtualized settings, accuracy drops to 14%. The remaining accuracy corresponds to disk-heavy workloads. Improving security, however, comes at a performance penalty of 34% on average in execution time, as threads of the same job are forced to contend with one another. Alternatively, users can overprovision their resource reservations to avoid degradation, which results in a 45% drop in utilization. This means that the cloud provider cannot leverage CPU idleness to share machines, decreasing the cost benefits of cloud computing. Note that enforcing core isolation alone is also not sufficient, as it allows a detection accuracy of 46%.

Discussion: The previous analysis highlights a design problem with current datacenter platforms. Traditional multicores are prone to contention, which will only worsen as more cores are integrated in each server, and multi-tenancy becomes more pronounced. Existing isolation techniques are insufficient to mitigate security vulnerabilities, and techniques that provide reasonable security guarantees sacrifice performance or cost efficiency, through low utilization. This highlights the need for new *fine-grain*, and *coordinated* isolation techniques that guarantee security at high utilization for shared resources.

7. Conclusions

We have presented Bolt, a practical system that uses online data mining techniques to identify the type and characteristics of applications running on shared cloud systems, and enables practical attacks that degrade their performance. In a 40-server cluster, Bolt correctly identifies 87% out of 108 workloads, and degrades tail latency by up to 140x. We also used Bolt in a user study on EC2 to detect the characteristics of unknown jobs submitted by multiple users. Finally, we show that, while existing isolation techniques are helpful, they are not sufficient to mitigate such attacks. Bolt reveals real, easy-to-exploit threats in public clouds. We hope that this work will motivate cloud providers and computer scientists to develop and deploy stricter isolation primitives in cloud systems.

Acknowledgements

We sincerely thank all the participants in the user study for their time and effort. We also thank Daniel Sanchez, Ed Suh, Grant Ayers, Mingyu Gao, Ana Klimovic, and the rest of the MAST group, as well as the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported by the Stanford Platform Lab, NSF grant CNS-1422088, and a John and Norma Balen Sesquicentennial Faculty Fellowship.

References

[1] Amazon ec2. http://aws.amazon.com/ec2/.

[2] Aman Bakshi and Yogesh B. Dujodwala. Securing cloud from ddos attacks using intrusion detection system in virtual machine. In *Proc. of the Second International Conference on Communication Software and Networks (ICCSN)*. 2010.

[3] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[4] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Proc. of the International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. Busan, South Korea, 2014.

[5] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proc. of the 24th ACM International Conference on Supercomputing (ICS)*. Tsukuba, Japan, 2010.

[6] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*. Paris, France, 2010.

[7] Eric Brewer. Kubernetes: The path to cloud native. http://goo.gl/QgkzYB, SOCC Keynote, August 2015.

[8] Martin A. Brown. Traffic control howto. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[9] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, November 2002.

[10] Apache cassandra. http://cassandra.apache.org/.

[11] Intel ®64 and IA-32 Architecture Software Developer's Manual, vol3B: System Programming Guide, Part 2, September 2014.

[12] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, September 2007.

[13] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium*. Washington, DC, 2003.

[14] Marwan Darwish, Abdelkader Ouda, and Luiz Fernando Capretz. Cloud-based ddos attacks and defenses. In *Proc. of i-Society*. Toronto, ON, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013.

[16] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[17] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *ACM Transactions on Computer Systems (TOCS), Vol. 31 Issue 4*. December 2013.

[18] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014.

[19] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[20] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[21] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.

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

[23] 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 *Proc. of the ACM Symposium on Cloud Computing (SOCC)*. San Jose, CA, 2012.

[24] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*. Brasov, Romania, 2007.

[25] Alexander Felfernig and Robin Burke. Constraint-based recommender systems: Technologies and research issues. In *Proceedings of the ACM International Conference on Electronic Commerce (ICEC)*. Innsbruck, Austria, 2008.

[26] Brad Fitzpatrick. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*.

[27] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[28] Google container engine. https://cloud.google.com/container-engine.

[29] Asela Gunawardana and Christopher Meek. A unified approach to building hybrid recommender systems. In *Proc. of the Third ACM Conference on Recommender Systems (RecSys)*. New York, NY, 2009.

[30] Sanchika Gupta and Padam Kumar. Vm profile based optimized network attack pattern detection scheme for ddos attacks in cloud. In *Proc. of SSCC*. Mysore, India, 2013.

[31] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher Leckie. Security games for virtual machine allocation in cloud computing. In *4th International Conference on Decision and Game Theory for Security*. Fort Worth, TX, 2013.

[32] Amir Herzberg, Haya Shulman, Johanna Ullrich, and Edgar Weippl. Cloudoscopy: Services discovery and topology mapping. In *Proceedings of the ACM Workshop on Cloud Computing Security Workshop (CCSW)*. Berlin, Germany, 2013.

[33] Ben Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*. Boston, MA, 2011.

[34] Jingwei Huang, David M. Nicol, and Roy H. Campbell. Denial-of-service threat to hadoop/yarn clusters with multi-tenancy. In *Proc. of the IEEE International Congress on Big Data*. Washington, DC, 2014.

[35] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *Proceedings of CCGRID*. Newport Beach, CA, 2011.

[36] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, 2013.

[37] Yaakoub El Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *Proceedings of CloudCom*. Indianapolis, IN, 2010.

[38] Krzysztof C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. In *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001.*

[39] Ruby B. Lee. Rethinking computers for cybersecurity. *IEEE Computer*, 48(4):16–25, 2015.

[40] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of EuroSys*. Amsterdam, The Netherlands, 2014.

[41] Host server cpu utilization in amazon ec2 cloud. http://goo.gl/2LTx4T.

[42] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*. San Jose, CA, 2015.

[43] Fei Liu, Lanfang Ren, and Hongtao Bai. Mitigating cross-vm side channel attack on multiple tenants cloud platform. In *Journal of Computers, Vol 9, No 4 (2014), 1005-1013, April 2014*.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Minneapolis, MN, 2014.

[45] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[46] Mahout. http://mahout.apache.org/.

[47] Dave Mangot. Ec2 variability: The numbers revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_re%vealed.

[48] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of ISCA*. Tel-Aviv, Israel, 2013.

[49] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2012.

[50] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011.

[51] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review (CCR)*, April 2004.

[52] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium (SS)*. Boston, MA, 2007.

[53] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of EuroSys*. Paris,France, 2010.

[54] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Lecture Notes on Cloud Computing*. Volume 34, p.115-131, 2010.

[55] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007.

[56] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing, SCC@ASIACCS*. Hangzhou, China, 2013.

[57] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

[58] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proc. of the ACM Workshop on Cloud Computing Security (CCSW)*. Chicago, IL, 2009.

[59] Suhail Rehman and Majd Sakr. Initial findings for provisioning variation in cloud computing. In *Proceedings of CloudCom*. Indianapolis, IN, 2010.

[60] 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 *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL, 2009.

[61] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the 38th annual International Symposium in Computer Architecture (ISCA-38)*. San Jose, CA, June, 2011.

[62] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings VLDB Endow.*, 3(1-2):460–471, September 2010.

[63] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*. Prague, Czech Republic, 2013.

[64] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *Proc. of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. Boston, MA, 2010.

[65] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, 2012.

[66] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the cpu without superuser privileges. In *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium*. Boston, MA, 2007.

[67] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proc. of the Conference on Computer and Communications Security (CCS)*. Raleigh, 2012.

[68] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *Proc. of the 23rd Usenix Security Symposium*. San Diego, CA, 2014.

[69] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *Proc. of the 24th USENIX Security Symposium (USENIX Security)*. Washington, DC, 2015.

[70] Huaibin Wang, Haiyun Zhou, and Chundong Wang. Virtual machine-based intrusion detection system framework in cloud computing environment. In *Journal of Computers, October 2012*.

[71] Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu. A-drm: Architecture-aware distributed resource management of virtualized clusters. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE)*. Istanbul, Turkey, 2015.

[72] Ian H. Witten, Eibe Frank, and Geoffrey Holmes. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[73] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proc. of the 21st USENIX Conference on Security Symposium (USENIX Security)*. Bellevue, WA, 2012.

[74] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proc. of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW)*. Chicago, IL, 2011.

[75] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *Proc. of the 24th USENIX Security Symposium (USENIX Security)*. Washington, DC, 2015.

[76] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *Proc. of the 23rd Usenix Security Symposium*. San Diego, CA, 2014.

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI*. San Jose, CA, 2012.

[78] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proc. of the IEEE Symposium on Security and Privacy*. Oakland, CA, 2011.

[79] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Scottsdale, AZ, 2014.

[80] 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 ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC, 2012.

[81] Yinqian Zhang and Michael K. Reiter. Duppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. Berlin, Germany, 2013.

[82] Fangfei Zhou, Manish Goel, Peter Desnoyers, and Ravi Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. *J. Comput. Secur.*, 21(4):533–559, July 2013.

[83] Jieming Zhu, Pinjia He, Zibin Zheng, and Michael R. Lyu. Towards online, accurate, and scalable qos prediction for runtime service adaptation. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. Madrid, Spain, 2014.

[84] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of the Fifteenth Edition of on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Pittsburgh, PA, 2010.