A Side-channel Attack on HotSpot Heap Management

Xiaofeng Wu, Kun Suo, Yong Zhao, and Jia Rao

The University of Texas at Arlington
{xiaofeng.wu, kun.suo, yong.zhao, jia.rao}@uta.edu

Abstract

CPU time-multiplexing is a common practice in multi-tenant systems to improve system utilization. However, the sharing of CPU and a single system clock makes it difficult for programs to accurately measure the length of an operation. Since a program is not always running in a time-sharing system but the system clock always advances, time perceived by one program could be dilated as it may include the run time of another program. Applications employing time-based resource management face a potential security threat of time manipulation.

HotSpot, a widely used Java virtual machine (JVM), relies on timing garbage collections to infer an appropriate heap size. In this paper, we present a new active side-channel attack that exploits time dilation to break the heap sizing algorithm in parallel scavenge, the default garbage collector in JDK 8. We demonstrate that a deliberate attack targeting a specific type of GC is able to crash a Java program with out-of-memory errors, cause excessive garbage collection, and leads to significant memory waste due to a bloated heap.

1 Introduction

For over two decades, Java has retained its popularity and been widely adopted to build various computer systems, including Big Data systems [2, 23], machine learning frameworks [4], search engines [3, 5], and NoSQL databases [1]. Among the many nice features, such as cross-platform portability, Java's automatic memory management releases users from the burden of explicit memory allocation and deallocation. The built-in garbage collector (GC) in the Java runtime environment, i.e., the Java Virtual Machine (JVM), automatically reclaims heap memory for reuse when memory allocation fails due to insufficient heap space.

Modern JVMs, such as the HotSpot JVM [16], devise sophisticated heap management schemes to improve memory efficiency and guarantee quality-of-service (QoS) as well as avoiding out-of-memory (OOM) errors. The JVM grows or shrinks the heap size according to the memory demand of the Java application. For example, the HotSpot JVM uses command line options `-Xms` and `-Xmx` to specify the initial and the maximum heap sizes of a Java application at its launch time. During run time, the JVM adjusts the heap size based on the statistics of garbage collection. In general, the heap is shrunk if each individual GC takes too long and violates a user-defined pause-time target; the heap is expanded if GC is frequently performed and the total GC time constitutes a significant portion of the total application execution time, i.e., violating the throughput target; if both targets are met, the JVM gradually shrinks the heap to save memory.

The heap sizing policy is critical to the user-perceived QoS and memory efficiency. However, it is vulnerable to a deliberate side-channel attack in a multi-tenant system, where resources are often over-subscribed and shared among users. In our previous work [21], we found that time measurement in a time-sharing system can be inaccurate. Since there is only one system clock shared among multiple users, the time (i.e., the length of program execution) measured by one program using the difference of two consecutive calls of `gettimeofday` may include the period in which another program is running. Although this issue has been found to cause premature TCP timeouts [10] and erroneous program behaviors on mobile devices [14], it is believed that the effect of inaccurate timing is random and universally distributed to all programs, thereby unclear how it affects program correctness or performance.

In this paper, we demonstrate that a deliberate attacker can exploit the timing channel to break the heap management of a Java program. Most Java heap management schemes use measured GC time, which is based on wall-clock time, to determine an appropriate heap size. The attack interferes with GC timing to deceive the heap sizing algorithm such that the JVM mistakenly configures an insufficient or excessive heap. We empirically

validate that even the most sophisticated sizing policy in HotSpot, i.e., the parallel scavenge (PS) collector, is not immune to the side-channel attack. We design micro-benchmarks to exercise the PS collector and develop a proof-of-concept attack by directly tampering the source code of the PS collector. Results show that attacking the pause time target causes a benchmark to spend as much as 60% more time in GC; attacking the throughput target leads to a bloated JVM which uses up to four times more memory. Furthermore, we are able to create an attack that consistently causes a benchmark, which never fails when not attacked, to crash due to OOM errors. Finally, we demonstrate the feasibility of launching a realistic attack on heap management. We leverage eBPF to trace JVM execution and deliberately affect GC timing by slowing down GC threads. All attacks except the one crashes the JVM can be reproduced on real Java programs from the *SPECjvm2008* [7] and *DaCapo* [9] benchmarks.

Our findings raise a question: *Are all programs relying on time-based resource management vulnerable to such an attack in a multi-tenant environment?* The ultimate countermeasure to the timing side-channel attack is to use virtual time, which only advances when a program is running, in resource management.

2 Background

In this section, we describe the adaptive heap sizing algorithm in the parallel scavenge (PS) collector and analyze its vulnerabilities to the side-channel attack. Then, we explain how time measurement can be inaccurate in a time-sharing system.

2.1 Adaptive Heap Sizing in PS

Parallel scavenge is a throughput-oriented collector and it pauses application threads, i.e., mutators, during GC. This GC period is called a stop-the-world (STW) pause. PS employs multiple GC threads to concurrently scan the heap and frees objects with unreachable references. It monitors the length of each GC and the mutators run time before being interrupted by GC. Based on GC and mutator time, PS dynamically adjusts the JVM heap size to meet two goals: 1) pause time – the STW pause time should not exceed a user-defined upper bound; 2) throughput – the portion of mutator time in the total program execution time (i.e., mutator time + GC time) should not be less than a desired ratio (the default target is 99%). If both goals are met, PS shrinks the heap to save memory.

PS divides the heap space into multiple generations: *young*, *old*, and *permanent*. The young generation is further divided into one eden space and two survivor spaces, i.e., from-space and to-space. New objects are always first allocated into the eden space. When the eden space is filled up, a minor GC is performed. Referenced objects in eden and from-space are moved to the to-space, and unreferenced objects are discarded. After a minor GC, the eden and the from-space are cleared, and objects survived in the to space have their age incremented. After surviving a predefined number of minor GCs, objects are promoted to the old generation. Similarly, as the old generation is filled up, a major GC is triggered to free space in the old generation. The adaptive sizing algorithm adjusts the sizes of the young generation and the old generation separately based on the measurement of minor GC and major GC, respectively.

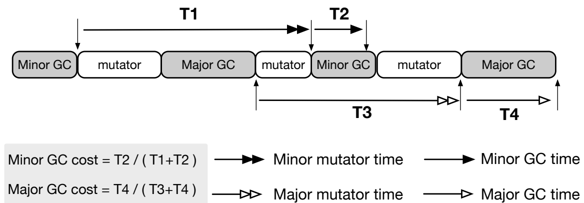

Figure 1: **The calculation of GC costs in PS.**

Out-of-Memory failure A memory allocation failure occurs if the JVM heap does not have enough free space to accommodate new objects. With parallel scavenge's generational garbage collection, the HotSpot JVM performs five allocation attempts before throwing an OOM error and terminating the program: 1) PS first performs a minor GC to clear the young generation, where the new objects are being allocated; 2) PS performs a major GC to clear the old generation and frees space in the young generation by promoting mature objects to the old generation. PS still tries to allocate objects in the young generation; 3) if both attempts fail, the JVM tries to allocate the new objects directly to the old generation; 4) if this attempt fails, PS performs a more aggressive major GC by clearing objects with soft references and tries to allocate the objects in the young generation; 5) the last attempt tries to allocate objects directly to the old generation. After each failed GC, PS invokes the adaptive sizing algorithm to expand the heap. As will be shown, an attack on the sizing algorithm could impede the heap expansion and thus cause OOM errors.

GC cost is the key metric used in the adaptive sizing algorithm to determine an appropriate heap size. Figure 1 shows the calculation of the minor GC cost and the major GC cost. GC time is the length of a single GC of a particular type; mutator time is the length of the period between two adjacent GC of a particular type. For example, T_1 and T_3 refer to the minor mutator time and major mutator time, respectively, and they are the intervals between two adjacent minor and major GCs. Note that the mutator time may also include the time spent in the other type of GC. The major mutator time T_3 includes a minor GC time T_2. GC cost is defined as the ratio of the time

spent in the most recent GC to the period since the last time the same type of GC occurred. GC cost is a robust metric to measure the overhead of different types of GC even they interleave with each other and the mutators. This helps the JVM deal with the timing issue as inaccurate time measurement is likely to affect both minor and major GC costs. However, a deliberate attacker focusing on manipulating time measurement on a particular GC type can easily deceive the heap sizing algorithm. To avoid abrupt changes in the heap size, PS calculates the GC costs based on the moving average of a few recent GCs. All GC and mutator time measurements are based on wall-clock time, i.e., using `gettimeofday`.

2.1.1 Adjusting young generation

The most important adjustment to the young generation is to change the size of the eden space, where most new objects are allocated.

Pause time-oriented adjustment shrinks the eden space if the minor GC time exceeds a user-defined threshold. PS iteratively reduces the eden size at a fixed rate until reaching the minimum eden size.

Throughput-oriented adjustment expands the eden space if the minor mutator time falls below (by default) 99% of the total execution time, i.e., $\frac{T_1}{T_1+T_2} < 99\%$ as shown in Figure 1, to reduce the frequency and thus the overhead of minor GC. PS increases the eden size at an adaptive rate:

$$E_i = E_{i-1} \times \left(1 + \text{scale}_{minor} \times \text{INC_STEP}\right), \quad (1)$$

where E_i is the desired eden size in the next round and E_{i-1} is current eden size. scale_{minor} controls how fast the eden space is expanded and is defined as:

$$\text{scale}_{minor} = \frac{C_{minor}}{C_{minor} + C_{major}}, \quad (2)$$

where C_{minor} and C_{major} are the minor and major GC costs, respectively. PS expands the eden space at a faster rate if the minor GC cost contributes to a greater portion of the total GC cost.

Survivor space adjustment is based on the moving average of historical survivor sizes. In addition, PS sets the survivor space to its maximum size if the young generation is full. Therefore, the survivor spaces reach their maximum size upon a young generation overflow, then gradually fall back until stabilizing at a smaller size.

Footprint adjustment gradually reduces the eden size if both the pause time and throughput goals are met. The size decrement is at an adaptive rate controlled by $\text{scale}_{footprint_minor} = \frac{E}{E+P}$, where E is the eden size and P is the desired promoted size in the old generation, which will be determined by the major GC (see Section 2.1.2).

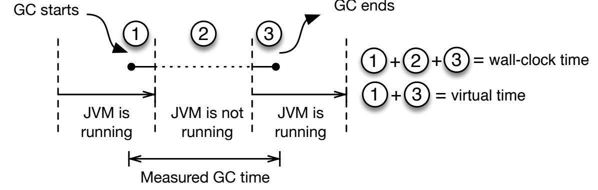

Figure 2: **Dilation of GC time due to CPU multiplexing.**

2.1.2 Adjusting old generation

Old generation adjustment aims to provide sufficient headroom for objects allocation while preserving memory as much as possible. It calculates a target size of headroom, i.e., free space, and adjusts the current free space in the old generation accordingly. The target free space is calculated as the sum of two parts: 1) the size of objects recently promoted from the young generation to the old generation and 2) the desired promoted size. The promoted size (part 1) guarantees a lower bound on the free space and the desired promoted size (part 2, denoted as P) is determined based on the statistics of GCs.

The sizing of the desired promoted size P is similar to that in the young generation sizing algorithm, which goes through the pause time- and throughput-oriented, and footprint adjustments. The sizing process is iterative and based on the desired promoted size in the previous round. Different from the young generation adjustment, the throughput-oriented expansion is controlled by $\text{scale}_{major} = \frac{C_{major}}{C_{minor}+C_{major}}$ and footprint adjustment shrinks P at a rate of $\text{scale}_{footprint_major} = \frac{P}{E+P}$

An important difference in old generation sizing is that the violation of pause time target of a major GC also triggers the shrinking of the eden space.

2.2 Time Dilation in Multi-tenant Systems

Time stamp counter (TSC) is a commonly used clock source in modern operating systems for timekeeping. It is an auto-incremented register on CPU at the clock rate. Timing utilities, such as `gettimeofday`, read TSC values to track time passage. Figure 2 shows how the measurement of GC time can be dilated in a multi-tenant system. Due to CPU time-sharing, the GC time measurement based on wall-clock time can include other program's run time. As a result, the GC appears to be much longer than it actually was. This opens up opportunities for an attacker to manipulate the heap sizing algorithm.

3 A Side-channel Attack

In this section, we exploit GC time dilation to attack the heap sizing algorithm in PS. The attacks aim to 1) crash a Java program by causing OOM errors, 2) degrade performance by incurring excessive GCs, and 3) waste memory by causing a heap bloat. All experiments were conducted on a 64-core machine using OpenJDK 1.8 and Linux

	Baseline	Attacked	Degradation
# minor GC	1187	1971	66.05%
# major GC	30	49	63.33%
# total GC	1217	2020	65.98%
GC CPU time (sec)	146.59	240.03	63.74%

Table 1: **Performance degradation of *h2* caused by an attack on pause time-based heap sizing.**

4.15.0. The JVM was configured with 15 GC threads.

3.1 A Proof-of-Concept Attack

We begin with a proof-of-concept attack, in which we modify the code of the sizing algorithm to emulate an attack. We enlarged the actual measurements of GC time in the sizing algorithm and used the modified GC time to calculate the GC costs. This allows us to tamper the timing of any GC of our choice.

First, we demonstrate how to crash a Java program and cause OOM errors. We created a micro-benchmark that continuously inserts small fixed-size (50-byte) objects into an array. This allows the heap sizing algorithm to slowly ramp up the heap. Finally, the benchmark inserts a much larger object (20 MB), which inevitably causes an allocation failure. The sizing algorithm is expected to expand the heap to resolve the failure, when we launch the attack. We dilated the major GC time to violate the pause time target. Recall that this will cause both the old and young generations to be shrunk. As the attack persisted, the sizing algorithm failed to expand the heap and thus threw an OOM error. We confirmed that the benchmark never crashed and had sufficient memory without the attack.

Second, we attack to degrade performance. Specifically, we created an attack on the pause time-oriented sizing. Similarly, we dilated the major GC time to violate a user-defined pause time target of 100*ms*. The program under attack was *h2* from the *Dacapo* benchmarks [9]. *h2* is a transaction-based benchmark and has a strict response time requirement. As shown in Table 1, the attack deceived the JVM to shrink the heap and thus incurred excessive GCs. Not only the number of minor and major GCs was increased, but also did the total GC time. Intuitively, a small heap size triggers more frequent GCs but each GC takes less time. If the total number of objects to clear remains unchanged, the GC time should remain the same. The results show that the attack incurred much wasteful work, which was mostly due to the repeated scan of live objects. The overall degradation on GC performance was around 65%.

Third, we attack throughput-oriented heap sizing and footprint adjustment. We created a micro-benchmark with fluctuating memory demand. It inserts objects into an array and later removes all the objects. As shown

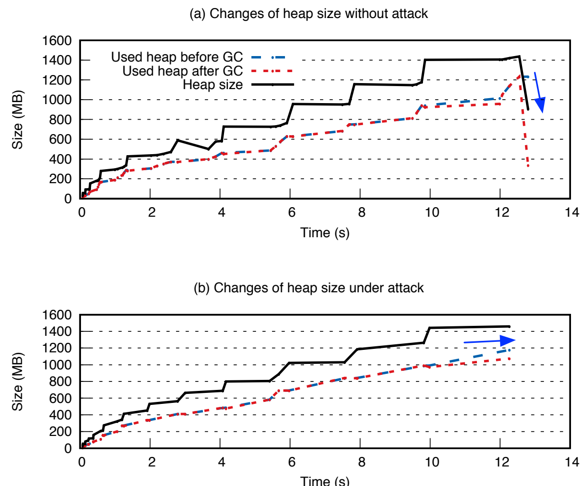

Figure 3: **Heap size of a micro-benchmark before and after attack.**

in Figure 3 (a), without the attack, the sizing algorithm effectively expanded and shrank the heap, matching the memory demand. Figure 3 also plots the used heap before and after each GC. At around the 13th second and after the objects were removed from the array, GC cleared unreferenced objects and allowed the sizing algorithm to free the unused space. In contrast, as shown in Figure 3 (b), under an attack, the sizing algorithm failed to shrink the heap after the memory demand dropped. The attack targeted only minor GCs and aimed at violating the throughput goal. The effect is that, under the attack, the sizing algorithm kept expanding the heap than it needed to and never shrank the heap. As a result, the micro-benchmark used about 61% more memory under attack compared to that without the attack.

4 A Realistic Attack

This section validates the feasibility of the side-channel attacks on Java heap management in real systems. According to our experiments with the proof-of-concept attacks, we found that these attacks are only effective and able to break the sizing algorithm when particular types of GCs are targeted. The pause-time and OOM attacks need to temper the timing of the major GC while the throughput and footprint attack targets the minor GC. To this end, we leverage the extended Berkeley Packet Filter (eBPF) [6], an in-kernel virtual machine, to trace the execution of GCs and launch a targeted attack. We assume that the attacker has gained the root privilege of the machine. We monitored the `libjvm.so` library of the JVM and dilated GC time by slowing down the threads that perform GC. Specifically, we traced `GCTaskThread::run` to obtain the PID of a GC thread and attack this thread whenever it starts a GC. For example, we used symbol

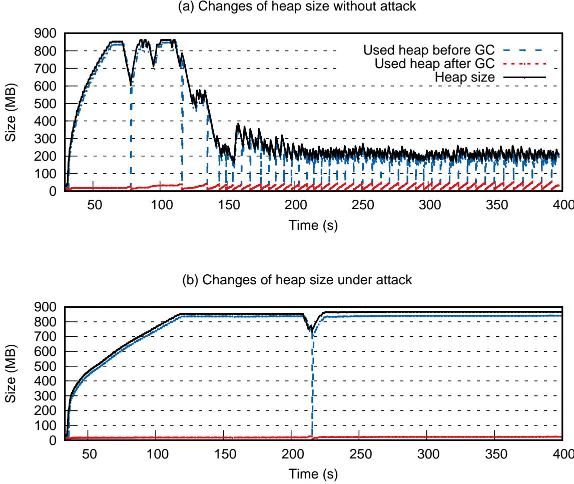

Figure 4: **Heap size of mpegaudio before and after attack.**

`_ZN18AdaptiveSizePolicy22minor_collection_beginEv` to identify the start of a minor GC. We slow down GC threads using Linux `cgroups` to limit the amount of CPU the JVM is able to use during GC. The CPU limit was lifted when a GC completed.

We were able to reproduce the pause-time attack and the throughput attack. We had a similar result when attacking the pause time target of *h2* from *Dacapo*. Under attack, *h2* spent 88% more time on GC. The throughput attack even caused higher damage to a realistic benchmark, *mpegaudio* from SPECjvm2008 [7]. As shown in Figure 4 (a) and (b), *mpegaudio* used up to four times of memory under attack. From Figure 4 (b), we can see that there is a large gap between heap usage before and after GC, indicating that the GC was effective. However, the sizing algorithm was deceived by the dilated minor GC time and unable to shrink the heap, leaving a large amount of memory wasted. Unfortunately, we were unable to reproduce the crash attack on *Dacapo* or *SPECjvm2008* benchmarks because they do not have sudden spikes in memory demand as we did in our microbenchmark. Nevertheless, we believe that the crash attack is still possible for some realistic workloads, which poses a great security threat.

5 Discussion

Vulnerable Java programs In general, all Java programs could be vulnerable to the side-channel attack since HotSpot uses wall-clock time to manage the heap. However, we found that Java programs with fluctuating memory demands are most vulnerable to the throughput attack because the sizing algorithm is frequently triggered to expand or shrink the heap, allowing the attack to deceive the algorithm. On the other hand, programs with stable heap usage are most vulnerable to the pause time attack, which forces the JVM to reduce the heap size and incurs more GCs. Finally, programs with high heap utilizations are more prone to crash under attack.

Countermeasures A straightforward defence against the side-channel attack is to fix the heap size at program launch, which disables the sizing algorithm. However, it is difficult to predict how much memory a program needs. Under-provisioning leads to OOM errors while over-provisioning wastes precious memory. An ultimate countermeasure is to use virtual time, which only advances when the JVM is running, in the sizing algorithm. While it is easy to maintain virtual time on a per thread basis, there lacks a definition of virtual time for a JVM with group of threads, which are often inter-dependent.

6 Related Work

Side-channel Attack Most cache-based side-channel attacks [18, 8] rely on prime-and-probe [17] and Flush-Reload techniques [12] to exploit timing information of memory accesses so that it can infer secrets from victim applications. Xu et al., [22] took advantage of an untrusted operating system [19] to construct page-fault channel, and Marcus et al., [13] transcended temporal and spatial limitations of the page-fault channel to launch high resolution attack by utilizing the single stepping features of the hardware. Ristenpart et al., [20] showed that multi-tenancy in public clouds makes cross-VM side-channel attacks feasible. Yinqian et al. [24] further demonstrated a cross-tenant side-channel attack on commercial Platform-as-a-Service (PaaS) clouds. In this paper, we discovered a new type of *active* side-channel attack on the timing of victims.

Time manipulation has been used in computer systems to accelerate simulation and expose performance bottlenecks. Timekeeper [15] made time in a VM advances slower or faster than wall-clock time to artificially scale simulations which interact with external devices. Curtsinger et al., [11] proposed COZ, a causal profiling tool that uses virtual speedups to pinpoint the performance bottlenecks. COZ virtually speeds up a code segment in order to quantify the impact of a potential optimization by slowing down all other threads. Abhilash et al., [14] discovered that sleep-induced time bugs (SITB), a similar timing issues as discussed in this work, can cause erroneous behaviors on Android devices. In this work, we interfere with the timing of GC to launch three types of attacks on Java heap management.

7 Conclusion

We present a new active side-channel attack on Java heap management. The attack exploits potential time dilation in multi-tenant systems to break the heap sizing algorithm in the parallel scavenge collector. We demonstrate the feasibility to the attack to crash a Java program, cause excessive garbage collection and memory waste.

References

[1] *The Apache Cassandra project: open-source distributed NoSQL database management system*. http://cassandra.apache.org/.

[2] *The Apache Hadoop project: open-source software for reliable, scalable, distributed computing*. http://hadoop.apache.org/.

[3] *The Apache Lucene project: open-source information retrieval software library*. https://lucene.apache.org/.

[4] *The Apache Mahout project: open-source software for distributed, scalable machine learning frameworks*. https://mahout.apache.org/.

[5] *Elasticsearch: open-source, distributed, RESTful search engine*. https://www.elastic.co/.

[6] *The extended Berkeley Packet Filter*. https://lwn.net/Articles/740157/.

[7] *The Standard Performance Evaluation Corporation (SPEC) JVM 2008 benchmark suite*. https://www.spec.org/.

[8] BERNSTEIN, D. J. Cache-timing attacks on AES.

[9] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., ET AL. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (2006).

[10] CHENG, L., WANG, C., AND LAU, F. C. M. PVTCP: towards practical and effective congestion control in virtualized datacenters. In *Proceedings of the 21st IEEE International Conference on Network Protocols, ICNP* (2013).

[11] CURTSINGER, C., AND BERGER, E. D. Coz: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015).

[12] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011).

[13] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA* (2017).

[14] JINDAL, A., HU, Y. C., MIDKIFF, S. P., AND JOSHI, P. Unsafe time handling in smartphones. In *USENIX Annual Technical Conference* (2016).

[15] LAMPS, J., NICOL, D. M., AND CAESAR, M. Timekeeper: A Lightweight Virtual Time System for Linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2014).

[16] LINDHOLM, T., YELLIN, F., BRACHA, G., AND BUCKLEY, A. The Java Virtual Machine Specification-Java SE 8 Edition, March 2014.

[17] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers Track at the RSA Conference* (2006).

[18] PERCIVAL, C. Cache Missing for Fun and Profit, 2005.

[19] PORTS, D. R., AND GARFINKEL, T. Towards Application Security on Untrusted Operating Systems. In *HotSec* (2008).

[20] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. 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* (2009).

[21] SUO, K., ZHAO, Y., RAO, J., CHENG, L., ZHOU, X., AND LAU, F. Preserving I/O prioritization in virtualized OSes. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017).

[22] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[23] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets.

[24] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).