A Side-channel Attack on HotSpot Heap Management

Xiaofeng Wu, Kun Suo, Yong Zhao, Jia Rao

The University of Texas at Arlington

UNIVERSITY OF TEXAS ARLINGTON

HotCloud'18

July 9, 2018

Side-Channel Attack

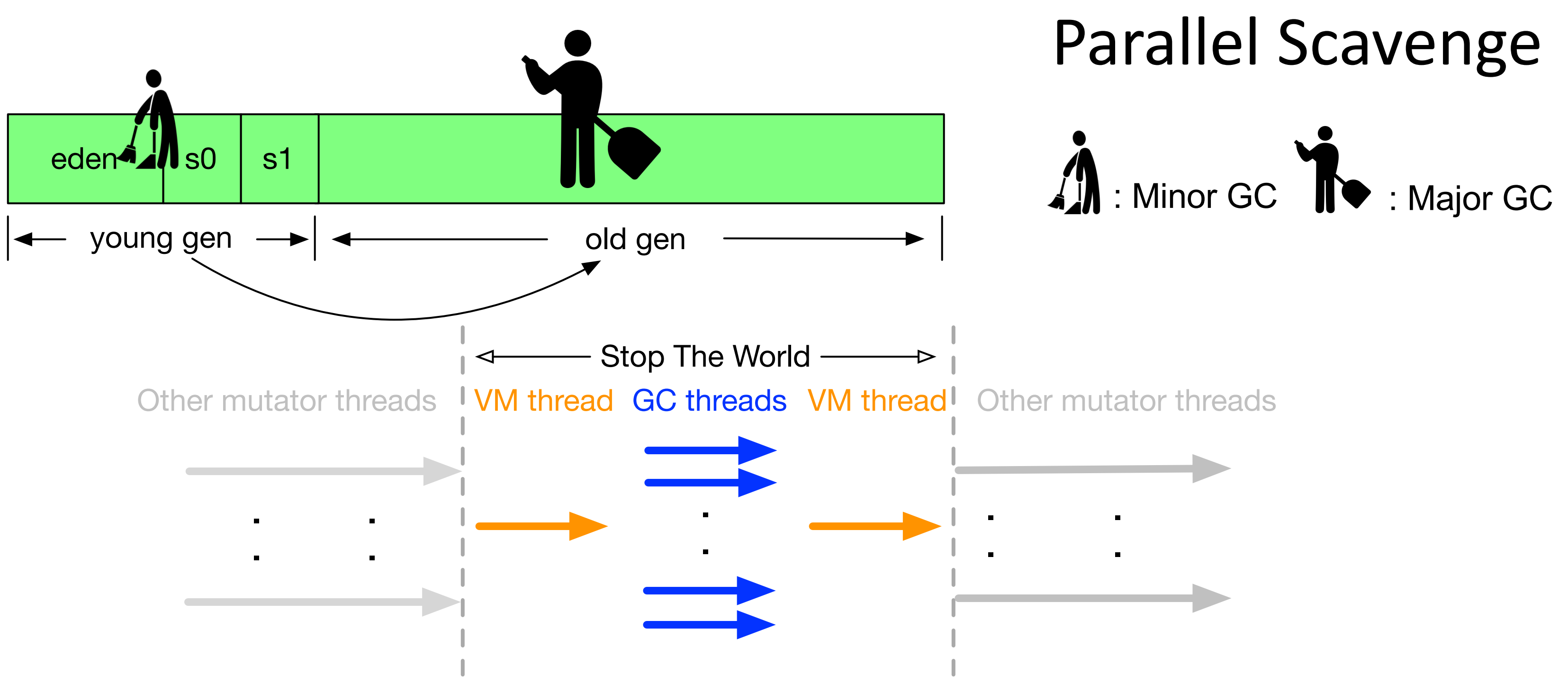
- Attack based on information gained from the implementation of a computer system
  - Shared cache
  - Timing
  - Power consumption
  - Acoustic measurement

Steal or infer secrets

Infer user activities to launch well-timed attack

Attack **shared clock** in multi-tenant systems to manipulate users' **time measurement**

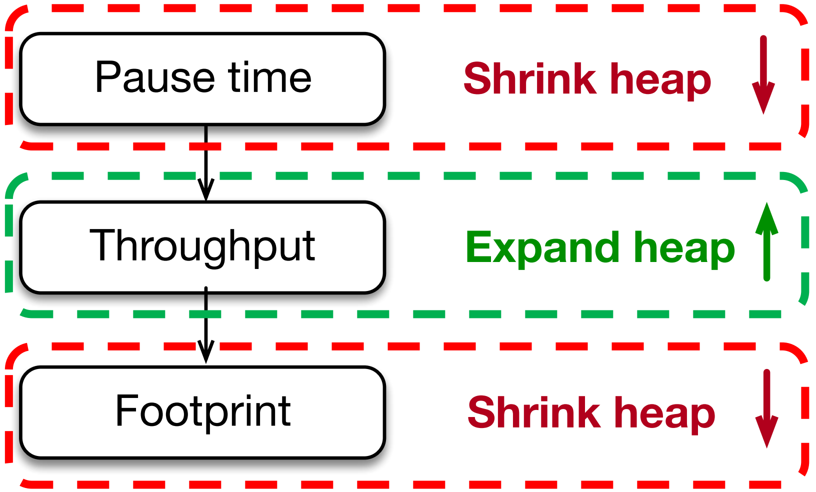
Garbage Collection in HotSpot JVM



- Each individual GC shouldn't take too long – large heap
- Total time spent in GC shouldn't be too much – small heap, too frequent GC

Adaptive Heap Sizing in PS GC

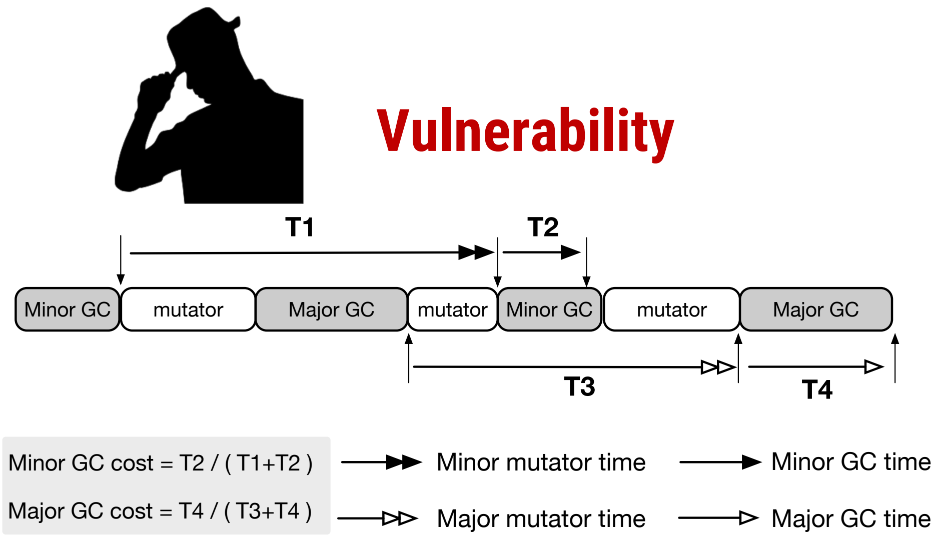
- Three objectives
  - Meet pause time target
  - Meet throughput goal
  - Minimize memory footprint

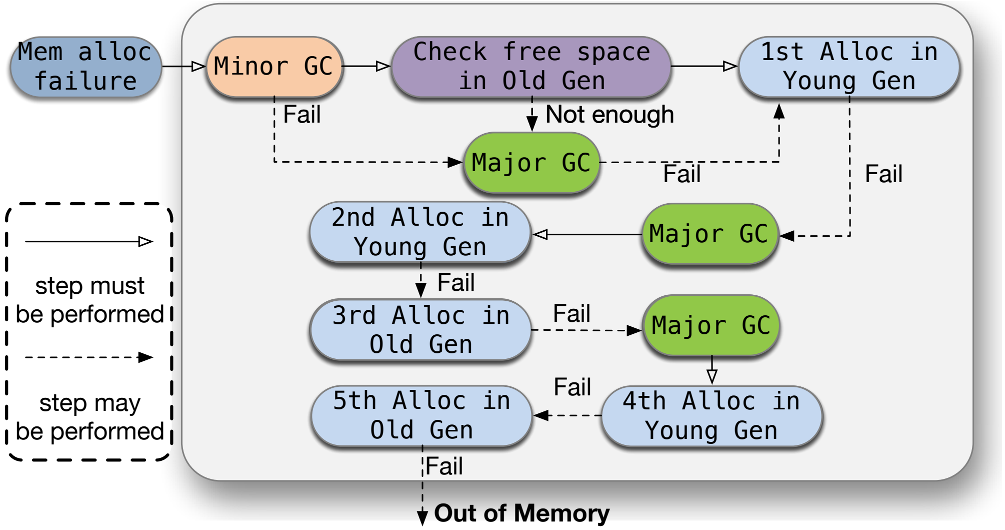


JVM automatically determines the heap size in the range of the initial (-Xms) and the maximum (-Xmx) heap sizes

Time is used as an **indirect measure** for **memory efficiency**

Minor and Major GC

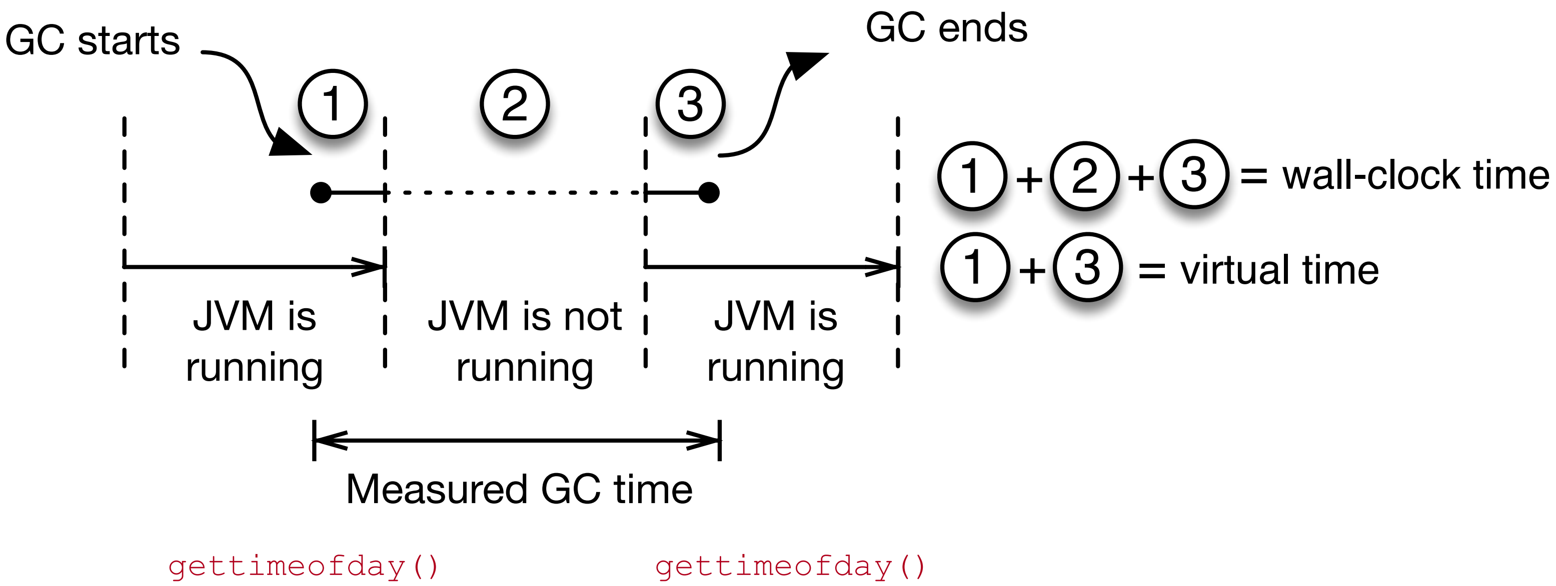




JVM infers heap efficiency based on measured lengths of minor and major GCs, and adjusts heap size accordingly

JVM throws an out-of-memory (OOM) error if five GCs fail to resolve the memory allocation failure

Shared Clock



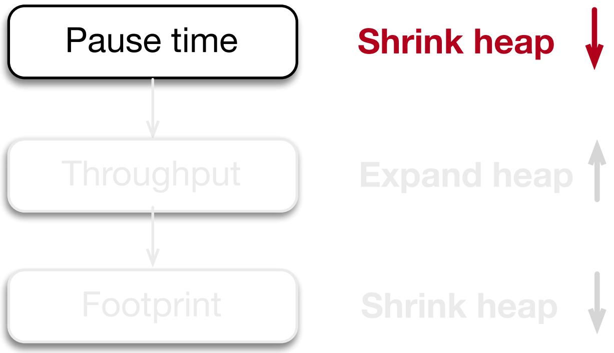
Time measurement can be inaccurate in the presence of CPU multiplexing

Three Types of Attacks

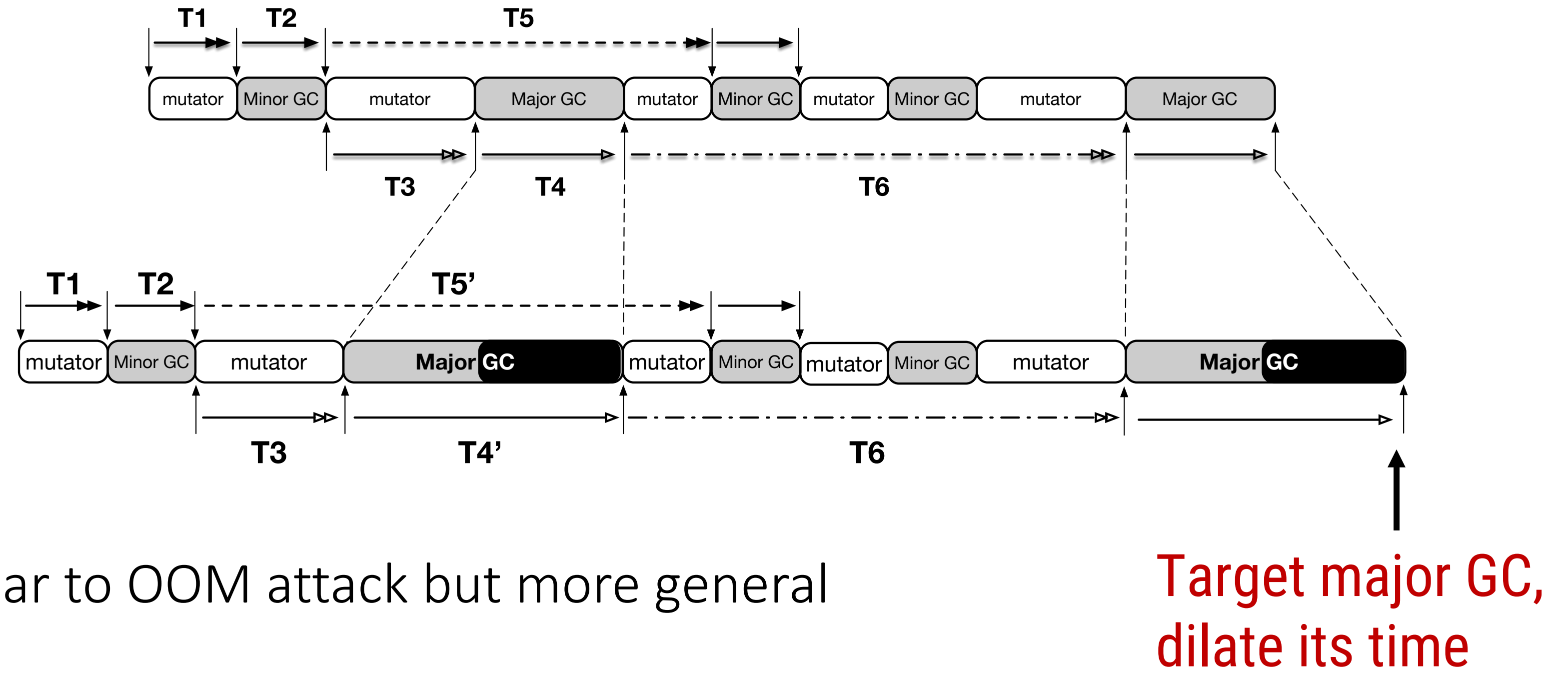
- Cause OOM errors
  - Prevent JVM from expanding the heap in 5 GCs
- Cause excessive GC
- Cause bloated heap

OOM Attack

- Attack pause time target
  - When there is a spike in memory demand and allocation failure, attack major GC measurement
  - Dilated major GC time cause the heap to shrink, missing the opportunity to avoid OOM errors

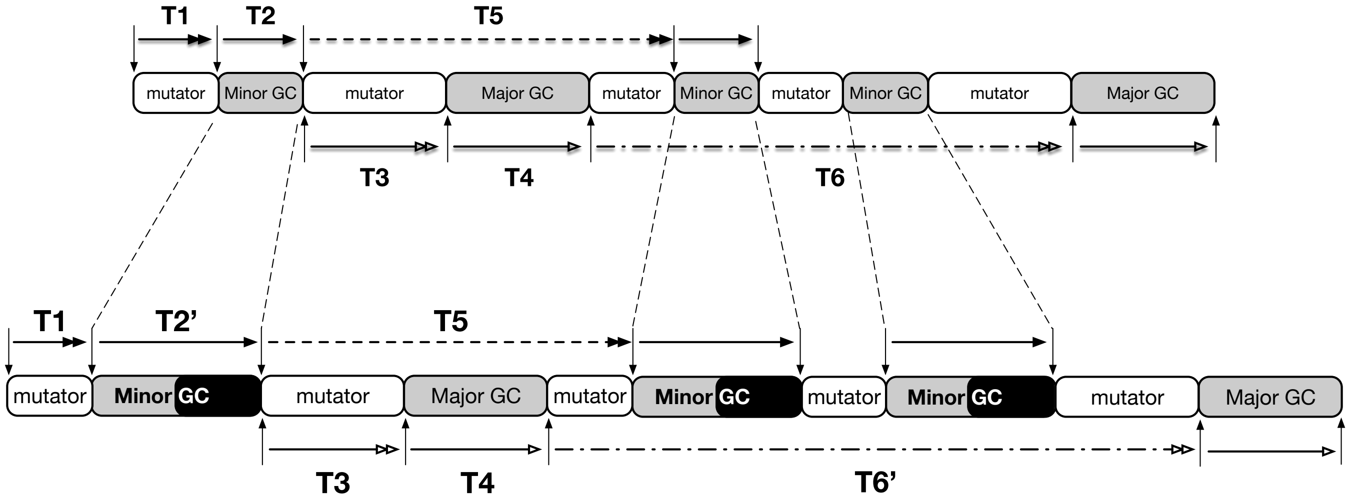


Excessive GC Attack

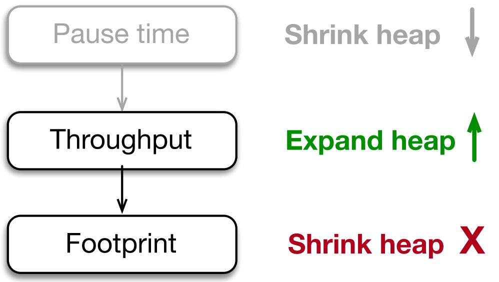


- Similar to OOM attack but more general
- Old generation have a tendency to drop quickly, and the decrement of heap size results in more GCs

Memory Bloat Attack



Violate throughput target



$$Throughput = \frac{T1}{T1+T2} \longrightarrow Throughput' = \frac{T1}{T1+T2'}$$

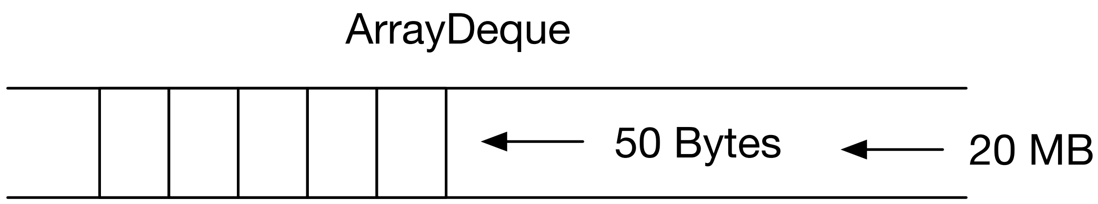
Attack minor GC to prevent the heap from shrinking even memory demand drops

Launch Attacks

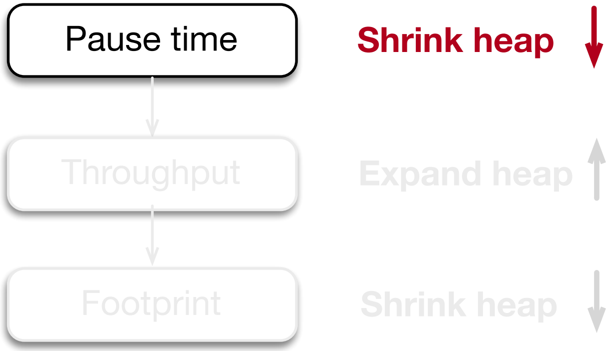
- Proof-of-concept attacks
  - Modify JVM source code to manipulate GC time in the adaptive sizing algorithm
- Realistic attacks
  - Use eBPF to monitor `libjvm.so` to obtain GC thread ID and slowdown a specific type of GC
  - Use cgroup to limit the CPU usage of GC threads and hence dilate GC time
- Results
  - Crash a Java-based micro-benchmark with OOM errors
  - Cause ~65% more GC time in DaCapo
  - Inflict up to ~400% memory bloat in SPECjvm2008

OOM Attack

- Attack major GC measurement
- JAVA_OPTION=
  - -XX:+UseAdaptiveSizePolicy
  - -XX:+UseParallelGC
  - -XX:+UseParallelOldGC
  - -XX:ParallelGCThreads=10
  - -Xms = 32m -Xmx = 2g
- Both proof-of-concept and realistic attacks crash the micro-benchmark



A micro-benchmark with a sudden spike in memory demand



Discussion

- Essence of the problem
  - Heap size should be determined by the characteristics of a Java program
  - But heap efficiency is measured by GC time, an indirect measure
  - External CPU contention can affect internal heap management
- Many programs designed for dedicated systems are vulnerable to similar attacks in multi-tenant systems
  - CPU multiplexing → wall-clock time or virtual time?
  - VMs, containers, conventional processes
  - Linux jiffies and userspace gettimeofday track wall-clock time
  - Linux CFS uses steal_clock to track virtual time for thread scheduling

See our [Suo-SoCC17] paper for another issue caused by time discontinuity

Is this a real problem?

- No
  - No evidence that many applications suffer from inaccurate time measurement.
  - Even so, the effect is random and universally distributed among applications.
  - Our attack is sophisticated and needs to target a specific type of GC, not easy.

- Yes
  - In theory, if not measuring absolute latency, time measurement that is only relevant to a particular program or to measure the relative progress of program threads, should use virtual time
  - This could be the source of erroneous program behavior, unpredictability and inefficiency

Should we devise a completely isolated virtual time interface for individual programs/VMs/containers ?

Thank you!

Questions?

xiaofeng.wu@mavs.uta.edu

Backup Slides …

A Realistic Attack

- All experiments were conducted on a 64-core machine using OpenJDK 1.8 and Linux 4.15.0.
- The JVM was configured with 10 GC threads.
- Benchmark
  - Dacapo: h2
  - SPECjvm2008: mpegaudio

Pause time-oriented Attack (excessive GC)

- A realistic attack using eBPF
- Benchmark: *h2* from Dacapo
- The initial and maximum heap sizes: 16 MB and 900 MB
- The maximum pause time is set to 100 ms

	Baseline	Attacked	Overhead
# minor GC	1223	2033	66.23%
# major GC	28	46	64.29%
# total GC	1251	2079	66.19%
GC CPU time(sec)	132.93	250.03	88.09%

The attack shrinks the heap, causing 88% more GC time

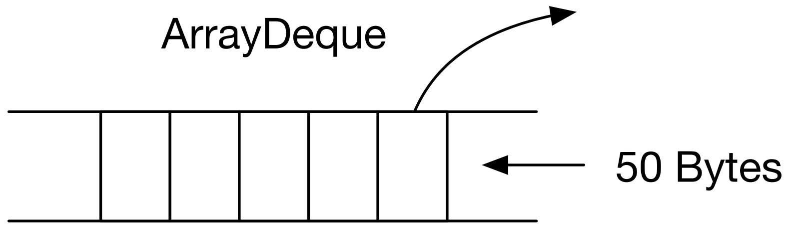
Cont'd - Pause time-oriented

- We choose *h2* from *Dacapo-9.12-MR1-bach* as a case study
  - execute a number of transactions
  - set the maximum pause time as 100 ms

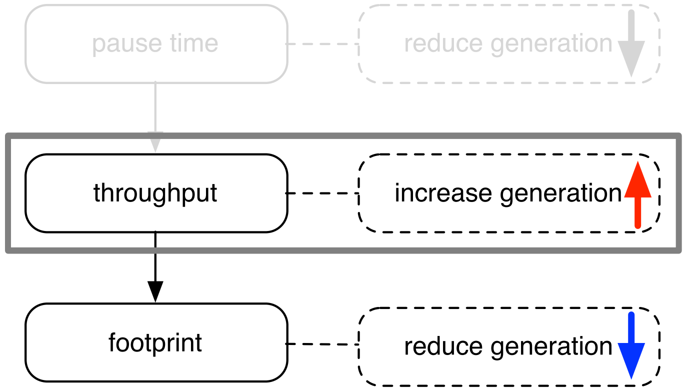
	Baseline	Attacked	Overhead
# minor GC	1187	1971	66.05%
# major GC	30	49	63.33%
# total GC	1217	2020	65.98%
GC CPU time(sec)	146.59	240.03	63.74%

The overhead induced by the pause time-oriented attack to the micro-benchmark.

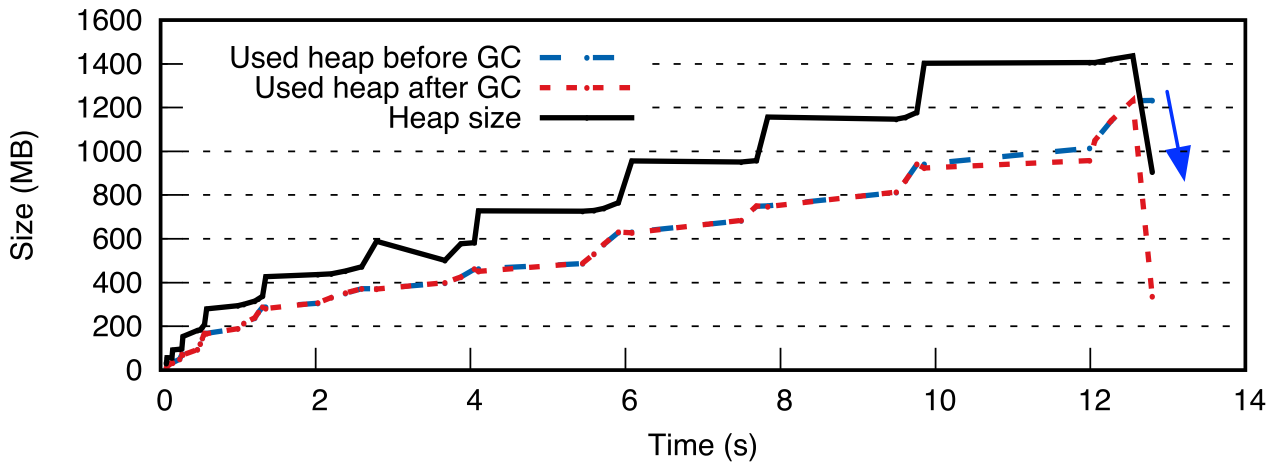
Cont'd - Throughput-oriented



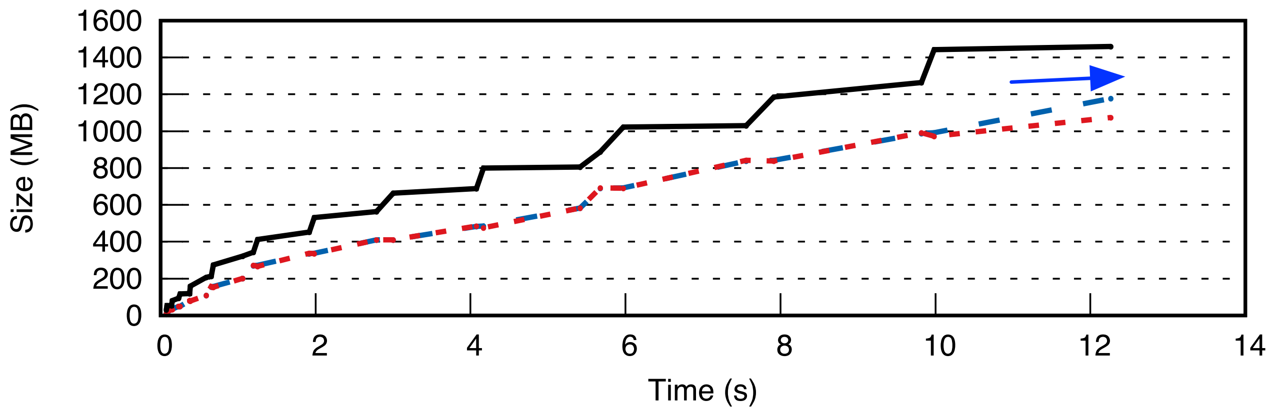
- -Xms32m -Xmx32g
- Heap size is 1.61× larger



(a) Changes of heap size without attack

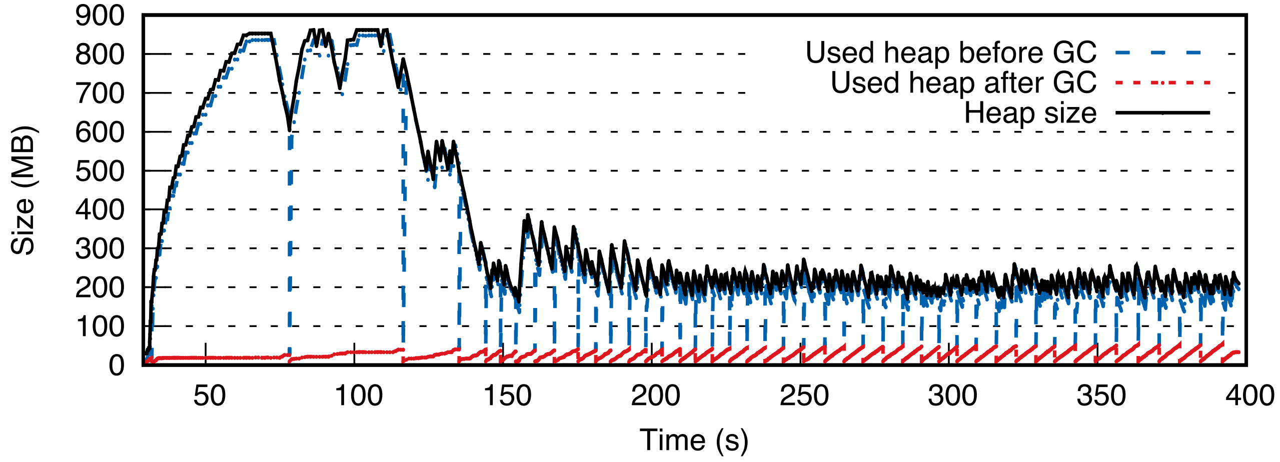


(b) Changes of heap size under attack

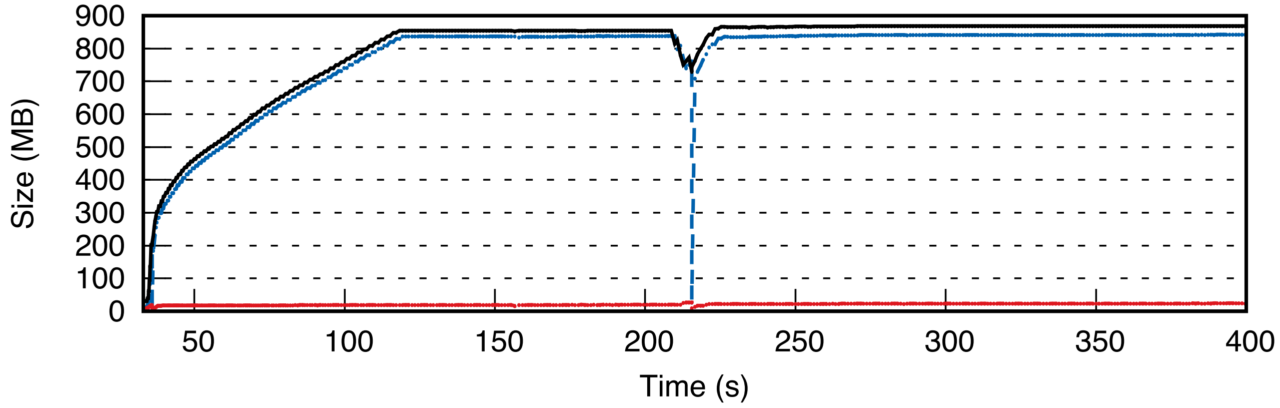


Throughput-oriented Attack (memory bloat)

w/o attack



under attack



- A realistic attack using eBPF
- *mpegaudio* from SPECjvm2008
- The initial and maximum heap sizes: 32 MB and 2.5GB

The attack prevents the heap from shrinking when memory demand drops, causing more than 400% waste of memory