RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure

Chang Lou[1], Cong Chen[2], Peng Huang[1], Yingnong Dang[2], Si Qin[3], Xinsheng Yang[4], Xukun Li[2], Qingwei Lin[3], Murali Chintalapati[2]

[1]Johns Hopkins University [2]Microsoft Azure [3]Microsoft Research [4]Meta

Abstract

Memory leak is a notorious issue. Despite the extensive efforts, addressing memory leaks in large production cloud systems remains challenging. Existing solutions incur high overhead and/or suffer from high inaccuracies.

This paper presents RESIN, a solution designed to holistically address memory leaks in production cloud infrastructure. RESIN takes a divide-and-conquer approach to tackle the challenges. It performs a low-overhead detection first with a robust bucketization-based pivot scheme to identify suspicious leaking entities. It then takes live heap snapshots at appropriate time points in carefully sampled leak entities. RESIN analyzes the collected snapshots for leak diagnosis. Finally, RESIN automatically mitigates detected leaks.

RESIN has been running in production in Microsoft Azure for 3 years. It reports on average 24 leak tickets each month with high accuracy and low overhead, and provides effective diagnosis reports. Its results translate into a 41× reduction of VM reboots caused by low memory.

1 Introduction

Memory leak is a prevalent issue in software, from applications [13] to OS kernels and device drivers [46]. At Microsoft Azure, its infrastructure contains many complex software components running on a massive number of machines with various workloads. Unsurprisingly, these components encounter memory leak issues from time to time. When a process leaks memory, the direct consequence is performance degradation and crash. Worse still, its impact often affects other components running on the same machine, such as causing excessive paging, innocent processes being killed, and node reboots.

Memory leak is notoriously difficult to deal with, especially in a production cloud infrastructure setting. The issues are usually only triggered by rare conditions and occur slowly, thus they easily escape testing and failure detectors [20]. After leak symptoms are detected, it is time-consuming and sometimes impossible to reproduce them offline. Unlike other failures like crashes that have clear points to start diagnosis, developers are often clueless in finding the leak's root cause.

Extensive solutions have been proposed to detect memory leak bugs. One approach uses static analysis techniques [10, 15, 18, 36, 47] to analyze the software source code and deduce potential leaks. The second approach detects memory leaks dynamically by instrumenting a program and tracking the object references at runtime [16, 21, 25, 39, 49].

While helpful, these solutions are insufficient to address the memory leak challenges in Azure. Static approach is limited by the well-known accuracy and scalability issues with static analyses. It also only focuses on leaks in which allocated objects are unreachable [24]. If memory objects are reachable but never accessed again, it still incurs the consequences of leaks. Such leaks are hard to detect statically. Moreover, memory leaks in cloud infrastructure can be caused by cross-component contract violations, which require too much domain knowledge to recognize statically.

Dynamic approaches better fit Azure's requirements. However, while the existing dynamic detection solutions are generally more accurate, they are intrusive and require extensive instrumentations that are cumbersome to apply to complex components [16, 21]. They also incur high runtime overhead that is prohibitive for deployment in production [6].

In this paper, we present RESIN, an end-to-end service designed to holistically address memory leaks in large cloud infrastructure, from detection to diagnosis and mitigation. RESIN is highly scalable—it analyzes all the host software components, including kernels, drivers, and system processes, on millions of nodes in Azure. RESIN has low overhead while running in production environment. At the same time, RESIN provides good accuracy and helps developers pinpoint the root causes of memory leak issues.

Two key insights motivate the design of RESIN and enable it to achieve the above properties. First, the conundrum of existing solutions is in part because they mix detecting a leak and pinpointing the leak bug in one step, so they have to make trade-offs among accuracy, scalability, and overhead. In our experience, we should decompose the detection and pinpointing into multi-level stages to catch memory leaks at production scale. Second, taking a centralized service approach that

leverages low-level system mechanism is essential to support many components transparently in a non-intrusive way. It also enables gathering valuable information from many nodes in the cloud to address the accuracy challenges.

Based on these insights, RESIN performs non-intrusive, low-overhead leak detection first. When a process is suspected of experiencing leaks, RESIN triggers a live heap-snapshot mechanism to capture sufficient evidence and runs diagnosis. RESIN leverages kernel-level monitors and profilers as its building blocks, so it directly supports all the running processes without cumbersome integration. Furthermore, RESIN builds a centralized service that analyzes processes across all hosts in Azure fleet together to capture complex leaks.

A key challenge for dynamic leak detection is the highly noisy nature of memory usage in modern software affected by the workload characteristics. Using simple static thresholds can easily generate many false alarms or false negatives. For instance, in an impactful real-world cloud service outage caused by memory leaks, no alarm was triggered despite the existence of a memory monitoring service [4].

RESIN addresses this challenge by designing a robust *bucketization-based pivot* scheme. It aggregates the memory usages of processes across machines, and groups them into different buckets. Then by performing a pivot analysis on the process name, bucket, and other attributes, RESIN can reliably detect leaks without being prone to fragile thresholds. Essentially, we focus on analyzing a component's global memory usage behavior, rather than the microscope of an individual process. The rationale is that a true memory leak comes down to some buggy release. Although the memory usage of an individual process is highly dependent on workloads, the workload effect is likely canceled out when inspecting the usage of the same component running in all machines.

Once a suspicious memory leak is detected, RESIN activates the second stage of taking *live* heap snapshots of the suspected processes, which contain information about the active allocations and their stack traces. This stage is more heavyweight but provides more evidence to help developers confirm and diagnose the issue. Since a leak is often sporadic, RESIN aims to "hit" the leak again and capture useful evidence. It carefully chooses the snapshot time points so that the obtained snapshots have a high chance of localizing the root causes while minimizing the snapshot cost. Besides taking heap snapshots of the suspected leaking process, RESIN performs a *fingerprinting step* that periodically takes heap snapshots of representative processes to build a reference database. This reference database is used in the diagnosis algorithm to further improve the diagnosis accuracy.

Finally, RESIN automatically mitigates a detected leak to minimize its impact on the service availability and performance. The mitigation engine in RESIN leverages the information from the detection and diagnosis engines, and determines the appropriate actions to resolve the leak symptoms while developers investigate the root causes and fixes.

RESIN has been running in production in Azure for more than 3 years. RESIN reported many memory leaks, helped developers diagnose the issues, and automatically mitigated the leaks before their impact becomes visible to customers. Within the recent year at the timing of writing, the unexpected VM reboots in Azure caused by out of memory are reduced by 41×, and the new VM allocation errors due to low memory are reduced by 10×. In addition, no severe outages in 2020 and 2021 at Azure were caused by memory leaks.

In summary, the main contributions of this work are:

- A holistic memory leak solution for cloud infrastructure.
- A novel bucketization-based pivot scheme to robustly detect memory leaks with low overhead.
- A live heap snapshot algorithm to effectively capture evidence in production and diagnose memory leaks.
- A lightweight automated leak mitigation design.
- Deployment of RESIN in a production cloud service.

2 Background and Motivation

2.1 Host Memory Compositions

In IaaS cloud infrastructure, servers are equipped with large memory, a significant portion of which is used by the virtual machines (VMs), while the other portion is used by the host software. The latter includes the hypervisor, host OS kernel, drivers, system processes, and various host agents, *e.g.*, an agent that manages networking of the VMs. In this work, we focus on memory leak issues in host software, not leaks in customer VMs. Unless otherwise specified, the kernel, drivers and processes hereafter refer to those in host software stack.

Leaks in the host software can cause severe performance degradation and even instability of the host OS. They can further impact the running VMs, because memory between VMs and the host is not strictly partitioned, typically controlled by a soft threshold [45]. They can also cause potential VM start-up failures due to insufficient physical memory available.

The host memory is divided into user-mode memory and kernel memory. The host OS in Azure's infrastructure distinguishes four states for pages in a process' virtual memory: *free*, *reserved* (for future use but no physical page is allocated), *committed* (memory has been allocated from physical memory or paging files), and *shared*. For memory leak detection, we only need to consider pages in the committed and shared states. For kernel memory, the kernel creates two types of memory pools: *non-paged* pools and *paged* pools. Virtual memory in the non-paged pool is guaranteed to reside in physical memory as long as the kernel objects are alive, whereas memory in paged pool can be paged out. Memory leaks in the kernel can happen in both types of pools.

2.2 Memory Leaks

Memory leak occurs when heap-allocated objects are not freed at appropriate time. It is manifested in two forms: (i) *unreachable* leak, in which an allocated object is no longer

```
1  // ConfigMonitorThread          5-sec timeout, previously
2  while (cm->running) {           it was set to INFINITE
3    waitStatus = WaitForSingleObject(
4                      fileChangeHandle, 5 * 1000);
5    if (waitStatus == WAIT_OBJECT_0) {
6      // object is signaled, config file has changed
7      ::Sleep(200);
8      cm->ReadConfig(); // read the file
9  +   if (!FindNextChangeNotification(fileChangeHandle))
10 +     throw ServerBaseException(
11 +         "Failed to get handle to config directory");
12   }
13 - FindNextChangeNotification(fileChangeHandle);
14 }
```

Figure 1: A production memory leak example in Azure from a host process that caused leaks of objects allocated at the kernel side.

reachable from the root objects such as global and stack variables; (ii) *forgotten* leak, in which an allocated object is still reachable but no longer accessed. The first type does not occur in managed languages like Java. For the second type, since the program still keeps references to the leaked object, it cannot be reclaimed even with managed languages [12]. Such leak is challenging to be detected because whether an object will be accessed in the remaining execution is undecidable. Thus, a leak detection solution can only output conservative (correct) answers, *e.g.*, the objects that are definitely dead at a given time point, or approximate answers (which may be incorrect) such as inferring based on the object's *staleness* [17].

Memory leaks in cloud software have further complications. For instance, while existing solutions focus on detecting leaks in an individual component, a memory leak in cloud infrastructure often happens because of API contract violations between different components, which is not well addressed. This type of leak is hard to expose in pre-production environment, because software components are often tested separately and integration testing cannot cover all possible interactions. Slow leaks also unlikely get detected due to testing time constraints.

Figure 1 shows a real example of such a leak in Azure (this case was successfully caught by RESIN). The process has a thread that monitors the configuration file updates using `WaitForSingleObject` with a 5-second timeout. In each loop iteration, it calls the `FindNextChangeNotification` API (line 13). Each invocation causes the kernel to allocate I/O request kernel objects from the non-paged pool memory. The contract of the `FindNextChangeNotification` is that it must be followed by a call to a wait function, and if the wait function returns for any reason other than the change notification handle being signaled (*e.g.*, timeout), the wait must be retried. In this case, although the process calls the wait function, it unconditionally calls `FindNextChangeNotification` even if the wait returns timeout. Thus, the kernel objects are allocated every 5 seconds without being cleaned up. In this incident, the culprit process' memory usage was not high. The kernel was experiencing memory leak in its non-paged pool, not because of kernel bugs but rather the improper API usage in the process' code. This memory leak was introduced during a bug fix for another issue: previously the process waits for the updates using an `INFINITE` parameter in line 4, but this caused service restart operations to be blocked, so developers changed the wait parameter to a timeout of 5 seconds.

2.3 Requirements

There are several challenges and requirements for addressing memory leaks in cloud infrastructure software:

- *Highly scalable.* Cloud system is large in the number of components, codebase size, and deployment scale.
- *Versatile.* Memory leaks in cloud infrastructure manifest themselves in various ways—in processes, kernel, unreachable leaks, forgotten leaks, cross-component leaks, *etc.*
- *Non-intrusive and low-overhead.* Solutions that require intrusive modifications or incur high runtime overhead are hard to be deployed in production.
- *Accurate.* True leaks should be detected. False positives should be minimized, because they would cause developers to waste significant time investigating false issues.
- *Timely.* If the leak detection is too slow, significant damage to customers may already occur.
- *End-To-End.* Only alerting memory leaks is insufficient. Developers also need considerable help in confirming the issue, pinpointing the root cause, and mitigating the leak.

Additional constraints include generality and efforts of integrating a solution. The software components in cloud infrastructure are written in different programming paradigms, and may depend on proprietary libraries. The millions of nodes in Azure also have heterogeneity with different OSes, libraries, and hardware versions. Supporting all of these varieties is challenging. For example, we made an experimental effort of integrating the LeakSanitizer [1], a popular run-time memory leak detector from the LLVM project, into one Azure host component's codebase. The integration effort was difficult (took one person month) due to complex compilation flags, and library compatibility issues. The MSVC compiler's full support for LeakSanitizer is still pending [3].

3 Overview of RESIN

Despite the extensive efforts to address memory leaks in conventional settings, they are insufficient to satisfy the unique requirements for tackling memory leaks in large cloud infrastructure (Section 2.3). To address this gap, we propose RESIN. RESIN is a holistic system running in the Azure production infrastructure to detect memory leaks in host software and provide diagnosis support to developers easily pinpointing the leak's root causes. RESIN further performs automatic leak mitigation to reduce the impact of detected leaks.

Approach A large cloud infrastructure can have hundreds of components owned by different teams. Prior to RESIN, tackling memory leaks in Azure is a team-by-team effort. Some teams started investigating after incident reports about slow or failing VMs, and developers discovered leak bugs in their components during manual investigation. Some teams added telemetry monitors in their testing cluster and used

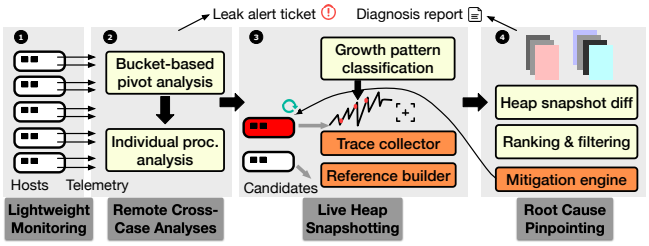


Figure 2: Workflow of the RESIN system.

hard-coded thresholds to trigger leak alerts in testing. Similarly, diagnosing leaks in Azure used to rely on developers to manually inspect the leaking nodes and run profiling tools. These individual practices were tedious and costed repeated engineering efforts. They also incurred significant false positives and could not handle cross-component leak issues.

RESIN takes a centralized approach instead. It does not require access to a component's source code, nor extensive instrumentation or re-compilation. RESIN uses a monitoring agent to each host that leverages low-level OS features to collect memory telemetry data. It automatically supports all components including the kernel. The data analysis is offloaded to a remote service, which minimizes the overhead to the hosts. By aggregating data from different hosts, RESIN can run more sophisticated analyses to catch complex leaks.

In addition, RESIN decomposes and tackles the memory leak problem in multi-level stages. It performs lightweight leak detection first and triggers more in-depth inspections on the fly when necessary for confirmation and diagnosis. This divide-and-conquer approach allows RESIN to achieve low overhead, high accuracy, and scalability together.

Workflow Figure 2 shows the workflow of RESIN. It starts with low-overhead monitoring (❶) at each host. A remote service analyzes (❷) the collected data across different hosts using a bucketization-pivot scheme. If a bucket is suspected of leaking, RESIN triggers an analysis on the process instances from that bucket. After the two steps identify a highly suspicious software component, RESIN automatically generates an alert ticket for that component along with a list of leaking process instances belonging to that component. Meanwhile, RESIN performs live heap snapshotting (❸) for the suspected processes. RESIN carefully chooses the snapshotting time using a growth pattern based algorithm to ensure the collected snapshots would be helpful. RESIN also samples normal processes to take regular heap snapshots and build a reference database. After generating multiple heap snapshots, RESIN tries to pinpoint the root causes (❹) by running a diagnosis algorithm on the snapshots. The analysis report will be attached to the alert ticket thread to assist developers. Finally, RESIN automatically mitigates the leaking processes.

4 Design of Leak Detection

In this section, we describe the RESIN's design for detecting memory leaks. In existing literature, the term "detection" refers to detect both (i) if a program or a process has a leak, and (ii) the bug in code or leaked objects. RESIN separates these as two tasks and uses the term detection to refer to (i) specifically. The diagnosis component (Section 5) targets (ii).

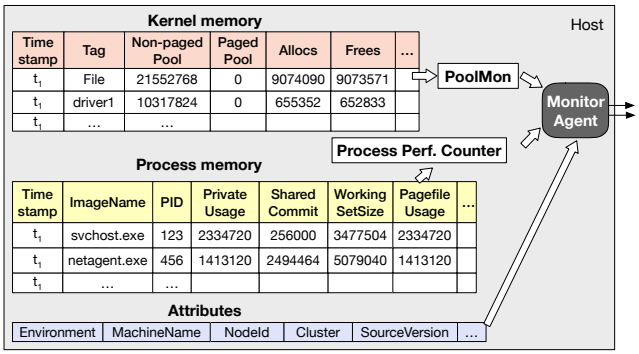


Figure 3: Monitor agent in each node collecting memory usage data.

4.1 Challenges

RESIN needs to address several challenges. First, cloud infrastructure software has highly noisy memory usage due to changing workloads and interference in the environment. Using static thresholds would generate many false positives. Standard anomaly detection algorithms [40, 41] do not work well either, because it is common for a component to exhibit memory usage spikes that are not leaks but legitimate increases in handling certain workloads.

Second, memory leaks in production systems are usually fail-slow faults [14] that last days, weeks, to even months (rapid leaks are likely caught in testing or deployment). Inspecting memory usage in a short time window would miss these slow leaks. It is necessary but challenging to capture gradual changes over a long period and still raise timely alerts.

Third, given the scale of Azure, collecting fine-grained data for a long time is impractical because of storage and overhead concerns. Therefore, RESIN can only collect limited, coarse-grained data and must work well under this constraint. Still, even with coarse-grained signals, the data volume is enormous. The detection algorithms must run efficiently.

4.2 Lightweight Memory Usage Monitoring

RESIN deploys a privileged monitoring agent on each host (Figure 3). This agent communicates with the host OS to track memory usage. It collects both kernel memory usage and per-process memory usage. The kernel usage is obtained from a pool monitor kernel module (`PoolMon`), and includes the usages of non-paged memory pool and paged memory pool for each *tag*. The tag is passed as an argument by the callers of the kernel allocation API [32] and represents a sub-system that has requested memory from the kernel allocator, *e.g.*, the file system, a driver. The per-process usage is obtained by querying the per-process performance counters from the host. It includes breakdowns of a process's memory, such as the private commit, working set size, paging file usage, *etc.*

We collect the memory usage breakdowns and tags instead of simply a single total memory usage metric, because mixing

different memory usage sources can introduce noises and miss important changes. For example, a 20 MB increase can be a leak for a driver but may be negligible for another component. Reporting the specific memory portion or tag that is leaking helps developers localize the buggy code. The breakdowns also help RESIN take more effective mitigation actions.

In addition to memory usages, the monitoring agent also records attributes such as the software version, hardware generations, node id, and cluster id. The attributes are used during leak detection analyses to increase accuracy. RESIN includes the common attributes of the leaked process in the detection report to give developers troubleshooting hints.

The monitoring agent is scheduled to run every 5 minutes. However, the data points from different hosts may not be perfectly synchronized. Some special events in a host such as node reboots also introduce missing or invalid data. Therefore, RESIN aggregates the time-series data into hourly granularity by removing extreme outliers and computing the mean of the remaining data points. This pre-processing step reduces the noises as well as the data volume. Using an hourly window is not too coarse-grained because most software components in cloud infrastructure are long running, and production leaks typically occur in a large time scale.

4.3 Detection Algorithms

RESIN uses a two-level scheme to detect memory leak symptoms: a global bucketing-based pivot analysis to identify suspicious components, and a local individual process leak detection to identify leaking processes. The detection output includes the suspected component, the list of top leaking processes of that component, the leak start and end times, severity scores, *etc.* The detection algorithms are language agnostic.

4.3.1 Bucketization-based Pivot Analysis

To address the challenges described in Section 4.1, our insight is that we should inspect at the *component* granularity across processes. This is because although an individual process' memory usage is influenced by workloads and highly noisy, the noises can be "canceled out" *en masse*. For a normal component, its process instances on different hosts may experience different workload effect at any time slice. But for a leaky component, the memory leak must be caused by some buggy release. Therefore, its processes should exhibit some global trend at certain time slices despite the workload effect.

Based on this insight, we design a simple yet robust bucketization-based pivot detection scheme (Figure 4). RESIN first groups the raw memory usage telemetry data into a number of buckets. In our implementation, we use 20 buckets (50 MB, 100 MB, 200 MB, ..., 40 GB, 50 GB). RESIN then applies pivoting to the data with a unique attribute tuple as the index and memory usage bucket as columns. The attribute tuple is (`ProcessImageName`, `ServiceName`) for user-level software, and (`TagName`, `PoolType`) for kernel subsystems, where `Type` is paged memory or non-paged memory. The aggregation function is the count of distinct nodes. Thus, each summary

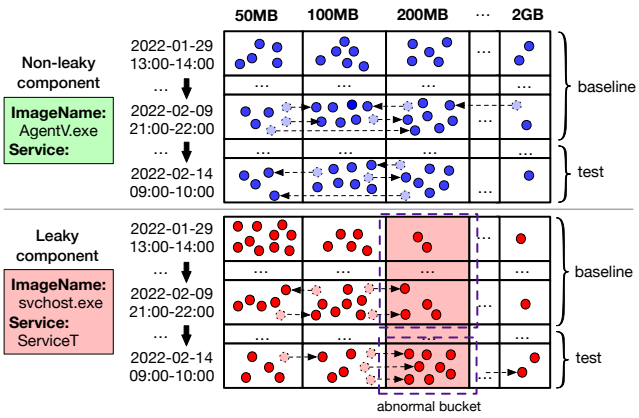


Figure 4: Group the memory usage into buckets and pivot by image name, service, and bucket size. Each circle represents one process. Shaded circle represents a process moving to another bucket.

cell represents the number of nodes that have running processes with a particular attribute tuple and these processes' memory usages fall into the specific bucket. RESIN computes the summary periodically and incrementally for data in each time interval. The results are saved into a database table.

We basically transform the memory usage data into summary about numbers of nodes in different buckets, which can more robustly represent the trends and tolerate noises due to workload effect (*e.g.*, the non-leaky component in Figure 4). RESIN then runs anomaly detection on the time-series data of each bucket for each component. It uses the most recent time period of summary data (default 15 days), with the first $2/3$ portion as the baseline and the remaining data points as the test. If a bucket's test period has data points that exceed the $\mu + 3\sigma$ of the baseline data (μ and σ represent the mean and standard deviation of the distribution), it is considered to be an anomaly. The start time and end time in the test period when the node count becomes the outlier are recorded.

One caveat is that if many processes of a component experience a sudden *drop* in memory usage, the node count would shift from a higher bucket to a lower bucket. But the lower bucket's node count significant increase is not an anomaly. To handle such scenarios, RESIN calculates cumulative bucket values during the anomaly detection. In other words, if the 100 MB bucket has a node count of n, it means there are n nodes that have the particular processes with memory usage equal to or larger than 100 MB, including processes (if any) that fall into the 200 MB bucket. In this way, a significant increase in a bucket almost always suggests an anomaly.

The bucketization approach also helps address the computation challenges. Before introducing this approach, it can take RESIN more than one day to run anomaly detection on the enormous data points from millions of nodes. After the pivot summary, RESIN only needs to run anomaly detection for the time-series data in each bucket, which can finish in less than one hour for all data (even without parallelization).

RESIN calculates a severity score for each bucket based on the deviations and node count in the bucket. It considers a

component is leaking based on a <*size_mb*, *score*> threshold: if a bucket is of size equal to or larger than *size_mb* and its severity score exceeds *score*. RESIN generates intermediate reports for the abnormal buckets. It de-duplicates the intermediate reports by only keeping the one for the largest bucket of a unique attribute tuple, and generates a ticket for that report.

4.3.2 Localizing Individual Processes

The bucketization pivot analysis works at the component granularity. RESIN uses a second-level detection scheme that works at the process[1] granularity. The motivation for this scheme is that a component has many process instances. It is important to localize the truly leaking processes in the alerting bucket. If we simply include all the processes in the abnormal bucket, developers can waste significant effort investigating innocent processes that fall into that bucket by coincidence.

The second-level detection scheme computes the leak likelihood and severity for a process based on its memory usage. RESIN uses all the memory usage data of a component in the most recent month to train two parametric models: (i) the *absolute usage model* and (ii) the *usage difference model*. Since different clusters and regions can exhibit drastically different characteristics, the tool builds separate models for each combination of region name and cluster type for a component.

Let $U_c(n_i,t_j)$ denote the memory usage value for a process of component c on node n_i at time t_j. RESIN assumes *absolute usage* $U_c(n_i,t_j)$ follows a Gaussian distribution $\mathcal{N}(\mu_1,\sigma_1^2)$ and fits the memory usage data by calculating the maximum likelihood estimators for μ_1 and σ_1. The absolute memory usage values can be severely distorted by occasional events such as VM creations. To account for such events, we consider the differential memory usage, *i.e.*, $\Delta U_c(n_i,t_j) = U_c(n_i,t_j) - U_c(n_i,t_{j-1})$. Based on our observations, when noisy events such as VM creations occur, $\Delta U_c(n_i,t_j)$ usually significantly deviates from its normal range. Thus, RESIN also builds a parametric Gaussian distribution $\mathcal{N}(\mu_2,\sigma_2^2)$ model for *usage difference* $\Delta U_c(n_i,t_j)$ and calculates the μ_2 and σ_2.

With the offline models, RESIN uses a *moving suspicious interval* algorithm (Algorithm 1) to examine a suspected process' memory usage in *real time*. This algorithm works by keeping a *suspicious leak time interval* $[T_0,T_1]$. The basic idea is to assume the leak still continues at the end of the time series and try to find the earliest time the leak trend starts by skipping over low-confidence points. This interval is initialized as $[t_1,t_1]$ upon reading the first data point in a time series. At the j-th step, RESIN reads $U_c(n_i,t_j)$, calculates the $\Delta U_c(n_i,t_j)$, and adjusts the time interval by moving T_1 and update T_0 adaptively. If $\Delta U_c(n_i,t_j)$ has a significant increase or drop (based on the 3-sigma rule for μ_2 and σ_2), T_0 is updated to t_j because the system status is likely changed by some event. If $U_c(n_i,t_j)$ is lower than $U_c(n_i,T_0)$ or there are few increasing points in the current interval, T_0 is also updated

[1]Here we use the term "process" to also include a running instance of a kernel subsystem in a particular host for kernel memory leak detection.

Algorithm 1: Moving suspicious interval algorithm

Input: $U_c(n_i,t)$: time-series memory usage for node n_i of component c; $\mathcal{N}(\mu_2,\sigma_2^2)$: offline usage difference model for component c.
Output: T_0, T_1: leak start and end time; no leak if T_0==T_1. N_{inc}: number of increasing data points

$t_n \leftarrow \max(t)$ in $U_c(n_i,t)$, $N_{inc} \leftarrow 0$
$T_0 \leftarrow t_1$, $T_1 \leftarrow t_1$
for $j \leftarrow 2$ *to* n **do**
  $T_1 \leftarrow t_j$
  $\Delta U_c(n_i,t_j) \leftarrow U_c(n_i,t_j)$ - $U_c(n_i,t_{j-1})$
  if `IsOutlier`$(\Delta U_c,\mathcal{N})$ || $U_c(n_i,t_j) < U_c(n_i,T_0)$ || $N_{inc}/n < \varepsilon$ **then**
    $T_0 \leftarrow t_j$
  if $T_0 == T_1$ **then**
    $N_{inc} \leftarrow 0$ `/* empty interval, no leak, reset */`
  else if `IsLarger`$(U_c(n_i,t_j),U_c(n_i,t_{j-1}),\mathcal{N})$ **then**
    $N_{inc} \leftarrow N_{inc}+1$ `/* a new increasing data point */`
return T_0,T_1,N_{inc}

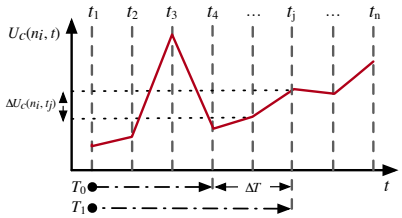


Figure 5: Applying the moving suspicious interval algorithm.

to t_j because a leaking trend should have enough increasing values. For other situations, we keep the T_0 intact. The loop stops when T_1 hits the last time point t_n. If the final T_0 is equal to T_1, this process is not considered as leaking.

Figure 5 shows an example of applying the algorithm. $[T_0,T_1]$ are initially set to $[t_1,t_1]$. When T_1 is set to t_2, $\Delta U_c(n_i,t_j)$ is positive thus we keep T_0 unchanged and continue to move T_1 forward. When T_1 is set to t_3, $\Delta U_c(n_i,t_j)$ is an outlier in the offline model $(\mathcal{N},\mu_2,\sigma_2^2)$, which we consider an occasional event instead of a leak. We reset T_0 to t_3 accordingly. When T_1 is set to t_4, $U_c(n_i,t_j)$ is significantly lower than $U_c(n_i,T_0)$ thus we also reset T_0 to t_4. After t_4 we did not encounter scenarios to reset T_0 (the memory usage drops slightly later but it is unnecessary to reset for such cases), so eventually T_1 reaches the end t_n, and $[T_0,T_1]$ is $[t_4,t_n]$.

RESIN calculates a severity score (Equation 1) for a process to indicate its leak probability and impact. Several factors are considered, including the normalized memory usage difference ($\Delta U_c = U_c(n_i,t_n) - U_c(n_i,t_1)$), the length of the suspicious leak interval ($\Delta T = T_1 - T_0$ in the unit of month), the increasing rate (number of increasing data points over the number of all data points), and the final memory usage at t_n.

$$SevScore = \frac{\Delta U_c}{\sigma_1} + \frac{N_{inc}}{n} + \Delta T + \frac{1}{1+e^{-U_c(n_i,t_n)/\mu_1}} \qquad (1)$$

For efficiency, the above analyses are run proactively. When the bucketization-based pivot detection step identifies a leaking bucket, RESIN triggers the individual process analysis for the processes in the bucket and can usually inspect the results without waiting. It outputs the suspected leaking processes, the leak start and end time, and the severity scores.

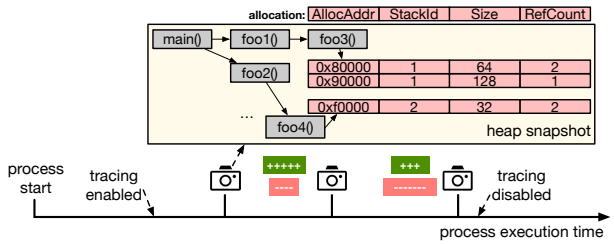


Figure 6: Periodic heap snapshot collection.

5 Diagnosis of Detected Leaks

Only detecting a leak is not enough. Without sufficient evidence and diagnosis support, developers are likely stuck in confirming and diagnosing the issue. RESIN designs a solution that automatically takes *live* heap snapshots and analyzes the snapshots to pinpoint the root cause of a detected leak.

5.1 Background: Heap Snapshot

RESIN provides diagnosis information at stack trace granularity. In our experience, if developers are presented with a stack trace containing the problematic allocations, they can often quickly debug the issue. RESIN leverages the Windows heap manager's snapshot capability to perform live profiling. The heap manager exposes APIs such as `HeapAlloc`, `HeapReAlloc`, and `HeapFree`, which are used by applications and C/C++ runtime to allocate heap objects. Thus the heap manager has the ability to collect the heap allocation sizes and stack traces.

RESIN uses the Windows Performance Recorder [2] to notify the kernel to start tracing heap allocations, typically for a specific process ID and occasionally for an image name (which enables tracing for all processes with the image). Then RESIN instructs the heap manager to take a snapshot at a certain time. Our current heap snapshotting mainly focuses on C/C++, which are the primary language choices for host software on Azure. Extending to other languages would take extra effort but are still straightforward, as their runtime typically already provides the functionality to capture allocation events.

To minimize overhead, the heap manager only stores limited information in each snapshot. Specifically, it stores (1) the stack trace and size for each *active* allocation after the tracing was enabled (if an allocation has been freed, no information is stored), (2) the total allocation sizes for each unique stack trace, and (3) the number of times a unique stack trace is invoked. It does *not* store more detailed information such as the allocation time or a pointer graph.

The information in a single snapshot is usually too noisy, as it includes all active allocations from the tracing start to the snapshot point. To get more accurate information for a time window, RESIN periodically takes *multiple* heap snapshots (Figure 6) to increase the chance of capturing truly leaking allocations between snapshots. RESIN uploads the snapshot files to a remote storage service. The diagnosis engine uses these snapshots to deduce the leaking allocation points.

5.2 Choosing Candidate Hosts to Profile

Picking the right hosts to take heap snapshot is vital for diagnosis effectiveness. Because heap snapshot incurs overhead, RESIN cannot afford to enable snapshot on all hosts containing the leaking processes the detection engine outputs. Simply choosing the hosts randomly is not a good strategy either, because the workloads on different hosts vary widely. For the same leak bug, it can exhibit in quite different patterns on different hosts. Thus, we may choose a candidate host in which the buggy allocations are triggered rarely.

We rank the candidate hosts in the suspected list based on three factors: 1) *severity*: choose processes with higher severity scores as described in Section 4.3.2, since more obvious symptoms suggest a better chance to be diagnosed; 2) *noisiness*: choose processes with a clearer growth pattern, which we will discuss in more detail in Section 5.3; 3) *impact*: choose hosts that have fewer user activities to minimize impact of profiling events. By default RESIN triggers snapshot collection for the top three hosts in the list in case the collection fails unexpectedly (*e.g.*, due to target process restart).

5.3 Deciding Trace Collection Strategy

With the candidate hosts selected, the next step is to decide if a *new* leak happens in the most recent snapshot interval and whether to take the snapshots. This step is different from the analysis in Section 4.3, which only finds leaking processes in *past* time. The decision making has two main challenges.

First, many production leaks are only triggered by specific events. Some leaks only occur once in several days. If we take snapshots at other times, the collected traces would not be helpful. To ensure rare leaks are captured, RESIN attaches the profiling workflow to the process for a long time and periodically (every half hour) takes snapshots in hope of capturing the leak. However, we cannot afford to keep uploading snapshots due to storage and overhead concerns. RESIN addresses the challenge with a *long-term, trigger-based* strategy: it uses a circular buffer that only keeps the most recently taken snapshots, and completes tracing once certain trigger is met.

Second, how to decide when the trace collection should complete, *i.e.*, the trigger. At the completion time, we should ideally (1) have snapshot(s) containing the buggy allocation; (2) have snapshot(s) for non-leaking scenarios; (3) minimize noisy allocations in the snapshot(s). One potential trigger is to complete the collection once the memory usage difference exceeds some threshold. This trigger can easily complete the tracing prematurely (fails to capture the buggy allocation) due to a legitimate memory usage spike, and/or produces snapshots that have many noisy allocations and mislead diagnosis.

To gain some insights on how to choose the triggers, we study the memory usage data of 51 real leak cases. Interestingly, most cases fall into three common patterns (and a mixture of them). Additionally, one leaking process in a specific host often has a *consistent* pattern. In 63% of the cases, the leaking process shows a *steady* pattern. One example is

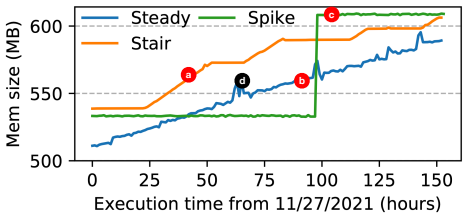


Figure 7: Memory growth patterns and completion point choices. Each data point is real memory usage from production processes.

Pattern	Characteristics	Completion Trigger
Steady	Almost linear growth	$R_j^2 > \lambda_a$.
Stair	Steady growing and flat curves alternately appears.	$\lvert 1 - slope_j/slope_k \rvert < \lambda_b$ && $R_j^2 < \lambda_c$
Spike	A few large allocations in a short period of time	$\Delta U_c(n_i,t_j)/\Delta U_c(n_i,t_k) > \lambda_d$

Table 1: Leak patterns, characteristics and their completion triggers.

the bug shown in Figure 1, in which the leak exists in periodical update tasks. The other two common patterns are *stair* (the memory usage occasionally grows only when the procedure containing leaks is activated) and *spike* (the leak only occurs once in a while due to rare events). Each pattern has its unique usage growth characteristics and clear completion point candidates ⓐ, ⓑ, ⓒ shown in Figure 7. ⓓ is not a good completion point as it is right after a period of some noises.

Guided by the findings, RESIN takes a pattern-based approach to decide the trace completion triggers. It uses the target process' memory usage data in the most recent week and classifies it into one of the three patterns. Specifically, RESIN first identifies nearly flat segments in the time-series data and removes them. It then performs linear regression on the remaining growing segments and outputs results including the slope $slope_k$, coefficient of determination R^2, and absolute memory usage increase $\Delta U_c(n_i,t_k)$. If the data contains no flat segments, RESIN marks it as a *steady* pattern. If the data has flat segments in between growing segments, it is marked as a *stair* pattern. If a large increase in memory usage only occurs in a few data points, it is marked as a *spike* pattern.

After the pattern is classified, the workflow starts to monitor and analyze the recent memory usage. We compute the $slope_j$, R_j^2, and $\Delta U_c(n_i,t_j)$ for the most recent six hours and check the pattern's completion trigger based on rules listed in Table 1 (λ_a, λ_b, λ_c and λ_d are set to 0.8, 0.1, 0.1, and 0.5, respectively). Once the trigger is satisfied, RESIN stops tracing and uploads the trace file that contains the most recent few snapshots. It ensures each trace has at least three snapshots.

5.4 Collecting Reference Snapshots

One challenge in using snapshots for diagnosis is the presence of many noisy but benign allocations. Even with multiple snapshots, they may remain active and mislead the diagnosis. RESIN collects *reference snapshots* to address this challenge.

For the reference snapshots to be useful, they should be comparable to the snapshots from the leaking process. A poor choice of a reference snapshot may be even counter-

Algorithm 2: Heap snapshot diagnosis algorithm

Input: A_{n-1},A_n: sets of allocations in two heap snapshots, S^r: a list of outstanding stacks from reference hosts, *pattern*: classified pattern, *estimate_leak*: upper bound of estimated leaking size
Output: S^o: a list of top N stack traces that likely caused leaks

```
S^o ← [], S^diff ← []
S^n ← A_n.groupBy(alloc=>alloc.stackid)
S^{n-1} ← A_{n-1}.groupBy(alloc=>alloc.stackid)
foreach stack ∈ S^n do
    if stack ∈ S^{n-1} then A^diff ← S^n[stack.id] \ S^{n-1}[stack.id]
    else A^diff ← S^n[stack.id]
    if A^diff ≠ ∅ then
        foreach a ∈ A^diff do stack.size ← stack.size+a.size
        S^diff.add(stack)
S^diff.orderBy(stack=>stack.size)
if pattern ≠ SPIKE then
    S^diff.removeAll(stack=> stack.size > estimate_leak)
S^diff.removeAll(stack=> stack ∈ S^r)    /* filter references */
S^o ← S^diff.top(N)                      /* only keep top N stack traces */
return S^o
```

productive and filter out the culprit allocations. RESIN uses a periodical *fingerprinting* process to build reference snapshots. It randomly samples hosts for common leaking services to take heap snapshots. We currently define the fingerprints to be the attribute tuple `(cluster_id, OS version, service version, date)`. This is based on our observations on the locality of memory leaks. These snapshots are saved in a reference database and cleaned up when they become stale.

At the diagnosis stage, after RESIN chooses the candidate leaking hosts to profile (Section 5.2), RESIN checks if the database already has reference snapshots with similar fingerprints. If not, RESIN triggers reference collection. It first scans the qualified hosts (not in the detection engine's suspicious list and have similar fingerprints to the leaking hosts) and samples a few that have active memory activities and modest memory usage. Then RESIN applies the growth pattern analysis (Section 5.3) to check if this host is leaking. If not, it takes snapshots and uploads the traces to the reference database.

5.5 Trace Analyses for Diagnosis

The next step is to analyze the collected snapshots to output the root cause stack traces. The challenge is to handle many noisy allocations and localize the buggy allocations.

RESIN designs a diagnosis algorithm listed in Algorithm 2. The inputs are the allocations from the two most recent snapshots of a trace file (A_{n-1}, A_n), stack traces from reference snapshots, and the estimated leaked size upper bound calculated in the pattern analysis. For the *steady* and *stair* patterns, we estimate the leaked size upper bound by multiplying the slope with the time interval of the growing segments and a coefficient (by default 2). The goal is to find the stack traces that allocate objects of sizes closest to the estimated leak.

The diagnosis engine first groups allocations in A_{n-1}, A_n by the stack trace id and get two maps S^n and S^{n-1}. Each map value is all the allocations that come from a stack trace. It then traverses each stack trace in S^n to calculate the aggre-

gated allocation size. The engine then identifies stack traces that contain unique allocations, and ranks these traces based by their allocated object sizes. Stack traces allocating sizes larger than the estimated leak size are likely noises and thus removed. Finally, the diagnosis engine cross-checks the reference snapshots to filter out benign stack traces. If the output list is empty, the engine repeats the analysis for the next snapshot pairs (A_{n-2}, A_{n-1}), *etc.*

6 Mitigating Leaks

When a memory leak is detected, it can take time for developers to come up with and deploy the bug fix. To avoid further customer impact, RESIN attempts to automatically mitigate the detected leak issues. Depending on the nature of the memory leak, mitigation can be done in several ways. Rebooting the host OS in general can mitigate all kinds of leaks. However, this is costly and potentially causes VM downtime.

RESIN leverages the results from its detection engine and uses a rule-based decision tree to choose a mitigation action that can minimize the impact. If the memory leak is localized to a single process or Windows service, and this process or service is not required to be always alive to provide services to customers, RESIN attempts the lightest mitigation by simply restarting the process or Windows service.

For some processes, the mitigation requires additional steps. RESIN allows component teams to define custom scripts and invoking conditions. If the leak is located in buggy drivers, RESIN unloads and reloads the driver to mitigate the issue.

For safety, RESIN uses allowlists for each action category to make sure auto-mitigation is not misused. It defines an initial allowlist for the processes and drivers that are known to be safe to restart. A feature team can opt in auto-mitigation by adding the name of the process or tag to the allowlist.

For leaks in the OS kernel memory such as I/O request objects and file objects, if the detection engine can attribute the leak to a process or a service, RESIN attempts to restart the culprit process or service. This action is usually effective because it allows the leaked kernel objects to be properly freed without the need to reboot the OS.

OS reboot will resolve any software memory leak but takes a much longer time and can cause VM downtime. Thus, it is the last resort when a leak cannot be mitigated by the above actions or the name is not in the allowlist. RESIN checks if the host is empty and does the OS reboot if so. Empty hosts could also leak memory due to past user activities or current background processes. For a non-empty host, RESIN first performs live VM migrations [8]. Then it attempts a kernel soft reboot, which skips hardware initialization. If the soft reboot is ineffective, a full OS reboot is performed.

To minimize the impact of mitigation actions, RESIN closely monitors the leaking hosts. It prioritizes the actions on 1) nodes that fire low-memory related events, such as E2004 (low virtual memory) from the Windows resource exhaustion detector, E3122 (not enough memory to start VM) from Hyper-V; 2) nodes in regions with capacity issues; 3) nodes with host memory reservation overage; 4) nodes ordered by the leak size, leaking rate, and the predicted time-to-failure.

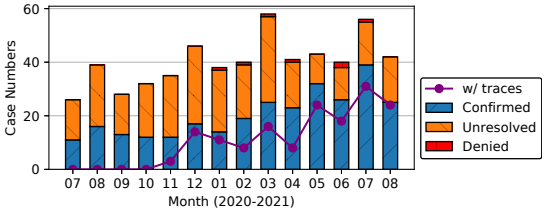


Figure 8: Memory leak cases RESIN detected and reported.

RESIN stops applying mitigation actions to a target when the detection engine no longer considers the target leaking. This typically occurs *without* manual intervention. For example, after developers identify the root cause and apply a fix, the leak symptom disappears, so RESIN stops the mitigation. Sometimes, after a mitigation action, the leak symptom no longer re-appears, which naturally stops further mitigation.

RESIN also coordinates with its diagnosis engine (Section 5) in performing the mitigation actions. If the diagnosis engine plans to or is taking heap snapshots for a candidate host, RESIN defers the mitigation actions to avoid losing the critical opportunities for capturing the leaking allocations.

7 Evaluation

Our evaluation answers several questions: (1) how effective is RESIN in detecting memory leaks? (2) how accurate is the detection? (3) can RESIN help developers diagnose and mitigate leaks? (4) what is the overhead of trace collection?

7.1 Deployment Status and Scale

RESIN has been running in production in Azure since late 2018. It covers millions of hosts, over 600 different host processes and over 800 different kernel pool tags daily. The detection engine in RESIN analyzes more than 10 TB memory usage data every day. The diagnosis module collects 56 trace files on average (10–200 MB) daily. Every month, the mitigation engine performs a median of 1,592 process restarts, 1,290 kernel soft reboots, and 4,649 node reboots.

7.2 Detecting Production Memory Leaks

Azure has various solutions that help eliminate memory leak bugs before production, including code reviews, static bug finding tools, testing, and safe deployment policies. As a result, only complex memory leak bugs occasionally escape these solutions. RESIN serves as the last defense to effectively catch these bugs in production.

Figure 8 shows the memory leak tickets RESIN reported in Azure from July 2020 to August 2021. Overall, RESIN reported 564 tickets in 14 months, among which developers explicitly resolved 291 tickets.

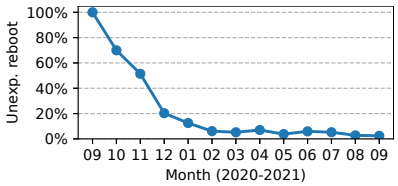


Figure 9: Unexpected VM reboot.

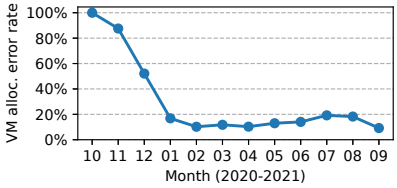


Figure 10: VM allocation error rate.

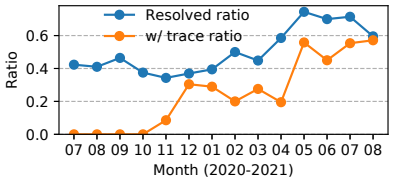


Figure 11: Ticket resolved and w/ trace ratio.

7.3 End-to-End Impact

The end-to-end benefits brought by RESIN are clearly demonstrated by two key metrics: (1) *VM unexpected reboot*[2]: the average number of reboots per one hundred thousand hosts per day due to low memory; (2) *VM allocation error*: the ratio of erroneous VM allocation requests due to low memory.

As shown in Figures 9 and 10 (data is normalized for confidentiality), the improvement RESIN provides is significant. Both metrics show large decreases: VM reboots are reduced by 41× from September 2020 to September 2021, and allocation error rates are reduced by 10× from October 2020 to September 2021. Note that these key metrics have been continuously dropping before the starting points in Figures 9 and 10. We omit to plot these earlier points because the raw data are no longer in the database due to data retention policies.

The improvement also shows in the reduction of service incidents. In 2020 and 2021, no severe outages in Azure were caused by memory leaks (such outages occurred previously).

7.4 Effectiveness of Detection

Detecting memory leaks in a complex, frequently changing cloud infrastructure like Azure is challenging. RESIN aims to minimize the false positives and false negatives.

Precision To evaluate the detection accuracy, we count how many cases RESIN reported are flagged by developers as false alarms. Overall, RESIN only incurs 7 false positives out of 291 resolved cases in 14 months.

There are two common patterns of false positives: (i) after a component adds a new feature that consumes significantly more memory; (ii) after a configuration change, *e.g.*, *"we are collecting and correlating lot more counters and # of disks per node have also increased"*. The memory usages in these cases typically stabilize and become new baselines, which RESIN automatically picks up without requiring adjustment.

Recall We count how many cases RESIN misses or fails to detect in time. The criterion is a memory leak causing noticeable service impact and getting reported by users, developers, or other monitoring services before RESIN detects it. We search keywords including "memory leak" and "leak" on Azure's issue tracker used by all teams. We compare them with tickets automatically created by RESIN. Overall the false negatives are few: only 4 cases are not on RESIN's ticket list.

We also inspect the reason for each case. One case in July 2020 was caught by us manually when analyzing a heap snapshot trace, before the issue triggered RESIN's alert. In other two cases, the leak impact was not significant, but developers found the leaks when they were closely monitoring their new deployment. The last case was both found by developers and RESIN, but developers found the issue faster. This happened because the issue manifested itself earlier on a testing cluster that had a different workload from the production clusters RESIN monitored; developers were also using an aggressive threshold in that cluster to expose potential issues.

Timeliness It can be difficult to determine the exact starting time of a leak and hence the exact detection delay. We observe that RESIN's detection typically occurs within two hours of a leak's clear manifestation. In general, a too-long detection delay would be reflected in a false negative since developers or other monitoring tools would detect the issue earlier.

Resolution rate Not all cases are eventually resolved by developers. On average, RESIN reports a resolution rate of 52% within reported date range. We believe this resolution rate is underestimated compared to actual responsive rate: in many cases, developers took actions but did not update the open tickets (only tickets tagged with a high severity level are mandatory to resolve according to ticket platform's policy). "Unresolved" also does not necessarily mean the ticket is unimportant. We observed that some teams tend to have lower response rates, likely due to their limited resources and the overwhelming number of urgent tickets.

This result is also influenced by our design goal. RESIN is designed to catch memory leaks early before the leak escalates to catastrophic issues. RESIN further provides automatic leak mitigation. Thus, a could-have-been-severe leak would appear to be low-risk. When developers prioritize resolving high-impact tickets, it adversely affects the resolution rate for our leak tickets. Due to the large volume of reported cases from various groups, we could not afford to inspect all unresolved tickets and check with their owners.

7.5 Effectiveness of Diagnosis

The diagnosis module of RESIN in total collects traces and generates reports for 157 cases from July 2020 to August 2021 (Figure 8). For tickets related to kernel leaks or clusters with legacy OSes, RESIN is unable to collect traces. Before November 2020 the diagnosis module was in trial runs and diagnosis reports were not appended to tickets. We gradually enabled it for more clusters after the trial run was over.

Figure 11 shows how the ratio of tickets with collected traces (and diagnosis reports) increases, which correlates with

[2]In this paper, we only count those VM unexpected reboots and allocation errors caused by low memory and memory exhaustion.

```
1  virtual void AddDsmsCertificate(CertificateStore& ... {
2  -    for (; certHead != nullptr; certHead = certHead->Next) {
3  +    for (auto currentCert = certHead; currentCert != nullptr;
4  +         currentCert = currentCert->Next) {
5  -       if (certHead->Versions == nullptr)
6  +       if (currentCert->Versions == nullptr)
7             continue;
8  -       auto latest = certHead->Versions->Latest;
9  +       auto latest = currentCert->Versions->Latest;
10         if (latest == nullptr)
11            continue;
           ...
20     freeCertLst(certHead)
```

Figure 12: The fix for ServiceH leak.

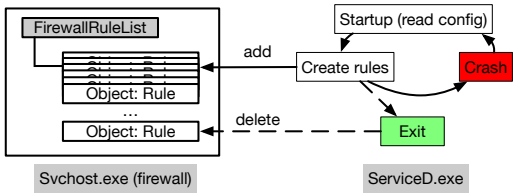


Figure 13: Contract violation induced leak on ServiceD.

the improvements on the ratio of resolved tickets.

Usefulness To evaluate the usefulness of the diagnosis reports RESIN generates, we randomly sample 14 issue tickets and closely follow up with the developers (all cases are eventually resolved and fixed). In 11 cases (79%), developers directly use the diagnosis reports to pinpoint root causes. Among them, in 5 cases the bug fix is in the same function in the allocation stack; in 5 cases the allocation stack and bug fix are in the same source file; in only 1 case the allocation stack and bug fix are in different components. Out of the other three cases, two are solved by memory dumps because the developers are quite experienced: upon seeing the sizes of leaked memory objects, they immediately realized which function the leaks came from. In one case the traces captured in the production cluster are not particularly useful. Instead, developers successfully captured some snapshots consisting of leaking stack on their own testing cluster.

Feedback Over time, developers build up high confidence in RESIN. We receive many pieces of positive feedback:

"(The result is..) incredibly useful. The information I had was enough."

"Thanks for pinpointing out the memory leak that we had been trying so hard to find over the past few days."

"Stack trace was sufficient for debugging this, it included the API call that was problematic."

Case studies We share two representative cases. The first case occurs in ServiceH[3]. This process' memory usage keeps increasing and gets restarted every few days. The diagnosis module in RESIN collects heap snapshots and pinpoints the root cause stack trace. After the diagnosis report is attached to the ticket, developers confirm and fix the issue in 3 hours.

In this case, the program uses a pointer to manage the list of certificates, and frees the pointer at the end of the function. However it also uses the pointer to traverse the list. In the end the pointer has moved and only a part of the list is freed (Figure 12). This is a day-0 bug introduced a long time ago, but is recently triggered due to added certificates to the machines.

[3]The service names are anonymized for reasons of confidentiality.

Mitigation	Count	50%	75%	90%	99%
Process restart	27,039	1.62	5.74	6.50	30.70
Kernel soft reboot	8,292	24.64	34.47	49.14	141.69
Node reboot	278,005	248.58	274.36	362.10	1382.61

Table 2: Single mitigation action execution time (seconds).

The second case represents another common (6 out of 14 cases we studied) type of leaks in cloud infrastructure: leaks due to contract violations in cross-component interactions. After RESIN reports a firewall-related `svchost` is leaking, the diagnosis module collects traces and reports a function in the rule list adding procedures after analyses.

Developers do not find bugs in this specific function at first, but the report prompts them to check the firewall rule lists on these machines. They then find the rule lists on these machines have been flooded with redundant rules. The reason is that the `svchost` process gets a firewall configuration from another program ServiceD. This program creates firewall rules at startup. Due to another bug, ServiceD keeps crashing, which causes it to miss deleting created rules and repeatedly recreate rules upon restarts (Figure 13). This in turn causes significant memory usage increases for the `svchost` program. Such a bug is hard to be detected statically.

Timeliness The diagnosis timeliness is also important to help developers. We measure the latencies of RESIN's heap snapshot collection and analysis. The median trace collection time is 61 minutes. For more than 80% of cases, the collection finishes within 10 hours. Note that the trace collection time is influenced by when a leak recurs in a suspected process. If the leak is sporadic, RESIN has to wait until the symptom reappears to capture the snapshot. For trace analysis, the median latency of the analysis jobs is 10 minutes.

7.6 Effectiveness of Mitigation

Mitigation procedure duration on leaked services Figure 14 shows the number of mitigated nodes of a kernel leak due to a buggy driver. At first, RESIN applied mitigation actions on a few nodes per day to test possible side effects. Once the mitigation actions reached production, RESIN applied mitigation to at most around 2,000 nodes per day with some fluctuations. The mitigation action volume then gradually dropped as the fix was being rolled out. Eventually the volume dropped to a few nodes a day, which were primarily nodes that failed in driver upgrading or other fix actions.

Mitigation action duration on single host We collect the frequencies and durations of each mitigation action between July 2020 to September 2021. As Table 2 shows, process restart is the most lightweight mitigation action. In most cases, it finishes within 6.5 seconds. Kernel soft reboot is also fast and in most cases finishes in a minute. Node reboot takes a longer time, with a median time of 4.6 minutes.

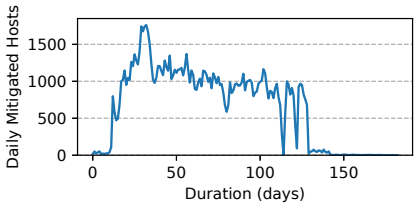


Figure 14: Mitigation for a leaking driver.

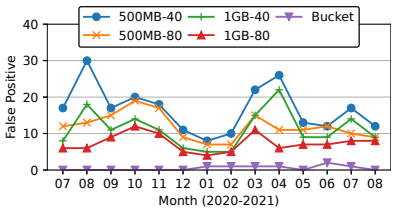


Figure 15: False positive of detection algo.

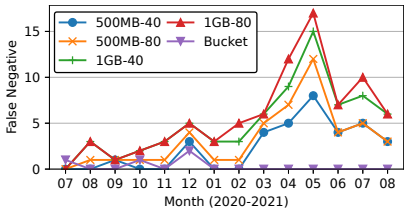


Figure 16: False negative of detection algo.

7.7 Comparison of Different Algorithms

Bucketization-based detection We first compare our core detection algorithm, the bucketization-based pivot analysis, with the practice of static threshold-based memory usage monitoring. We use four threshold policies, *e.g.*, policy "500MB-40" means generating leak alerts if a service's memory usage exceeds 500 MB on more than 40 nodes. We apply these hard thresholds to historical data and count how many cases will be wrongly reported as leaking (false positive) and how many leaking cases will be missed (false negative).

Figures 15 and 16 show the results. Our algorithm performs the best: it has both the lowest false positives and the lowest false negatives. In comparison, for other policies, it is often a dilemma to balance precision and recall. For example, policy "1GB-80" has the lowest false positives among the baselines at the cost of having the highest false negatives.

Pattern-based collection We compare our pattern-based collection with random collection. The experiment is conducted on ServiceS, ServiceV, and ServiceW. They have ongoing memory leaks on some hosts. We randomly choose six hosts and apply pattern-based collection on three hosts and random collection on the other three hosts. For the random strategy, we implement a workflow that periodically collects snapshots with at least two snapshots and completes the trace collection with a probability $1/6$. We inspect the collected heap snapshot traces to see if the leaking allocation exists in the snapshot.

Our pattern-based collection successfully captures leaking allocation stacks for all three services. Interestingly, the root cause of ServiceW was still unknown at the time we conducted the experiment. RESIN successfully captures an outstanding allocation that contains the bug within a real-time event processing function. In comparison, random collection only captures the buggy allocation for ServiceS, which has a frequent leaking interval (less than 1 hour).

Reference-assisted analysis We then evaluate the usefulness of reference snapshots with a controlled experiment on the ongoing leaking component ServiceS, which has the most noises among the three ongoing leak cases. We randomly sample eight hosts that have leaking patterns and collect snapshots until the leaking stack appears. We feed eight collected trace files to RESIN and compare the analysis results with and without reference snapshots. Figure 17 shows the result. Without the reference snapshots, the root cause stack trace ranks below the top three in all traces. With the reference snapshots, in 7 out of 8 traces, the root cause rank improves. In four traces, the rank rises to the top three, which largely narrows down the code regions developers need to investigate.

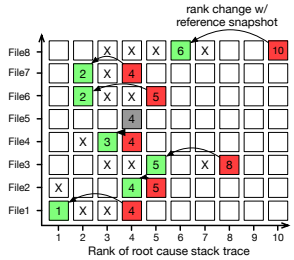


Figure 17: Ranks of root cause stack trace in diagnosis analyses on 8 trace files, w/ (green colored) and w/o (red colored) using reference snapshots. Cell with a number represents the rank. "X" marks the stack trace that gets filtered with the reference snapshots.

HasOverlap	Sessions	Nodes	25%	50%	75%	90%	95%	99%
FALSE	102,627	315	28	49	94	164	202	869
TRUE	165	31	38	50	59	86	241	888

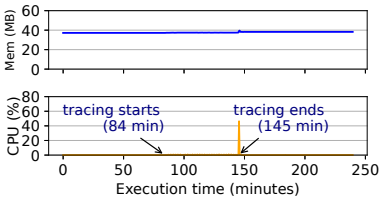
Table 3: VM deployment time (seconds) impact by trace collection.

7.8 Runtime Overhead

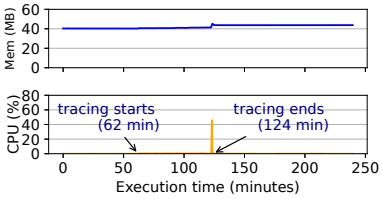
As a production service, RESIN should not impose significant overhead on the hosts. For the detection component, since RESIN leverages the kernel to collect performance counters infrequently and offloads the analyses remotely, the overhead is minimal. The main source of overhead is the heap snapshot trace collection. We use the VM deployment performance to quantify the end-to-end cost of trace collection, because VM deployment is the most important event for hosts and involves nearly all host services and triggers many critical code paths. A large overhead will be reflected in long deployment time.

We first check how many hosts RESIN performs trace collection on in November 2021. The result shows only 346 hosts are collected at least once, which is less than 0.1% of all nodes in a cluster. We then collect start and end timestamps of all VM deployment sessions and the heap snapshot tracing requests. We compare the timestamps in the two sets of events. In 315 (91%) of the 346 hosts, the deployment sessions do not have any overlap with tracing sessions, thus the tracing has no impact on these sessions.

Table 3 shows the end-to-end latency of the overlapped sessions compared to non-overlapped sessions: by 1 s for the median, and by 10 s for the 25th percentile. The latency increase could be notable for some short-duration deployments. However, this impact is limited to only a few sessions (0.16%) from a relatively small number of host nodes.



(a) host A



(b) host B

Figure 18: Memory size and CPU usage changes through tracing.

To measure the impact on memory size and CPU, we conduct experiments on two hosts that have active workloads. We trace one of the critical host processes. Figure 18 shows the memory and CPU usage during the experiment. Enabling tracing both slightly increases the average memory usage, 0.25 MB for host A and 0.53 MB for host B, and the CPU usage, 0.23% for host A and 0.22% for host B. When doing snapshot and dumping the trace, there is a clear spike for CPU usage: around 46% in both hosts. The memory usage also increases, but not significantly: 1.93 MB for host A and 3.89 MB for host B. After the tracing, the memory usage remains at a slightly increased level due to a one-time initialization required by tracer. Overall, the CPU and memory usage costs are acceptable in production deployment.

7.9 Tuning Effort

The software components and workloads in Azure infrastructure undergo frequent changes. As part of RESIN's design goals of minimizing false positives and false negatives, we aim to build robust algorithms that avoid fragile parameter tuning. For the detection part, throughout RESIN's production operation, we only made one major parameter change. We updated the alert threshold for the final result <*BucketSize*, *SeverityScore*> (Section 4.3.1) from <50 MB, 10> to <200 MB, 40> around 3 months after deploying RESIN. There has not been further tuning since then. For the diagnosis part, except for the initial trial runs in which we were experimenting with the snapshot algorithms, the parameters for the completion triggers have not been tuned after the diagnosis engine was enabled in production.

8 Lessons and Limitations

Lessons While memory leaks are generally taken seriously, developers tend to postpone the investigation if there are no convincing hints. Presenting clear evidence in results is critical, and significantly improves developers' responsiveness.

Many teams write extensive test cases that check if allocations are freed. Some teams also implement their own version of memory leak detection tool in their testing cluster. Developers mentioned a major pain point is that the testing environment has significant discrepancies with the production environment. For example, in one case, developers mentioned *"We don't have an environment where ServiceH runs for a really long time with hosts undergoing reboots."*, otherwise, their testing would have caught the memory usage anomaly.

Our initial thought in designing the diagnosis module is to analyze the source code of the detected leaking component. We later found that finding the root cause stack traces is usually good enough for developers to debug the issue based on their own experience and domain knowledge.

For production services, safety is of high priority. The cloud infrastructure is a complex and dynamic environment. Some workflow in RESIN can be interrupted abruptly, *e.g.,* due to transient network issues, interference with other profiling tools. On one occasion, RESIN accidentally left the trace collection running and triggered alarms in the detection engine. We set three lines of protection to prevent similar issues: (i) limit collecting on same cluster within one hour five times at maximum to reduce side effects; (ii) a forced cleanup operation whether the profiling succeeds or not; (iii) a workflow that periodically checks logs and cleans up for runaway hosts.

Limitations The telemetry data RESIN analyzes is relatively coarse-grained. Even the heap snapshot only contains limited information about allocations. Therefore, it has inherent inaccuracies and may miss detection of minor leak bugs. RESIN can be further enhanced by collecting more fine-grained signals, and leveraging semantic information from source code.

Developers may need to reproduce a reported memory leak issue for investigation or confirming bug fixes. But this is often challenging, because the issues are often triggered by complex workloads and rare conditions. RESIN does not address this challenge. We plan to automatically capture production triggering workloads for developers to reproduce leaks.

The patterns used in our heap snapshot trigger are based on empirical observations, which may be incomplete. Our classification method is simple. They can be improved with more comprehensive case studies and more advanced methods.

9 Related Work

Detecting memory leak bugs has been extensively studied in the context of conventional software. Our work focuses on addressing memory leaks in production cloud infrastructure, which face unique challenges as described earlier. Indeed, the memory leaks addressesed by RESIN are usually the ones that escape the bug detection and extensive testing practice in Azure and are only triggered in complex production workloads. The main research contribution of RESIN is its novel multi-stage approach and algorithms including the bucket pivot analysis and moving suspicious interval algorithm for leak detection, the live heap snapshot collection and analysis for leak diagnosis, and the decision tree based leak mitigation.

Dynamic leak detection. Many solutions have been proposed to dynamically detect memory leaks. There are broadly two approaches. In one approach, the tool records memory-related metadata by inserting checks to object code [16], using performance monitoring units in processors [24], instrumenting bytecode [39, 49], instrumenting intermediate representation [25, 34], instrumenting source code [21], modifying garbage collector or memory allocators [23, 35] or using metrics such as object staleness [17] as indicators to examine object lifetime. These works usually record fine-grained memory object information and have high accuracy, but they require rewriting codes or special hardware support, which is difficult and unsafe to apply in production settings.

Another approach analyzes heap snapshots/dumps [31, 33, 38, 44]. They are designed for interactive offline debugging and do not work well for long-running processes and services in production systems. They also rely on user-defined workloads as oracles to judge if memory growth is a leak. Obtaining such oracle workloads is difficult in practice. RESIN continuously monitors components in production cloud, designs robust algorithms to detect leaks without requiring oracles, and performs low-overhead live trace collection on-demand.

Some solutions [22, 40, 41] analyze memory usage patterns. They propose complex models to detect leaks in a single process or VM. The memory usage behavior of an individual process can be highly noisy due to workload effect and interference. Thus, they can have false positives and false negatives when applied in production cloud. Building complex models for each process in cloud scale also faces significant computation challenges. In comparison, RESIN focuses on the memory usage summary and global trend across processes, which enables accurate detection and efficient computation. RESIN additionally takes live heap snapshot and analyzes the snapshots to help developers localize the root cause.

Static leak detection. A wealth of work uses static analysis to find memory leak bugs. Many of them focus on improving the accuracies of static analyses [10, 15, 18, 36, 47]. Some other work focuses on finding specific leak code patterns. LeakChecker [50] finds objects created by the iteration are unnecessarily referenced by objects external to the loop. MLEE [46] finds leaks from early-exit paths by cross-checking the presence of memory deallocations on different early-exit paths and normal paths. Heapster [5] adopts a hybrid approach to leverage dynamic information to help static analysis. In general, while static approaches have the advantages of not requiring running a program, they face well-known scalability and accuracy challenges. They are also typically designed for a specific type of program. The software components in cloud infrastructure are highly complex and are written in a wide variety of programming paradigms. Also, static analyses cannot handle the forgotten leaks.

Leak fix and recovery. Some research work focuses on helping developers fix leak in addition to detecting them. LeakPoint [9] points developers to the potential fixable locations by taint analysis. LeakChaser [48] provides three layers of abstractions to assist programmers to diagnose memory leaks. Some other work focuses on automatically recovering the program from leaking. LeakSurvivor [42] and Melt [7] reclaim memory resources by swapping out objects to disks. LeakFix [11] inserts deallocations for leaks.

Statistical debugging. Statistical debugging [28, 29] uses statistical methods to identify predictors in the source code that correlate with a program failure. It requires instrumenting all predicates and re-running a program many times with normal runs and buggy runs. The diagnosis design in RESIN is complementary to statistical debugging. It collects live heap snapshots from production directly. Its algorithm identifies buggy stack trace based on the allocation information.

Failure detection and mitigation. Detecting memory leaks in production cloud is related to the topic of failure detection and mitigation in distributed systems [14, 19, 20, 27, 30, 43, 51]. Memory leaks are difficult to detect compared to other types of failures. IASO [37] detects fail-slow issues and supports mitigating slow issues with VM or node reboots. Narya [26] predicts node-level failures and performs mitigation actions. RESIN focuses on catching on-going memory leak issues, and provides a holistic solution. Its mitigation module leverages results from the detection engine to perform targeted mitigation to a specific process, service, driver, or host OS.

10 Conclusion

This paper presents RESIN, an end-to-end service designed to tackle memory leaks in production cloud infrastructure. RESIN takes a divide-and-conquer approach to decompose the memory leak problem, and designs a multi-level solution with novel algorithms including bucketization-based pivot analysis, live heap snapshot strategy, and diagnosis analysis. RESIN has been running in Azure for more than 3 years, and successfully reduces low-memory-induced VM reboots and new VM allocation errors by 41× and 10×, respectively.

Acknowledgments

We would like to thank our shepherd, Kathryn S. McKinley, and the anonymous reviewers for their thoughtful comments. We thank our colleagues who partnered with us on building the overall solution and providing feedback to us, including but not limited to Nathan Ernst, Anupama Vedapuri, Rui Ding, Carl Zhou, Francis David, Gaurav Jagtiani, Rakkimuthukumar Nallore Ponnusamy, Jayjit Phadke, Dustin Douglas, Matt Sebek, Harish Srinivasan, Kevin Broas, and Heejin Son. We would like to thank the strong support from Igal Figlin, Dongmei Zhang, Marcus Fontoura, Melur Raghuraman, Mark Russinovich, and Girish Bablani. This work was supported in part by the National Science Foundation grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757.

References

[1] LeakSanitizer – clang 13 documentation. https://clang.llvm.org/docs/LeakSanitizer.html.

[2] Windows Performance Recorder. https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder.

[3] Request for supporting LeakSanitizer. https://developercommunity.visualstudio.com/t/support-leaksanitizer/826620, 2019.

[4] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[5] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller. Heaps'n leaks: How heap snapshots improve android taint analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1061–1072, Seoul, South Korea, 2020.

[6] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, page 61–72, San Jose, California, USA, 2006.

[7] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, page 109–126, Nashville, TN, USA, 2008.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, NSDI '05, page 273–286. USENIX Association, 2005.

[9] J. Clause and A. Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, page 515–524, Cape Town, South Africa, 2010.

[10] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 72–82, Montreal, Quebec, Canada, 2019.

[11] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, page 459–470, Florence, Italy, 2015.

[12] M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. Memory and resource leak defects in java projects: An empirical study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 410–411, Gothenburg, Sweden, 2018.

[13] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering*, 25(1):678–718, 2020.

[14] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[15] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 310–323, Long Beach, California, USA, 2005.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, page 125–138, Berkeley, 1992.

[17] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, page 156–164, Boston, MA, USA, 2004.

[18] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 168–181, San Diego, California, USA, 2003.

[19] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[20] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[21] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra. MemInsight: Platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, page 345–356, Bergamo, Italy, 2015.

[22] A. Jindal, P. Staab, J. Cardoso, M. Gerndt, and V. Podolskiy. Online memory leak detection in the cloud-based infrastructures. In *International Conference on Service-Oriented Computing*, ICSOC '20, pages 188–200, Dubai, United Arab Emirates, 2020.

[23] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 31–38, Nice, France, 2007.

[24] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, page 825–836, Hyderabad, India, 2014.

[25] S. Lee, C. Jung, and S. Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on*

Software Engineering, ICSE '14, page 814–824, Hyderabad, India, 2014.

[26] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. Predictive and adaptive failure mitigation to avert production cloud vm interruptions. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[27] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, February 2020.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, Chicago, IL, USA, 2005.

[29] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, oct 2006.

[30] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 559–574. USENIX Association, Feb. 2020.

[31] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 115–124, Washington D.C., USA, 2010.

[32] Microsoft. Windows kernel api: ExAllocatePoolWithTag. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepoolwithtag.

[33] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, ECOOP '2003, pages 351–377, Darmstadt,Germany, 2003.

[34] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, San Diego, California, USA, 2007.

[35] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 397–407, Dublin, Ireland, 2009.

[36] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th International Static Analysis Symposium*, SAS '06, pages 405–424, Korea, 2006.

[37] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, Renton, WA, USA, 2019.

[38] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, page 116–134, Genova, Italy, 1999.

[39] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 194–203, Atlanta, Georgia, USA, 2007.

[40] V. Šor, P. Oü, T. Treier, and S. N. Srirama. Improving statistical approach for memory leak detection using machine learning. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 544–547, Washington, DC, USA, 2013.

[41] V. Šor and S. N. Srirama. A statistical approach for identifying memory leaks in cloud applications. In *Proceedings of First International Conference on Cloud Computing and Services Science*, CLOSER '11, pages 623–628, Noordwijkerhout, Netherlands, 2011.

[42] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *Proceedings of the 2008 USENIX Annual Technical Conference*, USENIX ATC '08, pages 307–320, Boston, MA, USA, 2008.

[43] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 315–326, New Delhi, India, 2010.

[44] J. Vilk and E. D. Berger. BLeak: Automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, page 15–29, Philadelphia, PA, USA, 2018.

[45] C. A. Waldspurger. Memory resource management in vmware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02. USENIX Association, Dec. 2002.

[46] W. Wang. MLEE: Effective detection of memory leaks on early-exit paths in OS kernels. In *Proceedings of the 2021 USENIX Annual Technical Conference*, USENIX ATC '21, pages 31–45, July 2021.

[47] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, page 115–125, Lisbon, Portugal, 2005.

[48] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 270–282, San Jose, California, USA, 2011.

[49] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 151–160, Leipzig, Germany, 2008.

[50] D. Yan, G. Xu, S. Yang, and A. Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 87–97, Orlando, FL, USA, 2014.

[51] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan. *Understanding and Detecting Software Upgrade Failures in Distributed Systems*, page 116–131. Association for Computing Machinery, New York, NY, USA, 2021.