Using Trātṛ * to tame Adversarial Synchronization

Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift

Computer Sciences Department, University of Wisconsin–Madison

Abstract

We show that Linux containers are vulnerable to a new class of attacks – *synchronization attacks* – that exploit kernel synchronization to harm application performance, where an unprivileged attacker can control the duration of kernel critical sections to stall victims running in other containers on the same operating system. Furthermore, a subset of these attacks – *framing attacks* – persistently harm performance by expanding data structures even after the attacker quiesces. We demonstrate three such attacks on the Linux kernel involving the inode cache, the directory cache, and the futex table.

We design Trātṛ, a Linux kernel extension, to detect and mitigate synchronization and framing attacks with low overhead, prevent attacks from worsening, and recover by repairing data structures to their pre-attack state. Using microbenchmarks and real-world workloads, we show that Trātṛ can detect an attack within seconds and recover instantaneously, guaranteeing similar performance to baseline. Our experiments show that Trātṛ can detect simultaneous attacks and mitigate them with minimal overhead.

1 Introduction

Shared infrastructure, where multiple tenants run on the same physical hardware, is common in data centers and cloud computing environments. As anyone can be a tenant, including competitors or malicious actors, such installations place a heavy burden on system software to isolate mutually distrusting tenants. Without strong performance isolation, the behavior of one tenant can harm the performance of other tenants, such as by monopolizing a resource. Past work has focused on isolating CPU usage [17, 23, 30, 68], memory usage [40, 67] storage traffic [2, 36, 60], and network traffic [33, 62] to reduce the effect of sharing the resources.

It is challenging to build a perfectly isolating system platform that is efficient. Virtual machines (VMs) per tenant on a shared physical platform are more efficient, but they share little content in memory and duplicate OS functionality for each VM. Containers, as used by Docker [42] and Kubernetes [55], are highly efficient as they cost little more than an operating system process but rely on a shared operating system kernel between tenants with internally shared data structures.

Our work observes that this high degree of sharing across tenants through operating system data structures creates an avenue for performance interference in two ways. First, to handle concurrent accesses, shared data structures must rely on synchronization mechanisms such as locks or RCU [39]. If one tenant makes heavy use of a data structure, that tenant may monopolize locks and cause victims to stall waiting for access to the structure. We term this a *synchronization attack*. This attack is also possible with other synchronization mechanisms such as RCU.

Second, a tenant can manipulate a shared data structure by adding elements to make all tenants spend more time traversing the structure. For example, suppose an attacker adds thousands of elements to a linked list that other tenants traverse as part of ordinary system calls. In this case, all tenants will experience a substantial slowdown from spending extra time in the list. Worse, tenants will increase contention when they hold locks for long periods to traverse the shared structure. We term this a *framing attack*, as after extending the shared structure, the attacker may stop accessing the structure, but the lock appears to be held by innocent victims.

These new attacks are related to *algorithmic complexity attacks* (ACA) [22] that exploit data structures without strong complexity guarantees [22, 37]. When applied to shared structures protected by synchronization mechanisms, such attacks stall tenants waiting to access the structure or make them run longer, leading to poor performance. While ACAs target preemptable resources, synchronization and framing attacks target non-preemptable resources.

In this paper, we examine synchronization and framing attacks on the Linux kernel when using containers for isolation. We demonstrate several kernel data structures accessed by common system calls – the inode cache and directory cache used by file systems and the futex hash table used for syn-

*Trātṛ (pronounced Tra true) in Sanskrit means guardian or protector.

chronization – are vulnerable to synchronization and framing attacks. Furthermore, we demonstrate how an unprivileged attacker can cause throughput reduction (nearly 3-12x) to real-world applications in a container-based environment.

Based on our experience with these attacks, we observe two conditions common to all the attacks: long critical sections and many kernel object allocations. We develop Trāṭr, a Linux kernel extension to defend against synchronization and framing attacks. As the problem is distributed across kernel data structures, Trāṭr provides a general framework for addressing these attacks using four mechanisms. Trāṭr tracks the contributions to data structure size per tenant to identify an attack, uses two conditions to detect attacks and identify attackers, mitigates the attack by blocking the attacker, and performs data structure specific actions to recover to baseline performance.

Using microbenchmarks and real-world applications, we show the effectiveness, efficiency, and responsiveness of Trāṭr. Trāṭr can detect attacks within seconds of launch, preventing the attack from worsening and recovering performance to baseline (no attack) levels. We conduct a thorough study of overhead incurred by Trāṭr, showing that at steady state, Trāṭr causes only 0-5% overhead for tracking; other mechanisms have negligible cost in the absence of an attack. We conduct a false-positives study to show that for a variety of applications and benchmarks, Trāṭr does not implicate victims as attackers. Lastly, we conduct a false-negative experiment, where a defense-aware attacker can launch an attack without getting detected. We find that randomness in the detection mechanism makes it challenging for the attacker to cause much damage.

The contributions of our work are threefold: (1) We describe two new classes of attacks on shared synchronization primitives – synchronization and framing attacks that lead to large denials of service; (2) We demonstrate Linux kernel vulnerabilities exposing tenants to synchronization and framing attacks from other unprivileged tenants; and (3) We describe Trāṭr that defends against these attacks with low-overhead tracking and detection mechanisms for in-progress attacks, and prevention and recovery mechanisms for restoring performance to pre-attack levels.

2 Synchronization under attack

In this section, we discuss how shared infrastructure in data centers rely on shared data structures protected by a variety of synchronization mechanisms, and how exploiting these synchronization mechanisms can lead to denial-of-service attacks.

2.1 Concurrent Shared Infrastructure

Shared infrastructure is common in the data center comprising the CPU, memory, disk, and network. System software such as a virtual machine monitor or operating system allows multiple tenants to concurrently share the hardware creating *concurrent shared infrastructure*. The most common tenant environments for shared infrastructure are virtual machines (VMs) and containers. With VMs, each tenant runs their operating system over virtual hardware resources provided by a virtual machine monitor, which space- or time-shares physical resources across virtual machines.

Container isolation is based on a combination of mechanisms. Containers rely on schedulers to fairly share preemptable resources such as CPU, disk, or network between containers. For memory, accounting and allocation limits prevent containers from overusing memory. The operating system provides private namespaces for each container that prevents them from accessing resources of other containers such as private file system directory trees and private sets of process IDs. An operating system kernel provides virtual software resources (files, sockets, processes) to each container. Substantial effort has gone into isolation so that one container or VM has minimal performance impact on others [23, 30, 33, 36, 40, 60, 67] and each container or VM obtains a fair share of the resources. However, these isolation controls are built atop shared kernel data structures; in many cases, the kernel maintains global data structures shared by all containers and relies on scheduling, accounting, and namespaces to prevent interference. Our work focuses on container-based isolation, as its higher-level interfaces create more opportunities for performance interference.

2.2 Synchronization and Framing Attacks

Container isolation mechanisms do not directly isolate accesses to the operating system's global data structures. Operating system kernels contain hundreds of data structures global to the kernel and shared across containers. These structures rely on synchronization primitives such as mutual exclusion locks, read copy update (RCU), and reader-writer locks to allow concurrent access. Multiple containers make unprivileged system calls to access the same kernel data structures using these synchronization primitives in a shared environment. We focus on mutual exclusion locks and RCU as they are heavily used in the kernel.

Synchronization primitives do not control how long one tenant can spend in a critical section accessing a data structure. Locks are mutually exclusive such that once held, they prevent any other process trying to acquire the lock from making progress. Likewise, RCU allows multiple readers to access the data structure, but updaters wanting to free objects must wait until all prior read critical sections complete [39]. We call the time spent waiting to acquire a lock or to let all the prior read critical sections complete *synchronization stalls*.

Consider a linked list in Listing 1 that supports `insertion` (insert()) and `search` (find()) operations. An attacker can cause lock contention by repeatedly accessing the list. If the list is short, the synchronization stalls will not be long, but if

```
struct node {
  int data;
  struct node *next;
};

void insert(struct node **list, struct node *n) {
  lock();
  n->next = *list; *list = n;
  unlock();
}

struct node *find(struct node **list, int data) {
  lock();
  struct node *n = *list;
  while (n) {
    if (n->data == data) {
      unlock();
      return n;
    }
    n = n->next;
  }
  unlock();
  return NULL;
}
```

Listing 1: Simple linked list example

an attacker can vastly expand the list, then the time spent in search operations will increase, and victims may stall waiting to access the list. We term this a *synchronization attack*, in which an attacker increases the critical section size to deny victims access to one or more shared data structures. Such an attack occurs when:

- Condition $S1$: A shared kernel data structure is protected by a synchronization primitive that can block such as a mutual exclusion lock or RCU.
- Condition $S2$: Unprivileged code can control the duration of the critical section by either
  - $S2_{input}$: providing inputs that cause more work to happen within the critical section
    OR
  - $S2_{weak}$: accessing a shared kernel data structure with weak complexity guarantees e.g., linear.
    AND
  - $S2_{expand}$: expanding the shared kernel data structure to trigger the worst-case performance.

We term the case when an attacker targets a synchronization primitive (condition $S1$) and uses input parameters (condition $S2_{input}$) to increase critical section size an *input parameter attack*. One known example of an input parameter attack occurs when a rename operation is performed on a large directory, holding a shared per-filesystem lock while traversing the entire directory [48]. We also found that AppArmor [1] holds a shared namespace root lock while loading profiles, so loading a large profile can hold the lock for tens of seconds. Existing solutions can address input parameter attacks by ensuring lock usage fairness (fixing condition $S1$) with Scheduler Cooperative Locks (SCLs) [48] or by using regression-based analysis [34] (breaking condition $S2_{input}$). Given these solutions, input parameter attacks are not the focus of this paper.

In this paper, we focus on the more challenging synchronization attacks that exercise conditions $S2_{weak}$ and $S2_{expand}$. For the linked list example, the lock protecting the critical section meets condition $S1$, the list exhibits weak properties meeting condition $S2_{weak}$, and elongating the list meets condition $S2_{expand}$. Even if RCU replaces the lock, the expanded list leads to a lengthy read-side critical section, stalling the victims who want to delete from the list.

Synchronization attacks are *active attacks* if the attacker itself executes the long critical section. However, in some cases, the attack can continue without the further participation of the attacker. For example, consider what can happen if other tenants traverse the elongated list. After an attacker adds millions of entries to the list, other processes will continue to traverse the longer list, leading to more time traversing the list and more time stalling on the lock.

We term this a *framing attack* because an inspection of who holds the lock will incorrectly frame innocent victim threads rather than identifying the attacker that expanded the data structure. Like a criminal framing someone innocent for a crime, this attack directs blame at other victims. This is a *passive attack*, as the attacker needs to do nothing to continue the performance degradation. More precisely, a framing attack is an extension of a synchronization attack and occurs when:

- Condition $S1 + S2_{weak} + S2_{expand}$: An attacker expands a shared kernel data structure with weak complexity guarantees, i.e., a synchronization attack is in progress or was launched earlier.
- Condition $F1$: Victim tenants access the affected portion of the shared data structure with worst-case behavior.

In framing attacks, for mutual exclusion locks, the excessive stalls are attributed to other victims traversing the list rather than the attacker that grew the list. RCU relies on the *grace period* to ensure that existing readers finish their access before a delete operation starts. For expanded data structures, the longer read-side critical section leads to a longer grace period impacting performance. Thus, the victims continue to observe poor performance due to the past actions of the attacker.

Synchronization attacks make the victims stall longer; framing attacks additionally make them spend more time in the critical section. Framing and synchronization attacks can happen at the same time. Consider a situation where a hash table uses the protected list to build hash buckets. The attacker may target a single hash bucket by adding many entries leading to a synchronization attack on that bucket. The victims will have to wait longer to acquire the lock. If one of these starved victims access the target hash bucket, they will traverse the elongated list and hold the lock longer, leading to a framing attack. Addressing framing attacks requires additional steps to repair the shared data structures even after the attacker stops executing to ensure condition $F1$ is not met. Merely preventing the continuation of an attack does not stop victims from accessing the expanded data structure.

Algorithmic Complexity Attacks vs. Adversarial Synchronization. Even though adversarial synchronization looks similar to algorithmic complexity attacks, they are fundamentally different. While the algorithmic complexity attacks target preemptable resources, synchronization and framing attacks target non-preemptable resources like mutual exclusion locks.

There have been numerous algorithmic complexity attacks that end up exhausting one or more CPUs in the system [4–12, 61]. As the CPUs are exhausted, they cannot execute the regular user workload leading to denial-of-services. As container isolation guarantees proper isolation of preemptable resources, such attacks may not impact all the containers running on the host.

On the other hand, synchronization attacks make victims stall longer, and framing attacks stall the victims and make them execute longer. As these attacks target shared synchronization primitives, more than one container that needs to access the shared synchronization primitive and the kernel services are impacted, leading to poor performance. Existing container isolation mechanisms do not treat synchronization as a resource and hence cannot handle the monopolization of the shared synchronization primitives.

3 Real-World Problems

In this section, we present the threat model and show how locks and RCU can turn adversarial in the Linux kernel.

3.1 Threat Model

We assume the following about the adversary and environment. One or more containers run on a single physical machine. All containers, including the one that plays the role of an adversary, hereafter called an attacker, run arbitrary workloads that can access OS services via system calls. We assume there is a 1-1 mapping between tenants to users, and each container is associated with a user. No container, including the attacker, has special privileges. Due to random cloud scheduling, we assume a single attacker, thereby removing the possibility of collusion. We place no limit on the number of containers a single user can run on a single physical machine.

The attacker targets one or more synchronization primitive in an operating system making other containers accessing the same primitives starve or waste CPU time, leading to poor performance or denial-of-service. The attacker can use either a single container or multiple containers to launch an attack.

3.2 Synchronization and Framing Attacks on Linux kernel

We describe three Linux kernel data structures that are vulnerable to Algorithmic Complexity Attacks (ACAs) and can be used to launch synchronization and framing attacks. The setup is the same as used in Section 5.

Synchronization attack on inode cache. The Virtual File System maintains the inode cache to avoid expensive disk accesses to read file metadata [14]. A global lock `inode_hash_lock` protects the inode cache (meets $S1$). The inode cache is implemented as a hash table meeting $S2_{weak}$ as collisions in a hash bucket are handled with a linked list of inodes with the same hash value. The number of buckets in the hash table is decided at boot time based on memory size.[1]

The inode cache hash function combines the inode number, unique to each file, and the address of the file system superblock data structure in memory. This address is set when a volume is mounted but varies across systems and boots. While the inode number for a file is visible to unprivileged users, the superblock address is not, and without that address, it is hard to predict which hash bucket an inode will reside in.

We have found a way to break this function, which we describe in detail in the Appendix. By creating files with specific inode numbers, a user can probe for the superblock address, allowing them to create files in a single hash bucket that grows and is slow to traverse. Although users cannot generally specify the inode number for a file, this is possible with a FUSE unprivileged file system in user-space [66]. For Docker, mounting needs CAP_SYS_ADMIN, which is privileged [32]. Linux supports unprivileged FUSE mounts [35], although Docker disables this by default. [2] As a workaround, we use the idea of Linux user namespaces [43] discussed by NetFlix [27] and elsewhere[2] to mount the FUSE file system in an unprivileged environment.

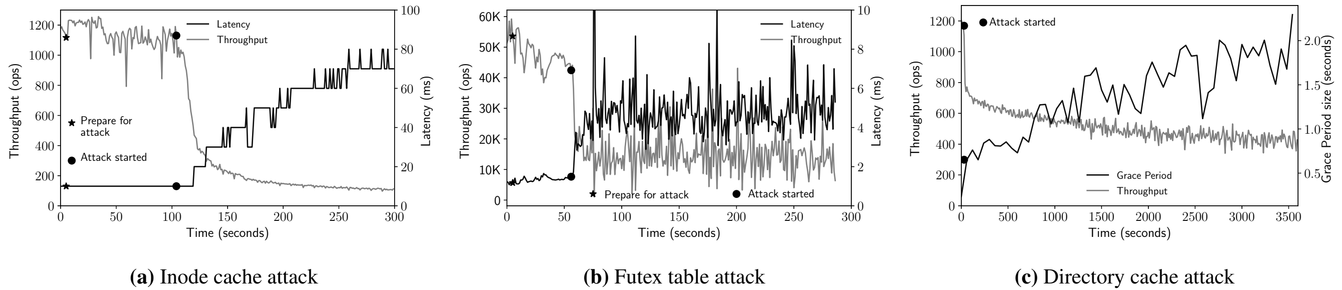
After mounting the FUSE filesystem, a user can create files with arbitrary inode numbers and create collisions in the inode hash, leading to long lists in some hash bucket (meets $S2_{expand}$). Because of the large number of hash buckets, it is difficult for the attacker to target a specific file for contention. Instead, the attacker continues to access the same bucket, elongating critical sections.

To show the impact on the victim's performance, we run an Exim mail server container as a victim and launch an inode cache attack from a separate container. We run MOSBENCH [16] scripts as the client from another machine to send messages to the Exim server.

Figure 1a shows the timeline of the throughput and average latency for the duration of the attack. Once the attack starts, the performance reduces significantly. The attacker initiates probing the inode cache to determine the superblock address. Around 100 seconds, the attacker finds the superblock address and then targets a random hash bucket. The lock is held while adding entries to this bucket, starving the Exim mail server and reducing its throughput by 92% (12x). The attacker

[1] For a system having 128 GB DRAM, the inode cache has $2^{22} = 4,194,304$ hash buckets.

[2] A bug is already filed to allow FUSE functionality by default - https://github.com/docker/for-linux/issues/321.



(a) Inode cache attack

(b) Futex table attack

(c) Directory cache attack

Figure 1: Performance of applications under attack. *(a) Throughput and Average Latency timeline of Exim Mail Server when under inode cache attack. (b) Throughput and Average Latency timeline of UpScaleDB when under futex table attack. (c) Throughput and grace period timeline of Exim Mail Server when under directory cache attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing and identify a target hash bucket. Once a target hash bucket is identified, the attack is launched.*

continues to add more entries to the hash bucket, increasing the lock hold time further. In comparison, when we run two other applications – DBENCH and UpScaleDB instead of the attacker to generate interference, we observe a 15% reduction in the performance.

Moreover, an economic impact is also associated as the victims spin while waiting to acquire the lock. We observe that around 33% of the total CPU used by the victim threads is spent on waiting to acquire the lock. Given enough resources and time, an attacker can further increase the wait times by adding more threads leading to even worse performance.

Framing attack on futex table. The Linux kernel supports futexes, a light-weight method to support thread synchronization in user-space [28]. A futex provides the ability to wait on a *futex variable*, which is any location in memory until another thread signals the thread to wake up. Futexes are used to build synchronization abstractions such as POSIX mutexes and condition variables. The `futex()` syscall lets the user-space code wait and signal futex variables.

Rather than maintain a wait queue for each futex variable, the kernel maintains a *futex table*, and each bucket in the table is a shared wait queue (meets $S2_{weak}$). The kernel hashes the futex variable address to identify the wait queue for a futex variable. When a thread waits on a futex, the kernel adds the thread to the wait queue dictated by the hash of the futex variable. Similarly, when waking a thread, the kernel must walk the shared wait queue looking for multiple threads waiting on that futex variable. As a result, several futex variables belonging to the same or different applications can share a single wait queue. A separate lock protects each hash bucket (meets $S1$). The number of buckets is decided at boot time and is a multiple of the number of CPUs in the system.[3]

As the number of hash buckets is small, the attacker uses bucket probing to identify a target hash bucket instead of breaking the hash function. The attack starts by allocating a few thousand futex variables to map them to different wait queues. The attacker then probes the wait queues by calling futex() to wake a thread for each variable while measuring the time it takes to complete the syscall. The syscall will take measurably longer to complete if victim processes are already using a wait queue, allowing the attacker to attack these wait queues.

After identifying a busy-wait queue, the attacker spawns thousands of threads that wait on the target futex variable, thereby expanding the wait queue (meets $S2_{expand}$). Upon expansion, any victims sharing the queue must walk the elongated queue to wake up their threads, leading to longer lock hold times, longer stalls, and poor performance.

We conduct an experiment by running UpscaleDB, an embedded key-value database [65], within a container as a victim to show the performance impact. We use the built-in benchmarking tool ups_bench to run an in-memory insert-only workload. Figure 1b shows the throughput and average latency timeline. Before the attack starts, UpscaleDB observes high throughput while the average latency remains constant. During the first part of the attack, the attacker probes the futex table, and around time 54 seconds, identify a busy-wait queue and starts creating threads to lengthen the queue. This leads to highly variable performance for UpScaleDB, reducing throughput between 65 to 80% (3x-5x). We also observe that the tail latency increases from around 10-15 milliseconds to 0.7-1.2 seconds, an increase of 45x to 100x. We observe a 10% reduction in the performance if we run DBENCH and Exim mail server instead of the attacker.

Unlike the inode cache attack, in this scenario, the attacker becomes passive and sits idle after creating the waiting threads, which demonstrates a framing attack – there is lock contention, but the attacker is not actively acquiring the lock. When the victim access the target hash bucket, the condition $F1$ is met. From an economic impact perspective, the victim spends around 40% of the total CPU time waiting to acquire the lock. Moreover, as the victim is forced to traverse an expanded list, we observe that the victim's total CPU usage increases by 2.3x times compared to baseline and may end up paying more for the extra CPU usage.

Synchronization attack on Directory cache. Lastly, we show a vulnerability that can be exploited by an attacker that can break the dcache hash function. The Linux directory

[3] For a 32 CPU system, the hash table comprises $256 * 32 = 8,192$ hash buckets.

cache (dcache) stores dentry structures to support filename lookups [64]. The dcache is implemented as a hash table where each bucket stores a linked list of dentries with the same hash value. The hash function uses the parent dentry address and the filename to calculate the hash value.

For efficiency, the dcache relies on RCU to allow concurrent read access, but freeing entries must wait for all concurrent readers to leave the read critical section. This wait, called a *grace period*, ensures that no reader is holding a reference to the deleted object. RCU provides synchronous (`synchronize_rcu()`) or asynchronous (`call_rcu()`) APIs for this purpose. While the synchronous API makes the user wait until the grace period ends, the asynchronous API registers a call back that the RCU subsystem executes after the grace period is over. As RCU is shared across the Linux kernel, any increase in the grace period stalls the victims.

The attack exploits the dcache's support for *negative entries*. These entries record that no such file exists. By breaking the hash function, an attacker can create millions of negative entries mapping to a single hash bucket, thereby meeting condition $S1 + S2_{weak} + S2_{expand}$. Before creating a negative entry, the lookup operation first walks through the hash bucket to check if the entry exists or not. The hash bucket walk is part of the RCU read-side critical section. Walking an expanded hash bucket increases the read-side critical section, thereby increasing the grace period size too. Victims using the `synchronize_rcu()` will stall until the grace period is over. In the case of `call_rcu()`, freeing objects will be delayed, and more work will pile up for the RCU background thread to execute the callbacks leading to lower performance or out of memory conditions [38, 51, 52, 58].

To demonstrate the attack and the impact on the victim's performance, we run an Exim mail server container as a victim and launch the dcache attack from a separate container. We modify the kernel to simulate an attacker targeting any hash bucket. Figure 1c shows the throughput (averaged over 10 seconds) for the duration of the attack. Once the attack starts, as the hash bucket size increases, the read-critical section size increases, increasing the grace period size. Towards the end of the experiment, the performance drops more than 90% (10x) for a few instances. The grace period size increases from 20-30 milliseconds to 2 seconds. The mail server generates hundreds of thousands of callbacks every second overwhelming the RCU background thread.

An attacker can launch the same attack without breaking the hash function by randomly creating hundreds of millions of negative entries instead of targeting a single hash bucket.

Existing Solutions. Attacks on synchronization primitives can be addressed by interrupting one of the criteria necessary for an attack by using lock-free data structures; or using universal hashing, balanced trees or randomized data structures [22] to break condition $S2_{weak}$ and $S2_{expand}$. However, randomized data structures are vulnerable to ACAs [15] and rewriting the kernel to use balanced trees is tedious [20, 49].

Relying on strong hash functions is not enough as an attacker can launch attacks without breaking the hash function. Moreover, it is not easy to convince developers to use secure hash functions such as SipHash due to performance concerns [21]. We observe around 5-6% performance reduction when we replace the existing hash function in the inode cache with SipHash while running a simple file create workload confirming developer concerns. Another approach of rehashing all the entries into a new hash table is possible but is invasive to the code and may cause long delays during rehashing.

SCLs can prevent lock usage dominance during a synchronization attack by guaranteeing lock usage fairness. However, they fail to handle the framing attacks as they are not aware of the cause of the longer lock hold times; they may treat the victims like the one dominating the lock usage and penalize the victims instead of the attacker. More details about SCLs performance can be found elsewhere [46].

Summary. In all these three attacks, the common piece is that the attacker can run arbitrary code to target the synchronization primitives. Using containers is one way to launch attacks by executing any user workload, especially as the container isolation techniques do not directly isolate accesses to the shared layers such as kernel, thereby becoming an easy target for such attacks. Other environments, such as multiple containers running within a single virtual machine, multiple virtual machines running on a shared hypervisor, etc., are vulnerable to such attacks. However, further investigation is needed to confirm if an attacker can target such environments.

Through the inode cache attack, we show how easy it is to obtain the superblock pointer address and break KASLR [25]. Attackers may employ numerous existing methods to break KASLR and extract the addresses needed to break the hash function and launch attacks. Thus, it is important to acknowledge that while we demonstrated the problem on three data structures only, the problem may be widespread as there are hundreds of kernel data structures that may meet the conditions for the attack. We manually analyzed 5429 critical sections protected by 617 locks, a small subsection of the total critical sections in the Linux kernel. We find that 1039 contain loops (19%) and 112 instances (2%) that call `synchronize_rcu()` in the critical section respectively that an attacker can potentially exploit. We briefly describe the manual analysis process in the Appendix.

Therefore, we take a multi-pronged approach to addressing these attacks. We seek (1) light-weight mechanisms to detect an in-progress attack, followed by (2) a combination of prevention strategies for active attacks to block a malicious actor from continuing an attack, and (3) recovery strategies that seek to restore the data structure to its normal access cost.

4 Trāṭṛ

Trāṭṛ is an extension to the Linux kernel that provides a framework to detect and mitigate synchronization and fram-

ing attacks. We present the goals for our design and then an overview of Trāt̤r design followed by the implementation of Trāt̤r with two recovery solutions.

4.1 Goals

We have four high-level goals to guide our design:

Automatic response and recovery. We seek an automated response to synchronization and framing attacks to reduce administrator effort and prevent them from harming performance. While preventing an attacker from continuing may be sufficient for synchronization attacks, framing attacks require recovery to restore data structure performance properties.

Low false positives and negatives. Due to automatic response, we want to reduce false positives and negatives. More so, as there is a thin line between heavy resource usage and denial-of-service, we seek detection mechanisms relying on multiple signals to avoid false positives and negatives.

Easy/flexible to support multiple data structures. Data structures may require specialized recovery solutions, so a single generic solution is not possible. Hence, it should be easy for developers to incrementally add protection to targeted data structures as attacks are identified.

Minimal changes to kernel design and data structures. Much effort has been put into selecting and designing kernel data structures [45]. We want to avoid extensive changes to the kernel or modifications to hash functions that could lead to performance issues.

4.2 Overview

Trāt̤r is a Linux kernel extension to defend against synchronization and framing attacks. The first step in using Trāt̤r is to identify vulnerable data structures that may be used in attacks. We performed this task manually, but it could be determined using static analysis or as part of an attack postmortem. After finding such a data structure, our general approach is to *track* resource usage in the steady-state and *detect* anomalous resource usage as a sign of attack. On detecting an attack, Trāt̤r acts to *prevent* the attack from continuing and to *recover* from the effects of the attack.

Two *detection conditions* are common to all three attacks:

- *Condition LCS: Long critical section.* Expansion of the data structure causes more work for the attacker or victims or both, making the critical section longer.
- *Condition HSUA: High single-user allocations.* A single user has created many entries associated with the data structure.

Neither condition on its own is sufficient to indicate an attack as there may be other reasons for high allocations or long critical section times (e.g., interrupt handling). Hence, Trāt̤r first checks *critical section size* and if it's too large, then checks if one user has a majority of the *object allocations*.

On detecting an attack, Trāt̤r quickly prevents the attack from worsening by stopping the attacker from extending the data structures for a period. However, without recovery, the attacker can still access the expanded data structure and continue with the synchronization attack, or the victim can access the expanded data structure leading to framing attack. Therefore, the final step is recovery, where Trāt̤r tries to repair the data structure to restore its performance to pre-attack levels. Trāt̤r relies on the type and purpose of the data structure to find an appropriate recovery mechanism.

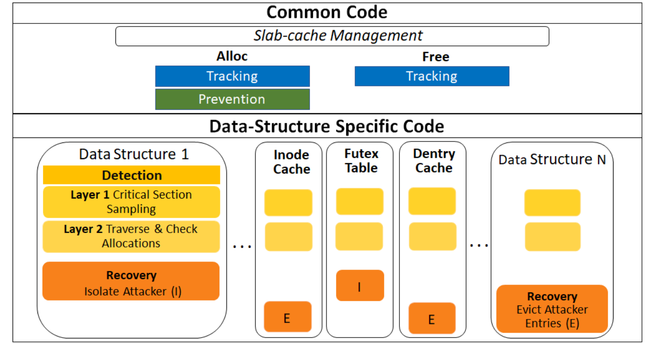


Figure 2: High-level design of Trāt̤r. *Design showing the four mechanisms of Trāt̤r. The Tracking and Prevention mechanisms are part of slab-cache management. Layer 1 Detection measures the synchronization stalls to indirectly measure long critical sections. On finding longer stalls, Trāt̤r triggers layer two checks if a user has a majority of object allocations. On finding one, Trāt̤r identifies that user as an attacker and initiates prevention and recovery mechanisms. The prevention mechanism prevents the attacker from allocating more entries. Depending on the type of data structure, an appropriate recovery is initiated. The upper box shows the common code, while the lower box shows data structure-specific code.*

4.3 Design & Implementation

A high-level design of Trāt̤r is shown in Figure 2. We start with the design for two data structures, the inode cache and futex table, and add the directory cache later to explain the steps for adding a new data structure to Trāt̤r. We implement Trāt̤r in Linux kernel version 5.4.62.

Tracking: The purpose of tracking is to support detection and recovery. Trāt̤r records the allocation and deallocation of objects associated with a vulnerable data structure. When allocating an object, Trāt̤r (i) tracks the total number of objects currently allocated by the user and (ii) stores the user ID in the allocated object. The number of objects allocated by each user helps identify the HSUA condition. The user ID information is used by recovery to identify the objects allocated by the attacker. Trāt̤r increases the object size by 4 bytes to attach the user ID at the end of the object.

Instead of writing a separate tracking mechanism for each object, we use the Linux kernel slab-cache infrastructure to track the objects. Linux associates a slab-cache with a data structure to manage object allocation and freeing [29, 53]. We modify the common slab-cache management code to selectively record the relevant information for specific slab caches. The current implementation allows for tracking the

Data Structure	Tracking		Detection	Recovery
	Slab-cache	Object	Primitives sampled	
Inode cache	ext4_inode_cachep fuse_inode_cachep	ext4_inode_info fuse_inode	inode_hash_lock	Evict
Futex table	task_struct_cachep	task_struct	futex_hash_bucket.lock	Isolate
Dentry cache	dentry_cache	dentry	RCU	Evict

Table 1: Implementation summary of Trātṛ.

objects allocated using kmalloc() and kmem_cache_alloc() and cannot track the allocations done via the vmalloc() and get_free_pages() APIs. The total objects allocated per user for each slab-cache is stored in a NUMA-aware global hash table, updated during allocation and free operations. For our prototype, Trātṛ tracks four objects associated with the inode cache, the futex table, and the directory cache, which is summarized in Table 1.

Tracking is on the critical path for object allocation and deallocation and does increase memory consumption for the user ID and the global hash table; as we show later, the performance and memory overhead is minimal. Although accounting can be performed across any entities that correspond to a possible attacker (e.g., threads, processes, users, or containers), we currently focus on per-user accounting.

We believe that Trātṛ is not a new avenue for launching adversarial synchronization attacks. Trātṛ only uses the global hash table to track object information per user. We assume that creating millions of users is not an easy task. Thus, an attacker cannot target Trātṛ to launch synchronization and framing attacks on the global hash table. Furthermore, enabling container-level tracking instead of user-level tracking will make it impossible to create millions of containers on a single node even if an attacker succeeds in collocating containers on the same node.

Detection: The primary objective of detection is to check whether the LCS and HSUA conditions are met. When met, Trātṛ detects an attack and initiates prevention and recovery. For efficiency, Trātṛ adopts a two-layered approach. The first layer checks for LCS, while the second layer checks for HSUA and identifies the attacker. For simplicity purposes, a separate kernel thread handles the layered checks and performs recovery for each slab cache. It is possible to merge the layered checks and recovery procedure for multiple slab caches for efficiency purposes. The flowchart of the kernel thread is shown in Figure 3.

To avoid large overheads, Trātṛ does not directly measure the critical section length . Instead, Trātṛ samples the synchronization primitives to measure synchronization stalls, since if a synchronization primitive is under attack, the stalls will be longer than expected. We call this check *Critical Section Sampling (CSS)*. Trātṛ relies on threshold values to detect if the wait time is above expectation; this threshold is determined by assuming a uniform distribution of objects in the hash table, and all CPUs participate in lock acquisition. By running workloads that access inode cache and futex table, we observe that worst-case lock hold times are around 2-3 μs. As our experiments are run on a machine that has 32 CPUs,

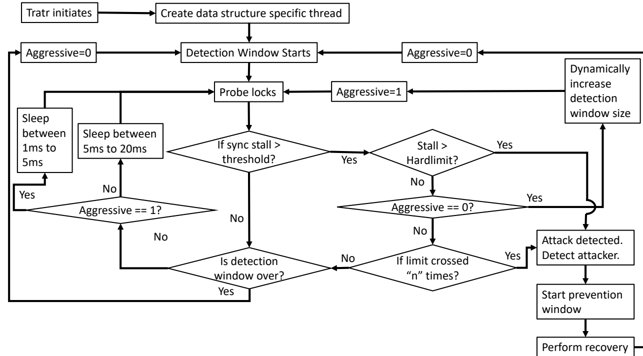


Figure 3: Flowchart of the kernel thread associated with a data structure. *The kernel thread that is associated with a data structure performs the detection and recovery mechanism. As part of the detection mechanism, the thread probes the synchronization primitives. If a synchronization stall is more than the threshold, appropriate action is initiated. Upon detecting an attack, the thread executes the TCA check to identify the attacker and initiate prevention window. Lastly, the thread initiates the recovery of the data structure.*

under heavy lock contention, the worst-case wait time will be roughly 64-96 μs as Linux uses queued spinlocks to protect the inode cache and futex table. Therefore, the current implementation uses a threshold of 100 μs. Table 1 shows the locks that Trātṛ samples to detect inode cache and futex table attacks. The sampling for RCU used in the dentry cache will be described later in Section 5.5.

The sampling process is shown in Figure 4. To avoid false positives, Trātṛ must repeatedly observe stall times above the threshold. Trātṛ is adaptive and randomized to handle defense-aware attackers. As shown, Trātṛ creates a *sampling window* within which a lock is repeatedly sampled; if the stall threshold is exceeded multiple times within this window, the CSS check finishes, and Trātṛ proceeds to the TCA check (discussed later). Once a sampling window ends, another window starts immediately. The length of the sampling window is randomly chosen and ranges between 1 and 5.3 seconds.

Within each window, Trātṛ introduces a random delay between two samples to incur low interference with normal user operations. The *sampling delay* begins between 5-20 ms. Trātṛ enters suspicious mode on measuring a stall above the threshold. When suspicious, Trātṛ increases the sampling window size and aggressively samples the lock every 1-5 ms to quickly detect an attack. If no attack is found in the elongated window, Trātṛ returns to normal mode. To ensure attackers cannot slowly expand a data structure, Trātṛ also has a hard limit for the length of a synchronization stall, beyond which an attack is detected immediately. This check ensures that the given lock cannot be held for an extremely long duration.

As the sampling window size and the sampling delay is randomized, Trātṛ makes it difficult for a defense-aware attacker to launch an attack. A defense-aware attacker cannot know when to launch an attack and stop to remain undetected. To remain undetected, the attacker would have to ensure it never holds the lock beyond the threshold and does not repeat it multiple times within the same sampling period. This sig-

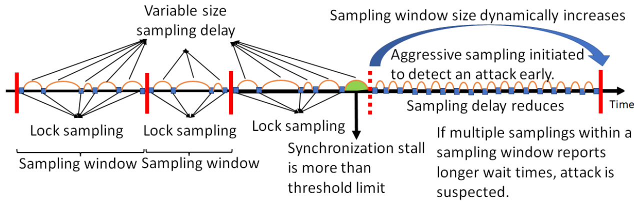


Figure 4: Working of Critical Section Sampling. *Trātṛ samples for longer critical section multiple times within a sampling window. Between two samplings, there is a sampling delay when the kernel thread sleeps. When suspicious, Trātṛ increases the sampling window's size, reduces the sampling delay, and aggressively starts sampling to detect an attack early.*

nificantly slows down the attacker and makes the attack less feasible.

If Trātṛ detects multiple long stalls, it initiates the second layer, *Traverse & Check Allocations (TCA)*. In TCA, Trātṛ traverses the data structure associated with the synchronization primitive and uses the 4-byte user ID embedded in each object to count each user's total number of allocations. If a particular user has allocated the majority of entries, Trātṛ tags that user as an attacker and passes their identity to the prevention and recovery mechanisms. Tagging the users holding the lock longer is insufficient to determine the attacker, as it may end up tagging the victim of a framing attack. Thus, the embedded user ID helps Trātṛ to identify the attacker. For the inode cache and futex table, Trātṛ selects the bucket with the most entries for traversal.

Even though Trātṛ tries to avoid false positives, poorly configured applications and stress testing [13] may cause incorrect detection. To avoid an excessive impact on innocent victims, Trātṛ takes a softer but adaptive approach for prevention.

Prevention: We identify two approaches to mitigating attacks. First, the system could terminate or suspend an attacker's container, stopping condition $S2_{expand}$; for some data structures, killing the container could trigger clean-up, stopping condition $F1$ as well. However, we believe killing the container is inappropriate as it may lead to application-level corruption when a user is wrongly identified as an attacker. Suspending the attacker is also inappropriate as the attacker may hold locks that could impact victims. Therefore, we investigate the second approach: attackers are rate limited by stalling them when they try to allocate memory to expand the vulnerable data structure (stopping condition $S2_{expand}$).

The prevention mechanism uses the existing slab-cache infrastructure to prevent the attacker from expanding the attack or launching future attacks. After identifying the attacker, Trātṛ blocks the attacker from allocating more objects from the slab cache for a specific period, called the *prevention window*. Threads from the attacker trying to allocate objects are put to sleep until the window expires. For the inode cache, this means attackers cannot create new in-memory inodes, blocking them from opening/creating files. For the futex table, this means attackers cannot create new threads. Note that Trātṛ does allow allocation requests with the ATOMIC flag, which denotes that the process cannot be blocked; however, this flag is rarely used with vulnerable kernel data structures.

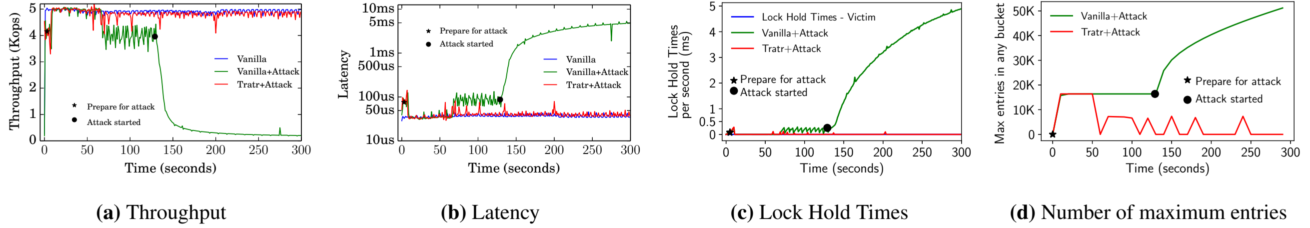
The length of each user's prevention window increases if it performs more attacks. Trātṛ maintains a separate prevention window size for each user. Trātṛ initializes the window size to 1 second and then increases its size depending on how frequently Trātṛ detects the user as an attacker. For an attacker that is continuously trying to launch an attack, the growth factor will be high as the attacks will be frequent. On the other hand, if a victim is wrongly identified as an attacker a few times, the prevention window will stay small, and the victim will be able to resume its work.

Recovery: With only prevention, victims may continue to observe poor performance as the expanded data structure still exists, which the attacker or victim can continue to access. Therefore, recovery is necessary to restore the performance to normal. To design recovery solutions that can support different data structures, Trātṛ offers two solutions.

One solution deals with cache-like data structures where the presence or absence of an entry does not impact correctness. For such caches like the inode cache, Trātṛ evicts all the entries belonging to the attacker. Victims do not lose much performance from the eviction of the attacker's entries, as they typically do not reference those entries. This approach breaks condition $F1$ as victims no longer have to traverse the attacker's entries. Implementing eviction for inode cache is straightforward as we reuse existing code. Trātṛ iterates through the file systems (fuse and ext4 currently) in use to enumerate all inodes and drop those allocated by the attacker.

The other solution deals with non-cache data structures where correctness is required. For the futex table, threads must be present on the waitlist to correctly implement synchronization. As each entry in the waitlist is a waiting thread, dropping any entry may leave the threads waiting, leading to problems. With these data structures, a shared structure is often used as a convenient mechanism to manage data from all processes but is only accessed by a single container. For the futex table, each process only accesses entries belonging to its futex variables despite sharing the wait queues.

Partitioning is a traditional approach used to design data structures to ensure isolation [47]. For such data structures, Trātṛ *partitions the entries* so that victims and attackers use separate, parallel structures, and victims do not have to traverse the attacker's entries. This isolation breaks condition $F1$. Trātṛ walks the data structure, identifies entries allocated by the attacker, and moves those entries out of the primary structure to a new *shadow structure*. On subsequent access, victims only access the original primary structure, while attackers only access the shadow structure. This ensures victims are not delayed by the number of attacker entries. Trātṛ dissolves the shadow bucket once the prevention window ends. This approach does not support futexes shared across users, but we believe this is extremely rare for containers across different users. We note that partitioning may not work with



(a) Throughput **(b)** Latency **(c)** Lock Hold Times **(d)** Number of maximum entries

Figure 5: IC benchmark performance without attack, with attack and with Trātṛ. *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel. With Trātṛ, the attacker is not able to launch an attack. (b) Timeline of the average latency showing the time to create the file. (c) Timeline of the lock hold times of the victim for Vanilla and Trātṛ kernel. (d) Timeline of the maximum number of entries of the attacker in any bucket for Vanilla and Trātṛ kernel.*

cache-like data structures, as it could create multiple copies of entries allocated by both victim and attacker and lead to inconsistencies.

While Trātṛ is performing recovery, an attacker can continue to access the expanded data structure, thereby slowing recovery. Trātṛ solves this by blocking the attacker from acquiring the synchronization primitive until recovery completes. For nested synchronization, Trātṛ needs to prevent acquiring the highest level primitive.

Adding a new data structure. Analyzing all existing data structures to check for vulnerabilities is challenging. Therefore, adding more structures to Trātṛ incrementally on finding a new vulnerability is important. To add a new data structure, a developer must pass a flag to the memory management system when creating the associated slab-cache at boot time. Additionally, the developer must implement CSS checks, TCA checks, and recovery functions for the new data structure. In Section 5.5, using the example of dentry cache, we show the effort needed to add a new data structure. A more thorough Linux kernel analysis is needed to identify and protect vulnerable data structures.

5 Evaluation

Using microbenchmarks, we show the effectiveness of all the four mechanisms, performance characteristics, and responsiveness of Trātṛ. We conduct a thorough study of the performance and memory overhead introduced by Trātṛ. We show how Trātṛ can handle multiple real-world applications and simultaneous attacks. Lastly, we conduct a study of false positives and false negatives to demonstrate the robustness of Trātṛ.

We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3 with two sockets; each socket has eight cores with hyper-threading. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 20.04 kernel version 5.4.62. All applications and benchmarking tools are run as separate Docker containers. We run each experiment three times and report the average value along with 95% confidence interval unless stated otherwise. We use the average value for the plots.

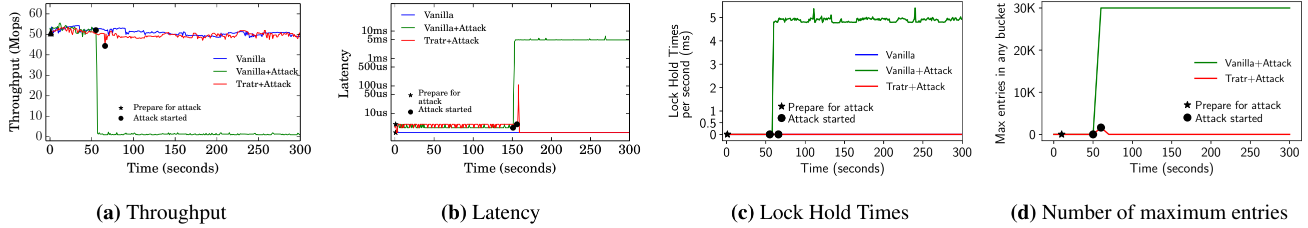
In our experiments, the standard Linux kernel is labeled *Vanilla*. The kernel with Trātṛ and all four mechanisms enabled is *Trātṛ*. *Trātṛ-T* denotes that just tracking is enabled; *Trātṛ-TDP* denotes tracking, detection, and prevention are enabled. The suffix *+Attack* notes that a kernel is under attack.

5.1 Performance under attack

We begin by showing that Trātṛ can effectively reduce the attack's impact on synchronization primitives compared to the Vanilla kernel. Next, we quantify the performance impact of an inode cache attack and a futex table attack on two different victims that are particularly sensitive to those attacks. Each experiment is run for 300 seconds with 8 CPUs and 8 GB of memory for the victim and attacker containers.

The *IC benchmark* contains a victim which is sensitive to the inode cache because it creates an empty file every 100 microseconds. The IC attacker identifies the superblock pointer and then targets a hash bucket by creating files whose inode number maps to that hash bucket. The *FT benchmark* victim depends on the futex table because it contains 64 threads which continuously acquires a shared lock for 100 microseconds. The FT attacker targets the futex table by allocating thousands of futex variables, probing the hash buckets to identify a busy bucket, and then parking thousands of threads on that hash bucket; as a framing attack, the attacker turns passive after parking the threads.

Figure 5 and Figure 6 shows the throughput and latency timelines, and the internal state of the data structures for both benchmarks. For the Vanilla kernel, the lock hold time increases tremendously in a short period leading to a significant drop in the victim's performance; at the end of the experiment, the throughput of the IC and FT victims drops by 95.95% $\pm$ 0.35% and 98.06% $\pm$ 0.31%, respectively. However, with Trātṛ, there is minimal impact on both throughput and latency when under attack compared to Vanilla (i.e., less than 2% $\pm$ 0.25% for the inode cache attack and less than 2% $\pm$ 0.51% for the futex table attack). We observe that the prevention and recovery mechanism never allows the attacker to expand the data structure. Note that for the IC benchmark, Trātṛ detects and ameliorates the attack even while the attacker is still preparing. For the FT benchmark, Trātṛ is able to return performance (especially noticeable for latency) to baseline



(a) Throughput **(b)** Latency **(c)** Lock Hold Times **(d)** Number of maximum entries

Figure 6: FT benchmark performance without attack, with attack and with Trāt̩r. *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trāt̩r kernel. (b) Timeline of the average latency of the time to release the lock measured using systemtap for a similar experiment. (c) Timeline of the lock hold times of the victim for Vanilla and Trāt̩r kernel. (d) Timeline of the maximum number of entries of the attacker in any bucket for Vanilla and Trāt̩r kernel.*

after recovery.

5.2 Effectiveness of Trāt̩r components

We evaluate the responsiveness of the detection, prevention, and recovery mechanisms in Trāt̩r for mitigating attacks on the IC and FT benchmarks.

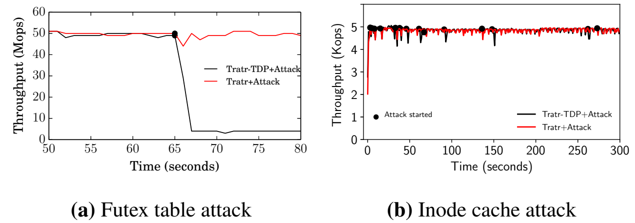
5.2.1 Detection

Early detection reduces the impact on a victim's performance. We present the initial timeline of the throughput of the IC and FT benchmarks in Figure 7; we begin the IC benchmark after the attacker's preparation phase has been completed. For IC, Trāt̩r detects the attack in its first sampling window (i.e., within 1 second of when the attack starts). For the FT attack, while the attacker probes the hash buckets, it does not disturb the lock hold times, and thus Trāt̩r does not suspect an attack. Once the attacker parks thousands of threads on the hash bucket, Trāt̩r detects the attack within 1 second of when the synchronization stalls cross the threshold limits.

5.2.2 Prevention

We now evaluate the prevention mechanism in which identified attackers are not allowed to allocate more objects. For the IC benchmark, when the attacker must first prepare, as originally shown in Figure 5, Trāt̩r detects the inode cache attack before the attacker is able to identify the superblock; as a result, the attacker is not able to launch the subsequent synchronization attack. In contrast, if the attacker already knows the superblock and can skip the prepare phase, as in Figure 7b, Trāt̩r is able to detect the synchronization attacks (shown as black circles), prevent the attacker from allocating more objects and keeping throughput high. In this scenario, Trāt̩r learns that the inode cache is under constant attack and grows the prevention window accordingly; at the end of the experiment, the prevention window is up to 250 seconds.

There is a slight dip in the throughput when an attack is detected and before recovery completes. As Trāt̩r needs to access the synchronization primitives to perform recovery, the victim observes a slight dip in the performance. For Trāt̩r-TDP, without recovery, the hash bucket continues to have the



(a) Futex table attack **(b)** Inode cache attack

Figure 7: Performance of Trāt̩r components. *Throughput timeline for the futex and inode cache attacks explaining the importance of detection, prevention and recovery. We also show the timeline for Trāt̩r-TDP. Trāt̩r-TDP denotes the kernel version that has the tracking, detection and prevention mechanisms enabled.*

entries belonging to the attacker. Thus, consecutive attacks slowly increase the hash bucket entries, leading to slightly more dips in the performance than Trāt̩r.

For the FT benchmark, Figure 7a shows that detection and prevention alone are not sufficient to isolate the attacker. Specifically for Trāt̩r-TDP, even after detecting the attack and preventing future objects from being allocated, the victim continues to traverse the expanded bucket. Thus, for framing attacks, even after an attacker has turned passive, the expanded data structure remains, and the prevention mechanism is not sufficient: recovery is needed to restore the data structure to a pre-attack state and recover baseline performance.

5.2.3 Recovery

Recovery is important to restore performance after an attack. Even though both the inode cache and futex table are hash tables, their recovery is significantly different. We show that for different recovery solutions, Trāt̩r can quickly bring performance back to normal.

For the inode cache, the recovery evicts all the entries belonging to the attacker. In Figure 7b, we observe that post-recovery, performance returns to baseline levels. In these experiments, the time to recover and evict thousands of entries is around 4-5 ms. Furthermore, the amount of work needed to recover is limited as the attacker can only inflict damage until the attack is detected.

For the futex table attack, Trāt̩r recovers by isolating the attacker from the victims. The time to recover and isolate thousands of attacker threads is 50-70 milliseconds. Further-

Application	Test Detail	Performance
CouchDB	Insertions	1.20% ± 0.02%
Cassandra	Read-Write mix	0.92% ± 0.003%
LevelDB	Random read	5.32% ± 0.01%
	Fill sync	-3.46% ± 0.01%
	Overwrite	-2.96% ± 0.01%
	Seek random	0.85% ± 0.02%
	Sequential fill	-2.49% ± 0.01%
RocksDB	Sequential fill	-4.16% ± 0.01%
	Random fill sync	-1.53% ± 0.002%
	Read while writing	0.41% ± 0.01%
InfluxDB	Concurrent write	-0.25% ± 0.01%
SQLite	Insertions	-3.33% ± 0.03%
Darktable	"Boat" test using CPU only	-3.05% ± 0.05%
Kripke	Equation solver	-1.90% ± 0.01%
RAR	Compress Linux kernel	-2.80% ± 0.003%
MNN	Inference on inception-v3 model	-0.11% ± 0.01%
NCNN	Inference on regnety_400m model	-1.31% ± 0.03%

Benchmark	Test Detail	Performance
Apache Benchmark	Static web page serving	-0.58% ± 0.01%
Blogbench	Read	-0.13% ± 0.01%
	Write	-9.67% ± 0.01%
Apache Siege	Concurrent connection on webserver	-2.26% ± 0.005%
Dbench	Concurrent clients doing I/O	-0.44% ± 0.02%
IOR	Parallel I/O tests with 1024 block size	-1.23% ± 0.003%
OSBench	Create files	-7.22% ± 0.03%
	Create threads	-2.25% ± 0.04%
	Launch Programs	-6.14% ± 0.02%
	Create Processes	-9.54% ± 0.04%
	Memory Allocations	0.31% ± 0.001%
Intel MPI Benchmark	PingPong test	-0.03% ± 0.005%
Neatbench	Neat Video render using CPU	5.88% ± 0.01%
FinanceBench	Bonds OpenMP Application	-1.19% ± 0.01%

Table 2: Performance overhead study for Phoronix test suite.*Comparison of tracking performance for various applications and benchmarks for the Vanilla kernel(baseline) and Trāt̤r-T with just tracking enabled for all the slab-caches relative to the Vanilla kernel. The numbers show the improvement over the baseline.*

more, as the attack is detected immediately after it is launched, the attacker cannot expand the hash bucket extensively, helping the recovery complete faster.

Prevention measures are necessary for faster recovery. Without the prevention measures, an attacker can continue to expand the data structure while holding the synchronization primitives. Since recovery must also acquire these synchronization primitives, longer lock hold times will make the recovery mechanism a victim.

To understand the performance impact of having a data structure isolated, we run another microbenchmark that comprises two processes – parent and child that use a pair of futexes located inside a shared anonymous mapping for synchronization. While the parent calls futex_wait() on the first futex and futex_wake() on the second futex after sleeping for 100 μs, the child process does the exact opposite. This way, only one process is active at a time. Using systemtap, we measure the time to complete the futex_wait() and futex_wake() calls. We iterate each process for one million iterations and run the microbenchmark on the Vanilla and Trāt̤r kernel. Without an attack, we observe that due to the extra code added for isolation in Trāt̤r, futex_wait() and futex_wake() is slower by 1-1.2% and 2-2.5% respectively. Using the cost for isolation, we can extrapolate the performance impact from a single data structure to multiple data structures. For example, if a system call accesses five data structures vulnerable to attacks, the theoretical worst performance cost will be around 5% to 12.5%. Remember that not all data structures can be isolated, as we discussed in Section 4.3. We believe that this performance gap can be reduced further with further tuning.

5.3 Overhead

We now show the performance and memory overhead introduced by Trāt̤r. For performance, we show the overhead incurred due to the tracking mechanism and by the additional kernel threads.

Tracking mechanism overhead. We measure the performance overhead of the tracking mechanism because it is the only code in Trāt̤r that is executed in the critical path; the work for detection, prevention, and recovery all occur in the background by kernel threads. To measure overhead, we use the Phoronix test suite, an open-source benchmark suite that supports more than 400 tests [50]. We run the Docker container provided by the Phoronix test suite for our experiments with unrestricted CPU access. Thus, multiple objects are created concurrently on all CPUs, stressing the tracking mechanism's parallelism. We average at least three runs.

We use 13 applications (CouchDB, Cassandra, LevelDB, RocksDB, InfluxDB, SQLite, Darktable, Kripke, RAR, MNN, NCNN, Apache Benchmark, and Apache Siege) and 7 benchmarks (Blogbench, Dbench, IOR, OSBench, Intel MPI Benchmark, Neatbench, Financebench) that heavily use the underlying kernel and allocate kernel objects, thus stressing the tracking mechanism. We use Trāt̤r-T, a special version that tracks all slab-caches. Note that tracking overhead is paid only when objects are allocated or freed and not while accessing.

Table 2 shows the performance of Trāt̤r kernel compared to the Vanilla kernel(baseline). The numbers only show the difference in the performance compared to the baseline. Raw numbers for the baseline and Trāt̤r can be found elsewhere [46]. For the majority of the applications and benchmarks, Trāt̤r's performance is within 0-5% of the Vanilla kernel's performance. Trāt̤r's NUMA-aware hash table for the user's accounting information minimizes the performance difference. We observe a slight performance improvement for a few workloads. In two benchmarks (i.e., Blogbench write; OSBench Create files, OSBench Launch programs, OSBench Create processes), the decrease in the performance is slightly higher (7-10%). These benchmarks create hundreds of kernel objects (e.g., processes or threads) in a short period, overwhelming the hash table used to track each user's accounting information. However, we believe that very few real-world applications create thousands of threads or processes rapidly.

Kernel threads overhead. Kernel threads sample the synchronization primitives to detect an attack and also initiate the prevention and recovery mechanisms. Even though these threads run in the background, they may still interfere with applications and compete for the CPU. To understand the

Application	Slab overhead in Trātṛ (in MB) (%)
CouchDB	10.14 (2.9%)
Cassandra	20 (9.52%)
LevelDB	5 (7.58%)
RocksDB	3.48 (3.8 %)
InfluxDB	11.37 (18.06 %)
SQLite	8.78 (9.49 %)
Darktable	1.71 (1.49 %)
Kripke	3.51 (7.55%)
RAR	3.1 (4.02 %)
MNN	7.94 (7.72 %)
NCNN	7.88 (8.32 %)

Benchmark	Slab overhead in Trātṛ (in MB) (%)
Apache Benchmark	5.1 (6.93 %)
Blogbench	19.66 (7.42 %)
Apache Siege	1.4 (3.68 %)
Dbench	19.93 (11.72 %)
IOR	11.44 (18.19 %)
OSBench	11.69 (13.41 %)
Intel MPI Benchmark	4.21 (7.7 %)
Neatbench	3.23 (11.25 %)
Financebench	4.48 (6.17 %)

Table 3: Memory overhead study for Phoronix test suite. Comparison of the memory overhead for various applications and benchmarks for the Vanilla and Trātṛ-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the second and fourth columns show the % increase in the total memory allocated to all the slab caches.

impact of these kernel threads, we use three CPU-heavy applications from the Phoronix test: N-Queens, CP2k Molecular Dynamics, and Primesieve. We run the same application in four containers and allocate each container 8 CPUs. By running the same test in all containers, we can easily isolate the performance variation caused by the kernel threads.

For all three applications, the performance difference between the maximum runtime with Trātṛ and the minimum runtime with Vanilla kernel is around 1-1.5%. This performance matches the design described in Section 4.3: on average, the kernel threads spend 120 μs sampling the synchronization primitives every 12.5 ms (approximately 1%).

Memory overhead. Trātṛ has two sources of additional memory overhead. First, Trātṛ uses a hash table with 32 buckets per NUMA node to track the memory allocations of each user for a slab cache. With two NUMA nodes, the hash table increases the size of each slab cache by 544 bytes. Therefore, the total memory overhead is less than 1 Megabyte (MB) per NUMA node to support thousands of slab-caches. Second, Trātṛ adds 4 bytes per object to track the user-id. As the Linux kernel uses slabs for object allocation, increasing the object's size by 4 bytes reduce the total objects allocated per slab.

To quantify the overhead on a running system, we use the `slabtop` command [18]. On an idle system with 150 slab-caches, we find 3.8% more memory is allocated to slab caches for Trātṛ-T (219 MB) than the Vanilla kernel (211 MB). Table 3 shows the memory overhead caused for the Phoronix suite. We do not report the confidence interval as we only measured the maximum memory usage throughout the entire test suite run and reported it. We notice that applications and benchmarks that regularly create objects are more likely to observe a higher memory overhead than the workloads that create few objects and use the same objects several times. We find that up to 20 MB more memory is allocated to slab caches with Trātṛ-T, which is less than 1% of the memory used in each experiment. We believe that given the amount of main memory available today, a few MB for slab-caches is reasonable.

Application	Workload
Filebench - DBENCH	128 threads executing client loadfile workload
UpScaleDB	ups_bench –inmemorydb –num-threads=32
Exim	Mosbench workload using 16 clients

Table 4: Application benchmarks and their workloads.

Container	Scenario 1 & 2 Futex table & Inode cache attack		Scenario 3 Multiple attacks	
	CPU	Memory	CPU	Memory
Exim	8	16 GB	8	16 GB
DBENCH	8	16 GB	8	16 GB
UpscaleDB	8	64 GB	8	64 GB
Attacker	1	8 GB	4	8 GB

Table 5: Scenario summary and resource allocation to each container.

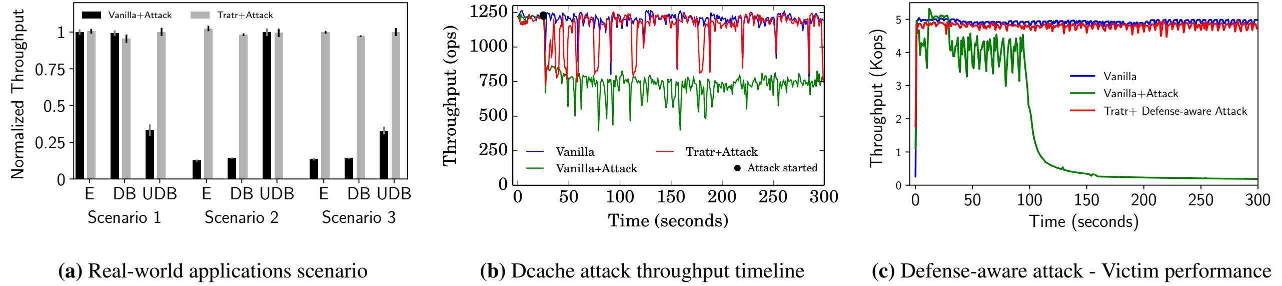
5.4 Real-World Applications

We demonstrate how Trātṛ can protect multiple potential victims from multiple attackers using real-world applications. We focus on three applications described in Table 4: Exim mail server (E), UpScaleDB (UDB), and DBENCH (DB). We run the workloads for 300 seconds as part of three scenarios summarized in Table 5. Scenario 1 uses a futex table attack, scenario 2 uses the inode cache attack, and scenario 3 uses both attacks simultaneously. Each application is running as a separate container on a single physical machine.

The normalized throughput of the applications with Vanilla and Trātṛ under the three attacks is shown in Figure 8a. For scenario 1, the futex table attack impacts UpScaleDB since it uses locks; however, Trātṛ brings the performance of UpScaleDB to match that with no attack. For scenario 2, the inode cache attack harms Exim and DBENCH with the Vanilla kernel; again, Trātṛ prevents and recovers from the attacks. Finally, scenario 3 shows that when both the futex table and inode cache attacks are launched simultaneously, all three victims observe poor performance with the Vanilla kernel. As desired, Trātṛ detects both attacks and employs different recovery solutions to mitigate each attack without impacting the victim's performance. Trātṛ is not limited to protecting a single victim or detecting and mitigating a single attack.

Cost of the attack. Launching a highly-damaging synchronization or framing attack against a Vanilla kernel does not require many CPU resources: for scenarios 1 and 2, 1 CPU is used to launch the attacks; for scenario 3, 2 CPUs are used. Thus, with minimal cost, an attacker can generate synchronization interference leading to poor performance.

Economic impact. Synchronization and framing attacks can cause an economic impact for victims. For example, in scenarios 1 and 3, with a Vanilla kernel, the UpScaleDB victim, may use 2.25-2.4x more CPU when under attack; however, with Trātṛ, UpScaleDB uses only 0.5% $\pm$ 0.05% more CPU. In scenarios 2 and 3, when attacked on a Vanilla kernel, the Exim Mail Server spends 32% $\pm$ 0.2% of its runtime waiting to acquire the inode cache lock (DBENCH spends 46% $\pm$ 0.35%); this waiting time corresponds to wasted resources with an economic impact.



(a) Real-world applications scenario

(b) Dcache attack throughput timeline

(c) Defense-aware attack - Victim performance

Figure 8: Performance comparison of the various applications across different scenarios. *Performance comparison of Vanilla and Trātṛ with and without attack when subjected to different attack scenarios. (b) Timeline of throughput showing impact on the throughput doe to the dcache attack for the Vanilla and Trātṛ kernel. (c) Throughput timeline for the inode cache attack highlighting the victim's performance.*

5.5 Adding new data structure complexity

One of our goals is to simplify the process of adding a new data structure to Trātṛ. We choose the dcache to understand the effort needed to implement the detection and recovery mechanisms. To enable tracking, we begin by setting the tracking flag for the dcache slab-cache. The dcache uses RCU and bit-based spinlocks to support concurrent access. We focus on RCU because the spinlock's critical section is small and difficult to make adversarial.

The dcache CSS measures the time to complete the `synchronize_rcu()` call; we choose a sampling threshold of 100 milliseconds. The dcache TCA walks the hash bucket with the most entries. Since the dcache is used for performance, the recovery procedure evicts all the attacker's entries; we re-use existing code to evict the attacker's entries from all superblocks. In total, we write 120 lines of new code to add the dcache to Trātṛ.

To test these code changes, we run the dcache attack described in Section 3. The attack targets a single hash bucket and creates thousands of negative entries to increase the RCU grace period. Figure 8b shows performance with the Vanilla and Trātṛ kernel with an attack. One can see that Trātṛ mitigates the attack. Given that the attacker can launch new attacks when the prevention window expires, the small initial window leads to relatively frequent dips in throughput; however, as time passes and Trātṛ increases the prevention window, performance dips occur less frequently.

5.6 False Positives

Another design goal is to have a low rate of false positives so that Trātṛ does not tag a victim as an attacker. Since Trātṛ relies on threshold limits to detect attacks, a poorly configured application or a system under high utilization could be tagged as an attack. We conduct a study to identify how many times false positives are detected. We use the 20 Phoronix applications and benchmarks, grouping five applications into a single container; we allocate 8 CPUs to each of the four containers and repeatedly run the stress tests over 24 hours.

During the 24 hours run, there were no situations where Trātṛ tagged an application as an attacker. This demonstrates that our thresholds are correctly calculated for heavy contention and the number of CPUs in the machine and that stressing the system does not breach the threshold limits. Automatically choosing the threshold values for different machines is an interesting avenue for future work.

We also deliberately misconfigured the filebench-webserver [63] workload to see if Trātṛ would tag it as an attacker. Not surprisingly, Trātṛ incorrectly identifies the workload as a futex table attacker a few times over the experiment. However, since the workload is not aggressively accessing the futex table, the prevention window size never proliferates; furthermore, the workload does not create more threads during the prevention window, so it does not have to stall. As a result, the workload sees a negligible reduction in performance (less than 1%).

5.7 False Negatives

A final goal of Trātṛ is to have few false negatives. Since Trātṛ relies on thresholds for detection, attacks that stay within the thresholds may not be detected. We describe a defense-aware inode cache attack and discuss the performance implications when Trātṛ cannot detect the attack.

The strategy of this attack is to expand a hash bucket such that inserting or accessing entries does not exceed the thresholds. To do this, the attacker creates entries until it reaches the thresholds, then stops and deletes those entries. Then, the attacker repeats this cycle, causing the victim to experience increased critical section sizes. However, the attacker cannot continuously create and delete files due to the random sampling window sizes and sampling delays in Trātṛ. The attacker must watch the sampling window size since Trātṛ moves into aggressive sampling mode if it finds that the synchronization stall exceeds the threshold even once.

To illustrate the impact of a defense-aware attacker on a victim, we use the IC benchmark. During the experiment, we verified that the lock hold times are always less than the thresholds, and thus the defense-aware attacker remains undetected. Figure 8c shows that even a defense-aware attacker is not able to cause much damage to the victim. Victims do not have to wait long for the inode cache lock since the attacker

cannot acquire the lock for more than the threshold. Thus, victims do not observe poor performance or denial-of-service with attackers that remain below the thresholds.

6 Limitations

As there is no distinct boundary defining a particular behavior as a minor performance inconvenience or a significant performance problem, Trātṛ uses sampling thresholds to detect an attack. A defense-aware attacker may be able to stay within these threshold boundaries and remain undetected. Under such a condition, the victim may continue to observe minor performance issues where the attacker can elongate the critical section size to threshold values. By lowering the threshold boundaries, Trātṛ can push the attack boundary lower. However, by doing so, Trātṛ may end up increasing the false positive rate. Our goal is to avoid false-positive cases as much as possible. By replacing the existing mutual exclusion locks with SCLs, the problem of minor performance inconvenience can be avoided as they guarantee lock usage fairness.

Secondly, an attacker can use other services available in the operating system to expand a data structure making the service accountable for its size. For example, the attacker can ask the print spooler to load files whose hash values map to a single bucket for printing. Trātṛ will treat the print spooler as the one who created the inodes and may tag it as the attacker. Similarly, it is possible that a remote attacker can employ a confused deputy attack [31] and target a service such as a web server by making Trātṛ believe that the targeted service is the attacker leading to a denial of service on the targeted service.

Thirdly, the current threat model assumes that an attacker cannot colocate containers on a single physical machine to launch a coordinated attack. If an attacker can launch colocation attacks using the techniques discussed elsewhere [59], the existing detection mechanism will fail to detect the attacker. Due to colocation, each user's total object allocation count can stay within the majority limits making it harder to identify an attacker. The existing detection mechanism needs to be further enhanced to include statistical properties of the object allocations to detect multiple attackers. Also, as only a few containers can run on a single physical machine, it is impossible to launch a coordinated attack involving tens of containers.

7 Related Work

Synchronization attacks and framing attacks are closely related to Algorithmic complexity attacks (ACA) [22]. To launch synchronization and framing attacks, an attacker exploits the weak complexity guarantees of a data structure. To avoid ACA's, researchers have proposed several techniques such as universal hashing [22], using balanced trees [22], randomized algorithms [22], or probabilistic algorithms [19].

However, using secure hash functions can have performance implications [21], replacing the existing data structures to use balanced trees is a tedious process [20, 49]; not all data structures can be replaced by trees. There have been attacks against randomized [15] and probabilistic algorithms [44, 56].

Detection and prevention is another approach to tackle ACAs. Khan et al. propose an alternative to randomization through regression analysis based model to prevent attacks [34]. Qie et al. propose an approach where they show annotating application code can help detect resource abuse and accordingly initiate rate-limiting or dropping the attackers. Similarly, FINELAME also uses annotation and probing to detect attacks [24]. DDOS-Shield assigns continuous scores to user sessions and checks these scores to identify suspicious users and accordingly prevent them from overwhelming the resources [54]. Trātṛ too looks for anomalous resource allocations by checking LCS and HSUA. Moreover, prevention is not enough to address framing attacks. No other approach considers synchronization as a resource, unlike Trātṛ.

Radmin learns and executes Probabilistic Finite Automatas offline of the target process of all the monitored resources and then perform anomaly detection by making sure that the target programs stay within the learned limits [26]. It will be an interesting avenue to explore building lock usage models and using them to detect attacks in Trātṛ.

8 Conclusion

We introduced a new class of performance attacks – synchronization and framing attacks in Linux kernel when using containers in a shared infrastructure environment. These attacks significantly impact victims' performance leading to denial-of-service. To remedy this, we introduce Trātṛ that can address these attacks by detecting them within seconds and instantaneously recovering from the attack. The performance of victims with Trātṛ is similar to the baseline performance.

A thorough analysis is needed to identify vulnerable data structures in the Linux kernel and other concurrent shared infrastructure. We look forward to analyzing the Linux kernel and expanding Trātṛ to support different data structures. Trātṛ's source code and the attack scripts will be made public.

9 Acknowledgments

We thank Anil Kurmus (our shepherd) and the anonymous reviewers of Usenix Security for their insightful comments and suggestions. We thank Thomas Ristenpart, Rahul Chatterjee, and David Dice for reviewing the initial draft and providing feedback. We thank the members of ADSL for their excellent feedback. We also thank CloudLab [57] for providing a great environment to run our experiments. This material was supported by funding from NSF grants CNS-1763810 and CNS-1838733, Google, Intel, Samsung, and VMware.

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] AppArmor Documentation. https://gitlab.com/apparmor/apparmor/-/wikis/Documentation.

[2] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] cscope. http://cscope.sourceforge.net.

[4] CVE-2003-0718. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0718.

[5] CVE-2004-0930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0930.

[6] CVE-2005-0256. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0256.

[7] CVE-2005-1807. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1807.

[8] CVE-2005-2316. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2316.

[9] CVE-2007-1285. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1285.

[10] CVE-2008-2930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2930.

[11] CVE-2008-3281. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3281.

[12] CVE-2011-1755. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1755.

[13] Nitin Agrawal, Leo Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Emulating Goliath storage systems with David. *ACM Transactions on Storage (TOS)*, 7(4):1–21, 2012.

[14] Tigran Aivazian. Inode Caches and Interaction with Dcache. https://tldp.org/LDP/lki/lki-3.html.

[15] Noa Bar-Yosef and Avishai Wool. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In *International Conference on E-Business and Telecommunications*, pages 162–174. Springer, 2007.

[16] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 1–16, 2010.

[17] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *Performance Evaluation Review*, 35(2):42, 2007.

[18] Chris Rivera and Robert Love. slabtop(1) - Linux manual page. https://man7.org/linux/man-pages/man1/slabtop.1.html.

[19] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic Data Structures in Adversarial Environments. In *19 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1334, 2019.

[20] Jonathan Corbet. How to get rid of mmap_sem. https://lwn.net/Articles/787629/.

[21] Jonathan Corbet. SipHash in the kernel. https://lwn.net/Articles/711167/.

[22] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*, pages 29–44, 2003.

[23] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-service. *VLDB Endow.*, 7(1):37–48, September 2013.

[24] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference, USENIX ATC'19*, pages 693–708, 2019.

[25] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[26] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: Early Detection of Application-Level Resource Exhaustion and Starvation Attacks. In *International Symposium on Recent Advances in Intrusion Detection*, pages 515–537. Springer, 2015.

[27] Fabio Kung, Sargun Dhillon, Andrew Spyker, Kyle Anderson, Rob Gulewich, Nabil Schear, Andrew Leung, Daniel Muinom and Manas Alekar. Evolving Container Security with Linux User Namespaces. https://netflixtechblog.com/evolving-container-security-with-linux-user-namespaces-afbe3308c082.

[28] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG Conference Proceedings*, volume 85. AUUG, Inc. Kensington, NSW, Australia, 2002.

[29] Amir Reza Ghods. A Study of Linux Perf and Slab Allocation Sub-Systems. Master's thesis, University of Waterloo, 2016.

[30] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, 2006.

[31] Norm Hardy. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[32] Docker Inc. Docker Engine managed plugin system. https://docs.docker.com/engine/extend/.

[33] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud. In *4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 8–8, 2012.

[34] Suraiya Khan and Issa Traore. A Prevention Model for Algorithmic Complexity Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 160–173. Springer, 2005.

[35] Michael Larabel. FUSE Gets User Namespace Support With Linux 4.18. https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.18-FUSE.

[36] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-based File System. In *11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, 2014.

[37] M. Douglas McIlroy. A Killer Adversary for Quicksort. *Software: Practice and Experience*, 29(4):341–344, 1999.

[38] Paul E. McKenney. The new visibility of RCU processing. https://lwn.net/Articles/518953/.

[39] Paul E. McKenney and Jonathan Walpole. What is RCU, Fundamentally. *Linux Weekly News (LWN. net)*, 2007.

[40] Paul Menage. CGroup documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2018.

[41] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018.

[42] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal*, 2014(239):2, 2014.

[43] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. https://lwn.net/Articles/772885/.

[44] Moni Naor and Eylon Yogev. Bloom Filters in Adversarial Environments. In *Annual Cryptology Conference*, pages 565–584. Springer, 2015.

[45] Neil Brown. Linux kernel design patterns - part 2. https://lwn.net/Articles/336255/.

[46] Yuvraj Patel. *Fair and Secure Synchronization for Non-Cooperative Concurrent Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2021.

[47] Yuvraj Patel, Mohit Verma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Revisiting Concurrency in High-Performance NoSQL Databases. HotStorage '18.

[48] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[49] Ben Pfaff. Performance analysis of BSTs in system software. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[50] Phoronix Media. Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking. https://www.phoronix-test-suite.com.

[51] Aravinda Prasad and K. Gopinath. Prudent Memory Reclamation in Procrastination-Based Synchronization. In *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 99–112, 2016.

[52] Aravinda Prasad, K. Gopinath, and Paul E. McKenney. The RCU-Reader Preemption Problem in VMs. In *2017 USENIX Annual Technical Conference, USENIX ATC'17*, pages 265–270, 2017.

[53] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577, 2020.

[54] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. DDoS-shield: DDoS-resilient scheduling to counter application

layer attacks. *IEEE/ACM Transactions on networking*, 17(1):26–39, 2008.

[55] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[56] Pedro Reviriego and David Larrabeiti. Denial of Service Attack on Cuckoo Filter based Networking Systems. *IEEE Communications Letters*, 2020.

[57] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[58] Dipankar Sarma and Paul E. McKenney. Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications. In *2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191, 2004.

[59] Sushrut Shringarputale, Patrick McDaniel, Kevin Butler, and Thomas La Porta. Co-residency Attacks on Containers Are Real. In *2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, page 53–66, 2020.

[60] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, 2012.

[61] Cristian-Alexandru Staicu and Michael Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. USENIX Security 18, pages 361–376.

[62] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, 2019.

[63] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[64] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. How to Get More Value From Your File System Directory Cache. In *25th Symposium on Operating Systems Principles*, pages 441–456, 2015.

[65] UpScaleDB Inc. UpScaleDB. https://upscaledb.com/.

[66] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies FAST'17*, pages 59–72, 2017.

[67] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th USENIX Conference on Operating Systems Design and Implementation*, OSDI '02, 2002.

[68] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, 1994.

10 Appendix

10.1 Extract super block pointer

10.1.1 Hash function parameters

The inode hash function is parameterized by two variables: `cshift` and `ishift`. `cshift` is log_2 of L1 cache line size in bytes. Most server machines have 64 bytes L1 cache line, so `cshift` is always 6 on these machines. `ishift` is log_2 of the number of buckets in the inode hash table, which is determined when the inode cache is allocated. `ishift` is 22 or 23 on the machines that we ran the experiments. To make discussion easier, we use `ishift=23` for the following examples and implications unless otherwise specified. A demonstration of hash implementation is shown in Fig. 9.

```
#define GOLDEN_RATIO_64   0x61c8864680b583eb
#define cshift   6
#define ishift   23
#define imask   ((1UL << ishift) - 1)
#define last23bits(x) ((x) & imask)
#define middle23bits(x) (last_23bits((x) >> ishift))
unsigned long ihash(unsigned long sb,
     unsigned long ino)
{
 unsigned long var1, var2, var3;
 var1 = ino * sb;
 var2 = (GOLDEN_RATIO_64 + ino) >> cshift;
 var3 = var1 ^ var2;
 bucket = last23bits(var3) ^ middle23bits(var3);
 return bucket;
}
```

Figure 9: Inode hash function implementation with `cshift=6` and `ishift=23`. All the variables are 64-bit. Only the last 46 bits of `var3` are used for computing `bucket`. `var3` is calculated as bit-wise XOR of `var1` and `var2`, so only the last 46 bits of these two variables are used.

10.1.2 Hash function properties

We observe the hash function has three properties, and we derive another three implications to better utilize these properties.

Property 1. For any super block pointer `sb` and inode number `ino`, the following equation always holds:
`ihash(sb, ino) == ihash(sb+m, ino+n)`, where $m \in \{i*2^{2*ishift} | i \in \mathbb{N}\}$ and $n \in \{i*2^{cshift+2*ishift} | i \in \mathbb{N}\}$.

For a machine with `ishift=23` and `cshift=6`, this property claims flipping the most significant 18 bits of `sb` and 12 bits of `ino` does not affect hash results. We call these bits *non-essential*. Intuitively, these bits are masked out at some point, so flipping them does not result in any difference in the hash output. Other bits that would affect the hash result are called *essential*. In addition, within $(2*$`ishift`$)$ essential bits of `sb`, the least significant 11 bits are all zero due to the fact that `sb` is a pointer to `struct super_block`. The essential bits have two implications.

Implication 1. We only need to extract the 46 essential bits of `sb`; with the 11 trailing zeros known, we only need to extract the other 35 bits.

Implication 2. For any given inode number, we could easily get another 4095 inode numbers that would be hashed to the same bucket by flipping these non-essential bits.

We call these inode numbers obtained by flipping non-essential bits *sibling* inode numbers.

Algorithm 1: Extracts the last 34 bits of the super block pointer

Input: threshold
Output: sb_last34bits

1. **for** n = 0 *to* 4095 **do**
2.   `InsertInodeCache`(*0x800000000* + $n \ll 52$)
3. ino_x = $1 \ll 12$;
4. latency = 0;
5. GOLDEN_RATIO_64 = 0x61c8864680b583eb;
6. **while** *latency* < *threshold* **do**
7.   t0 = `now()`;
8.   `InsertInodeCache`(*ino_x*);
9.   latency = `now()` - t0;
10.   ino_x = ino_x + $(1 \ll 13)$;
11. bucket = 0x4a697a;
12. var2 = (GOLDEN_RATIO_64 + ino_x) $\gg$ 6;
13. var2_last23bits = var2 & 0x7fffff;
14. var2_middle23bits = (var2 $\gg$ 23) & 0x7fffff;
15. var1_middle23bits = bucket ^ var2_last23bits ^ var2_middle23bits;
16. sb_12to34bits = `ExtendedEuclidean`(*ino* $\gg$ 12, *var1_middle23bits,* $1 \ll 23$);
17. sb_last34bits = sb_12to34bits $\ll$ 11;

Property 2. For any super block pointer `sb1` and `sb2`, `ihash(sb1, n) == ihash(sb2, n)`, where $n \in \{i*2^{2*ishift-11} | i \in \mathbb{N}\}$.

Property 2 means, if an inode number has 35 trailing zero bits, it will be hashed to a deterministic bucket regardless the value of `sb`. For example, `0x4a697a` is an anchor bucket because `ino=0x800000000` will always be hashed to `0x4a697a`. We call these buckets *anchor buckets*.

Combining property 1 and 2, we are able to add `ino=0x800000000` and its siblings into the anchor bucket `0x4a697a`. These 4096 inodes would form a long list in the bucket. Every time the kernel wants to insert another inode into this bucket, it must traverse through the list first to ensure this inode is not already present, which would result in observable latency.

Implication 3. After the anchor bucket `0x4a697a` being populated, if inserting another inode with inode number `ino` takes significantly longer, we know this inode goes to bucket `0x4a697a` and `ihash(sb, ino)==0x4a697a`.

Property 3. Given two super block pointers `sb1` and `sb2` whose the last k bits are same, the $(k+1)$th bit different, and `ihash(sb1, n) != ihash(sb2, n)`, where $n \in \{i*2^{2*ishift-k-1} | i \in \mathbb{N}\}$, then for any `sb3` having the same last k bit, the following equation is always true: `ihash(sb3, n) == ihash(sb, n)`, if the $(k+1)$th bit of `sb3` is same as `sb`, for `sb` $\in \{$`sb1`, `sb2`$\}$.

Property 3 means, given the last k bits of `sb` fixed, the hash result of a given inode number which has $(45-k)$ trailing zero bits only has two possible outputs, depending on whether the $(k+1)$th bit is zero or one.

10.1.3 Extract the last 34 bits of the superblock pointer

With the properties and implications studied, we describe algorithm 1 to extract the last 34 bit of `sb` . Algorithm 1 takes a latency threshold as the input to determine whether insertion latency in Implication 3 is "long" enough. It first populates bucket `0x4a697a` (line 1-2). The function `InsertInodeCache` can be implemented as any filesystem operation that would touch the inode cache. In our experiments, we developed a FUSE-based filesystem that implements name lookup as translating the name into inode number e.g lookup a file with name "0x123" will return inode number 0x123; we use syscall `stat("0x123", &statbuf)` as the implementation for `InsertInodeCache(0x123)`.

In line 3-9, it searches for an inode number `ino_x` satisfying 1) has 12 trailing zero bits. 2) the least significant 13th bit is one. 3) is hashed to the bucket `ex4a697a`. These requirements are used in lines 10-15 to reverse the 12th to 34th bits of `sb`. Requirement 1 ensures `var1` has the last 23 bits all zeros, so its middle 23 bits (the least significant 24th to 46 bits) can be reversed by bitwise XOR.

As `var1` is the product of `sb` and `ino_x`, we get a linear congruence equation after removing the trailing zeros:

$$ax = b, \bmod 2^{23}$$

where x is 12th to 34th bits of `sb`, a is 13th to 35th bits of `ino_x`, and b is `middle23bits(var1)`. This equation can be solved by the extended Euclidean algorithm [41]. To ensure the unique solution, the extended Euclidean algorithm requires a and 2^{32} are relatively prime. This is satisfied by the requirement 2 which ensures the a is odd.

Algorithm 2: Extracts the rest of essential bits of super block pointer

```
   Input: sb_last34bits
   Output: sb
1  Function ReverseHash(sb, bucket, trailing_zero)
2      i = 1;
3      while true do
4          ino = i ≪ trailing_zero;
5          if ihash(sb, ino) == bucket then
6              return ino;
7          i = i + 1;
8  sb = sb_last34bits;
9  bucket = 0x4a697a;
10 for b = 35 to 46 do
11     sb_if0 = sb;
12     sb_if1 = sb | (1 ≪ b);
13     ino_if0 = ReverseHash(sb_if0, bucket, 46 - b);
14     ino_if1 = ReverseHash(sb_if1, bucket, 46 - b);
15     t0 = now();
16     InsertInodeCache(ino_if0);
17     t1 = now();
18     InsertInodeCache(ino_if1);
19     t2 = now();
20     latency_if0 = t1 - t0;
21     latency_if1 = t2 - t1;
22     if latency_if1 > latency_if0 then
23         sb = sb_if1;
24     else
25         sb = sb_if0;
```

10.1.4 Extract the rest of essential bits: Guess then test

After knowing the last 34 bits of `sb`, we introduce algorithm 2 to extract the rest of the essential bits. It inductively guesses a bit of `sb` then tests whether this guess is true.

Suppose we already know the last 34 bits of `sb` and want to know the 35th bit ($b = 35$ in line 10). Property 3 suggests when the inode number has 11 trailing zero bits; the hash output purely depends on the 35th bit of `sb`. We thus name two candidates as `sb_if0` and `sb_if1`.

Line 13-14 search for an inode number with 11 trailing zero bits and would be hashed to the anchor bucket `0x4a697a` if the corresponding candidate is true. In lines 15-21, these two inodes are inserted into the inode cache. If the 35th bit is one, then inserting `ino_if1` should take significantly longer, vice versa. By comparing the latency (lines 22-25), we could know whether this bit is zero or one. We then inductively extract other bits.

A potential problem is, `ReverseHash` may fail to find any inode number that satisfies the requirements. This can happen if requiring too many trailing zeros. As an alternative, we are motivated to use algorithm 1 to get the last 34 bits first, so b in line 10 could start with 35 instead of 12. For $b \leq 35$ and $trailing_zero \leq 11$, there are two at least 2^{41} candidates having that many zero bits, and each candidate have $1/2^{23}$ chance produces the hash result we want.

10.2 Linux Kernel Critical Section analysis

We now discuss the manual analysis for the Linux Kernel's critical section. First, we identified global and static locks in a few directories in the Linux source code tree – filesystem (fs), kernel core functionality (kernel), memory management (mm), and security functionalities (security). Using the cscope tool [3], we then looked for the critical sections associated with the identified locks and started making notes. Within the critical section, we looked for instances that contain loops and instances that call synchronize_rcu(). On finding such instances, we note the information about the critical sections and consider that as a potential vulnerability that an attacker can exploit.

The manual analysis was hard and time-consuming as we dealt with debug code, function pointers, different coding patterns, etc. Over time, due to familiarity, we were able to quickly identify debug code and common function pointers, making things easier. There were other instances where the lock was acquired in one function and released in another function. One such example is the inode_hash_lock used to protect the inode cache hash. In find_inode_fast(), the lock is released in __wait_on_freeing_inode() if the inode state is being deleted and again acquired after waiting for the deletion process to complete so that the find_inode_fast() function can continue with the inode finding process. Similarly, we find that within a function, for a single instance of lock acquisition, there are numerous lock release instances. We considered such one-to-many mapping as one critical section to ease our analysis.

We identified other factors that can expand the critical section sizes, such as allocating an object, freeing an object, nested loops, hierarchical locking, number of functions called, etc. However, including all such factors makes the manual analysis process cumbersome and time-consuming. Therefore, we believe that developing a tool that can perform critical section analysis will be useful to identify vulnerabilities. However, our initial attempt to design such a tool did not work out due to function pointers, different coding patterns, etc. Thus, we had to perform the minimal critical section analysis manually.