Exploiting Kubernetes' Image Pull Implementation to Deny Node Availability

Luis Augusto Dias Knob, Matteo Franzil, and Domenico Siracusa

***Abstract*—Kubernetes (K8s) has grown in popularity over the past few years to become the *de-facto* standard for container orchestration in cloud-native environments. While research is not new to topics such as containerization and access control security, the Application Programming Interface (API) interactions between Kubernetes (K8s) and its runtime interfaces have not been studied thoroughly. In particular, the CRI-API is responsible for abstracting the container runtime, managing the creation and lifecycle of containers along with the downloads of the respective images. However, this decoupling of concerns and the abstraction of the container runtime renders K8s unaware of the status of the downloading process of the container images, obstructing the monitoring of the resources allocated to such process. In this paper, we discuss how this lack of status information can be exploited as a Denial of Service attack in a K8s cluster. We show how such attacks can impact worker nodes, generating up to 95% average CPU usage, prevent downloads of new container images, and increase I/O and network usage for a potentially unlimited amount of time. We argue that solving this problem would require a radical architectural change in the relationship between K8s and the CRI-API, which would be unfeasible in the short term. Thus, as a stopgap solution, we propose MAGI: an eBPF-based, proof-of-concept mitigation that detects and terminates potential attacks.**

***Index Terms*—Kubernetes, container security, cloud computing, eBPF.**

I. Introduction

SINCE its inception, Kubernetes (K8s) has been conceived as a container management system to orchestrate multiple workloads in a scalable, resilient, and reliable way [1]. This containerization deployment-oriented approach enables cluster and application automation, multi-tenancy, and reproducibility. Additionally, K8s' loosely coupled architecture based on asynchronous communication, isolation, and decoupling of concerns pioneered a remarkable change in the cloud-native computing landscape.

Being implemented with a modular structure, K8s uses a set of APIs to manage the network, storage, and container runtimes. This creates a scenario where several modules with distinct implementation and requirements need to communicate to execute a task (e.g., the creation of containers), usually in an asynchronous way. Such a design provides high availability, scalability, and performance but creates gaps that could potentially be used as attack vectors.

In this paper, we investigate the hazards of the asynchronous communications between two modules inside the K8s cluster, the Kubelet, the module that runs on each node and communicates with the central API, and the container runtime, which is tasked with handling the creation, management, and deletion of containers in K8s nodes. Our study figures that since the communication between these modules happened asynchronously, it does not maintain any intermediate status on each request, enabling a malicious user to exploit this behavior as a threat that can be used as a vector to several distinct attacks.

In particular, when deploying a Pod — a group of one or more containers — an image pull is usually triggered for the corresponding containers if unavailable locally. If a container is deleted before the image is fully downloaded, the API will not communicate this action to the container runtime. However, the runtime will still download the image, wasting resources in the process. Similarly, when a container is forcibly destroyed, its resources are not instantly released. While K8s will delete all information from the API as soon it receives the request, the complete elimination of the containers composing the Pod will only occur when all modules finish their deletion processes. Although this is intended behavior, such a process could be exploited to mount a Denial of Service (DoS) attack on the cluster's worker nodes. Indeed, these image download requests would prevent other users from scheduling new jobs without a cached image and force the tenant to consume unnecessary resources.

To understand the actual impact, we present several possible attack scenarios, studying the impact on the scheduling, system resources, and other containers running in the node. We also describe in detail the threat, with its limitations and requirements. We show how this issue affects all Kubernetes deployments, demonstrating its effect both in private and public cloud settings. Additionally, we discuss how this could be solved in K8s. Since the solution is not trivial and will require time and effort to be achieved, we present a temporary, eBPF-based proof-of-concept

Received 22 January 2024; revised 3 October 2025; accepted 2 December 2025. Date of publication 8 December 2025; date of current version 12 March 2026. This work was supported in part by Project SERICS through PNRR MUR Program funded by the European Union – NextGenerationEU under Grant PE00000014 and in part by European Union's Horizon Europe Programme under Grant Agreement 101070473 (Project FLUIDOS). *(Corresponding author: Luis Augusto Dias Knob.)*

Luis Augusto Dias Knob is with DAISY, Center for Cybersecurity, Fondazione Bruno Kessler, 38123 Trento, Italy (e-mail: l.diasknob@fbk.eu).

Matteo Franzil is with the Department of Information Engineering and Computer Science, University of Trento, 38123 Trento, Italy, and also with DAISY, Center for Cybersecurity, Fondazione Bruno Kessler, 38123 Trento, Italy (e-mail: matteo.franzil@unitn.it).

Domenico Siracusa is with the Department of Information Engineering and Computer Science, University of Trento, 38123 Trento, Italy, and also with the Center for Cybersecurity, Fondazione Bruno Kessler, 38123 Trento, Italy (e-mail: domenico.siracusa@unitn.it).

Digital Object Identifier 10.1109/TDSC.2025.3641311

solution to address the problem until the fix is fully implemented in all the relevant projects.

We also argue that the majority of K8s security research over the last ten years focused on lateral container movement, container escapes, and authentication and access control [2], [3], [4], [5], [6], [7]. These topics have been extensively researched, and several ideas have been proposed to address these issues. However, studying the interactions between K8s and its runtime interfaces is one area that is frequently disregarded both in the literature and in practice, mainly due to the challenges presented by a modular project with several distinct working groups. Taking that into account, the contributions of this work are summarized as follows.

New resource-based DoS threat in K8s: We demonstrate that it is possible to exploit a series of design flaws in the Kubernetes CRI-API to execute a DoS attack. We also discuss the importance of API status routes and how the asynchronous communication between distinct modules can be used as a vulnerability point.

Attacking strategies: We explore how such an attack can overcome a cluster's defenses and mount a DoS attack on a target node, blocking the image download of other applications and hampering the resource use by the containers. We also show that it is possible to manipulate the node's image cache, increasing the time needed to start applications in the cluster.

eBPF proof-of-concept solution: We discuss how the problem can be mitigated, with the pros and cons of each solution. We thoroughly explain how the current design flaws in K8s's CRI-API could be efficiently fixed. As a stopgap solution, we propose a proof-of-concept that employs the extended Berkeley Packet Filter (eBPF) to intercept and mitigate rogue downloading images.

II. Background

This section provides the technical background for the remainder of this work. The following subsections present a small review of the Kubernetes, the containerized application lifecycle, and the Container Runtime Interface.

A. Kubernetes

Kubernetes is an open-source orchestration system for automating deployment, scaling, and management of containerized applications, initially developed by Google and maintained by the Cloud Native Computing Foundation (CNCF) [8]. K8s is built from a set of composable modules through a set of standard APIs that can be extended by the users alongside the core components [1]. Over time, this allowed the development of several new infrastructure paradigms, like Function as a Service (FaaS) frameworks [9] and multi-cluster management [6].

K8s uses a modular structure to implement all its services, each with a distinct function, in the cluster. The API Server is the central communication hub for the entirety of K8s. It processes requests, validates them, and maintains the desired cluster state. After processing the request, the API Server is responsible for forwarding the desired state to other modules, like the Kube Controller or the Kube Scheduler. Furthermore, the API Server enforces access controls and authentication mechanisms, ensuring secure and managed access to cluster resources.

To deploy a containerized application on K8s, the minimum manageable object is the Pod. A Pod is a group of one or more containers that share resources, shown as a single access point for the other objects in the cluster. Pods are designed to be ephemeral, and represent a single instance of an executable resource in K8s. Rarely created as an individual object, it usually uses a more complex controller, such as StatefulSets and Deployments, to manage its deployment and replication on the cluster nodes.

B. Container Runtime Interface

K8s relies on a well-defined API called Container Runtime Interface, also called CRI-API or just CRI, which communicates through gRPC Remote Procedure Calls (gRPCs) [10] with the container runtime to instantiate the Pods in the nodes. Today, the default runtime used by K8s is *containerd*, also maintained by the CNCF. The Kubelet module, that is the node agent for the K8s API, uses the CRI to execute the operations in the container runtime. Finally, the container registry serves as a centralized repository for storing, organizing, and distributing container images to nodes when required. The registry additionally enables a secure and efficient way to manage the entire lifecycle of container applications.

The Container Runtime Interface (CRI) was developed as a replacement for Dockershim, which previously served as the bridge between K8s and Docker. The introduction of CRI aimed to standardize communication with container runtimes, eliminating the need for intermediary solutions such as Docker. This standardization enhances compatibility and fosters a more modular approach, allowing seamless integration with emerging container technologies such as Kata Containers and gVisor [11].

C. Container Image Deployment

To better understand how the Pod lifecycle is managed in the cluster, Fig. 1 shows the sequence diagram for three `kubectl` commands: first, when a Pod deployment is initiated using the *run* command; second, when that same Pod is deleted; and finally, when the deletion is forced with the `--force` option. In this last scenario, the API does not wait for the CRI to confirm the deletion status. It instead immediately removes any related entries in etcd (K8s's database) once the command is entered.

We can break the execution of all commands into two distinct steps. Initially, the user interacts with the API that redirects the command to the correct node. The second step is executed on the selected node asynchronously with respect to the first step. Only at the end the status of the execution is returned to the API. While, by default, the instantiation happens asynchronously, the deletion will wait until a response is returned. To change this behavior, `kubectl` implements two options, `--force` and `--grace-period=0`, that execute the deletion without waiting for the Pod completion process.

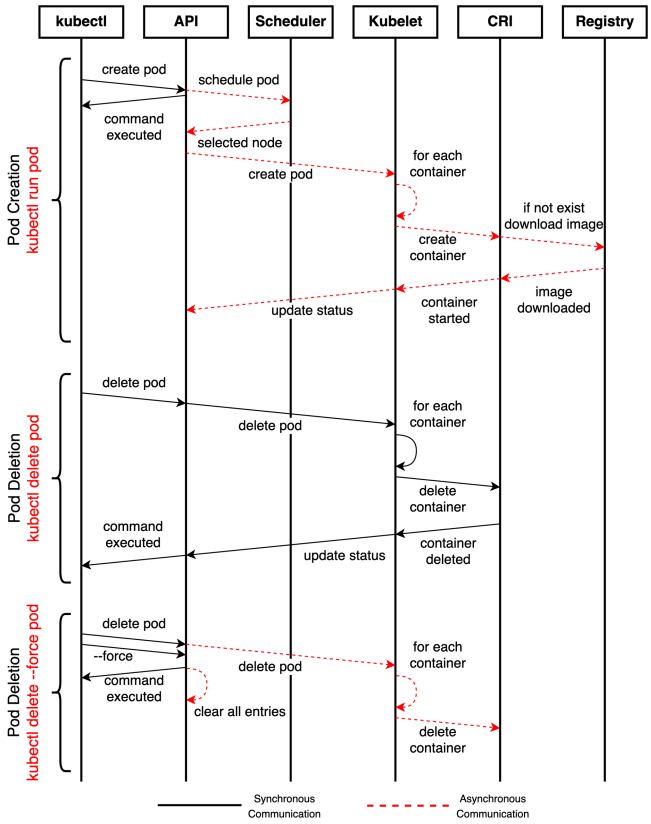


Fig. 1. Pod creation and deletion sequence diagrams.

Implemented to improve the agility of the cluster management, these options can, as described in the K8s official documentation[1], be disruptive for some workloads, or some resources may continue to run indefinitely on the cluster. We exploit this behavior to show that resources could be used by the node without any notice, mainly by how the CRI was developed.

III. Motivation and Attack Requirements

In this section, we first present the threat model and assumptions of our work. Second, we describe our motivation. Afterwards, we describe the requirements to execute the attack on K8s, including the configuration and image parameters that can increase the attack surface and its impact.

A. Threat Model and Assumptions

The attacker aims to increase resource usage inside a node without being noticed by the administrator. That behavior can be used as a single DoS attack or an auxiliary action to more complex threats. We assume that the attacker has access to a Service Account (SA) or an user credential from a compromised Kubelet config file, or through an exploited container with a SA linked to them. This SA requires a minimal set of permissions for the attack: creation and deletion of Pods.

As described in the Role-Based Access Control (RBAC) Best Practices official documentation,[2] it is expected that a single cluster will have several distinct SAs to limit the access to specific namespaces, to share the cluster between production and development, or to enable a continuous integration/delivery solution. So, it is reasonable to assume that one of these service accounts could be leaked from a development machine, a non-protected staging deployment used as a vector to access an inside container with a mount SA token, or an insider malicious user with access to the cluster.

We also assume that quotas and limitations on the number of API requests per second are active in the cluster. Further security configurations, e.g., image validation and locked-down private registries, can harden the attack execution but not wholly mitigate it. Indeed, a well-tuned configuration can make the threat viable even in a very constrained scenario.

On the restriction side, we understand that a compromised container or an SA token could be used for more advanced attacks, such as lateral movement [7], Out-of-Memory (OOM) attack [12], or deploying cryptomining in the node [13]. However, all these attacks can be straightforwardly identified by inspecting the resource usage or checking the elements running in the cluster. Notwithstanding, this attack wants to create a stealthy channel to cripple the cluster's resources that cannot be easily identifiable without an extensive evaluation of the node resources and processes, differentiating it from the other ones.

On the other hand, the attack can be performed with the same minimal permissions on a wide variety of settings, ranging from busy clusters with several concurrent deployments to smaller, soft multitenancy clusters[3], and multi-cluster topologies [14]. Attackers can choose to target a specific node, for example, to target a GPU-enabled node. Using a `nodeSelector` or `podAffinity`, attackers can ensure that the Pod are scheduled on the desired node(s), further increasing the attack's impact. Even with monitoring or auditing logs active, we argue that the attack will be hard to discover without a specific and deeper analysis of the node itself, which is not a trivial task. Indeed, even in cloud deployments — such as Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), and Amazon Elastic Kubernetes Service (EKS) — monitoring tools tend to provide an aggregated view of the cluster's resource usage, and administrators usually will not access directly the cluster nodes.

The registry employed by the attacker can be any public or private registry, as long as it is reachable from the cluster. Some public registries, like DockerHub, provide strict rate limits, which are computed per IP. Since the IP pulling the image is the node's IP, being banned translates into a denial of service for the node itself, which is exactly what the attacker wants. On the other hand, a private registry can also be used, allowing attackers to have a near-infinite arsenal of images to download.

Finally, we assume that the node and guest Pods have no known vulnerabilities and that all security mechanisms work properly.

[1] https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

[2] https://kubernetes.io/docs/concepts/security/rbac-good-practices/

[3] Soft-multitenancy refers to resource sharing and strong trust between the tenants. https://kubernetes.io/docs/concepts/security/multi-tenancy/#isolation

B. Asynchronous API Communication Status

As described in Section II-C, K8s uses an asynchronous communication to enforce the actions in several interfaces implemented in the cluster. For example, each Pod creation is executed independently by K8s, but each node's *containerd* will have the last say in defining the order of execution for each request.[4]

This behavior can cause problems since it is necessary to control the operations' atomicity to guarantee the consistency of the cluster. Furthermore, some operations are part of a bigger dependency tree that must be executed in a certain order. For instance, deploying a container requires the download of its corresponding image to be completed (if not already available).

Since after sending the request, K8s maintains no control over the request execution, it relies on the interface implementation to update the intermediate and final status on the object. However, this does not always happen. The CRI has no API related to the status of image download, and so, the only response *containerd* will provide will be either the download confirmation or some error on the downloading process, like a missing secret or an unavailable tag.

The Kubernetes Enhancement Proposals (KEP) 3542[5] proposes an update to the Kubelet, enabling download status updates to the API, solving the problem previously described. Even with this KEP, the problem is not entirely solved: indeed, it remains impossible to cancel pending downloads. Even using alternative container runtimes such as CRI-O or more secure ones like gVisor or Kata Containers does not change the situation. All these runtimes still rely on the CRI interface, which suffers from the issue described above.

IV. Attack Description

The following section proposes two attacks that exploit the aforementioned vulnerability in *containerd* and the CRI-API, first rendering a Kubernetes node unable to pull any Pod image for an extended period and, secondly, impacting a node's cache, increasing the amount of time needed to deploy an instance of an application.

A. Attack Vector

As mentioned in Section III-B, Kubernetes' API abstracts away several steps that must be taken behind the scenes to create a Pod. When a user creates a Pod through the API (as in Fig. 1), it communicates with the Kubelet to issue a Pod creation request. The Kubelet then communicates with *containerd* via the CRI to trigger the image pull, set up the container, and finally the Pod status is returned to the user. In normal conditions, the API waits for *containerd* to respond with the status of the Pod creation before sending a response to the user. However, the API does not wait for *containerd* to complete pulling the images. In fact, as soon as all required containers for a Pod have been created by *containerd*, the API immediately sends a successful response to the user.

While this choice has its roots in the separation of concerns between Kubernetes and *containerd*, it has a subtle side effect. Without having to notify the API about the status of the image pull, *containerd* is completely unaware of the addition or removal of Pods through the Kubernetes API. With that, a malicious actor can try to use a significant amount of resources by repeatedly instantiating new Pods in the node.

To further strengthen the attack, attackers can use the `--force` flag when deleting a Pod. As discussed in Section II-C, the `--force` flag automatically removes all API bindings for the resources, which means that the Pod will be deleted immediately from the cluster. On the other hand, it will not interrupt any image download that has already been started by *containerd*.[6]

Thus, after deploying a Pod an attacker can force delete it shortly after, and avoiding a lengthy wait for the image downloads to complete. This process can be repeated a potentially infinite number of times, in quick succession. By doing so, an attacker can clutter the *containerd*'s download queue with useless image pull requests as long as they have image tags available to pull. Since *containerd* is agnostic to the Pods' existence and Kubernetes is unaware about what images are being downloaded, the status quo can remain indefinitely. As such, the target container runtime will be continuously forced to keep track of and download unnecessary container images on behalf of the attacker.

B. Attack Requirements

The attack requires minimal prerequisites to be executed. First, it needs access to a cluster with standard user privileges and the capacity to create and delete at least one Pod. Any limitations imposed by the RBAC configuration or Pod Security Standards (PSSs) are irrelevant to its execution. Indeed, since the vulnerability resides in the CRI interface, any hardening or security measures applied to abstraction layers above the CRI, or changes to runC through hardening solutions such as Kata Containers or gVisor, will not prevent the attack.

Quotas are also easily bypassed: the minimum requirement is to be able to create and delete a single Pod in a single namespace. K8s' Quotas are designed to limit the number of objects that can be created in a namespace by a single user, preventing resource exhaustion attacks and ensuring fair resource distribution among users. However, since the attack relies on the constant deletion and recreation of Pods, a Quota of just one Pod is sufficient to execute the attack.

Second, the attacker requires a set of Docker images, which can be either different tags of a single image or entirely different ones altogether. They also need to be in a registry accessible by the node. Furthermore, the least layers the images have in

[4]The CRI specification does not mandate a specific logic for handling the order in which image pull requests are served, leading to possible differences between container runtimes.

[5]GitHub issue available at https://github.com/kubernetes/enhancements/issues/3542

[6]At the time of writing, a bug exists in K8s that additionally prevents the deletion of associated containers when a Pod is force deleted. See https://github.com/kubernetes/kubernetes/issues/119276. The bug is not planned to be fixed.

common, the better, since re-occurring layers will not be re-downloaded.

The content, structure, functionality, and size of the image are irrelevant for the attack to be successful. Their only limitation is that the total image size cannot exceed the limits imposed by the cluster administrator, because the download will be blocked before starting. Thus, attackers can use any image they have access to, including public images from Docker Hub or private images from their own registries, as long as they are accessible by the node. Such images either need not to be present on the target node, or their download must be forced using the `imagePullPolicy: always` parameter.

On the other hand, the number of images available to the attacker is critical for the attack's success. Since the image (or any of its layers) must not be present on the attacked node, the more available they are, the longer the attack will last. This must also scale up with the amount of storage available on the node.

C. Denial of Resources

This attack will generate a denial of resources in the node, and its impact is twofold. First, the endless cycle of image pull requests and deletions prevents the deployment of new containers without a cached image. This problem is exacerbated by the growing trend of running short-lived microservices: indeed, 72% of containers live fewer than five minutes [15]. An unresponsive cluster can be extremely problematic, even if the attack is short-lived: clusters that rely on short and fast batch jobs (e.g. Continuous Integration/Continuous Delivery (CI/CD)) will inevitably be delayed. Indeed, this DoS does not render a node actually *unavailable* for K8s: the scheduler will still see the node as healthy and will continue to schedule Pods on it, nor any Pod on it will be evicted.

Second, the attack drains the available resources on the worker node by forcing it to constantly utilize disk, bandwidth, and CPU to retrieve container images. As a result, other workloads hosted on the node may suffer a performance setback, possibly violating Service Level Agreements (SLAs). Since the Kubelet usually runs as a process in the node and it is not directly controlled by the quotas and limitations imposed by the Kubernetes, it will use 100% of the resources available to download and decompress the images, and by default, there is no way to limit this resource access. Furthermore, pinpointing the resource starvation to the attack is not straightforward, as the *containerd* process will appear as a busy process in the node, and the API will not provide any information about the ongoing image downloads.

Ultimately, the effectiveness of the denial of service will depend on the amount of resources available in the node and the amount of nodes in the cluster. Generally, K8s worker nodes have a limited size, and the K8s philosophy promotes smaller but more numerous nodes, enabling horizontal autoscaling and a better failure tolerance. Additionally, K8s is increasingly deployed in Edge environments, where nodes are often resource-constrained and have no redundancy. In these scenarios, the attack can be particularly effective, as it can quickly exhaust the available resources. On the other hand, larger nodes with more resources can withstand a higher number of image pull requests, but the attack can still cause a noticeable performance degradation.

Nonetheless, the amount of nodes in the cluster also has a direct impact on the attack. When attackers target a single or a few nodes, it can be easily attributed to a local malfunction, and its resource usage will be hard to pinpoint in an aggregated dashboard. On the other hand, executing the attack on many or all nodes in big clusters will likely cause a noticeable increase in resource usage across the cluster, making it easier to identify.

Finally, the possibility of a DoS is increased in multi-tenant scenarios such as Liqo [16], where resources are shared across clusters. As an example, a cluster may lend a subset of its resources to another cluster, allowing the latter to run workloads on it. To the renting cluster, this shows up as a local, virtual namespace, while Pods are actually running remotely. In this case, an attacker can target the shared namespace, resulting in a denial-of-service attack on the remote node(s).

D. Cache Manipulation

Since the images are not deleted after the download, they can be used to manipulate the cache in a given node. The Garbage Collector (GC)'s default configuration sets a series of thresholds, the most important ones being `ImageGCHighThreshold-Percent`, used to determine when the GC will be triggered, and `ImageGCLowThresholdPercent`, the minimum target percentage that the GC wants to achieve after starting to delete images. The default settings for these two thresholds are 85% and 80%, respectively.

After surpassing the upper threshold, the Kubelet marks images present on disk for more than two minutes (by default) as eligible for deletion, and starts removing them until at least the lower threshold is reached. It must be noted that the GC may keep deleting eligible images even after surpassing the lower threshold. This can be a problem in scenarios where the cluster hosts cron jobs, FaaS tasks or any application that does not have a container continuously running. Since their corresponding images will be deleted, these tasks will experience delayed execution, as *containerd* will be forced to download the images again.

However, K8s does not just limit to cleaning up unused images and containers. Another key part of the system is the Node-pressure Pod eviction system,[7] which is independent from the GC. When a node's resources reach dangerously low thresholds, the Kubelet intentionally terminates Pods to avoid possible contention and crashes. At the time of writing, hard eviction is handled by the `DefaultEvictionHard` option, whose default value is 90%. Once the threshold is reached, it will delete both Pods and their corresponding images, taking care in deleting as few of them as possible. This is meant to be an extreme measure, and usually causes further disruption on the node due to the critical lack of disk space and increased CPU usage.

[7] https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/

E. Responsible Disclosure

The attack presented in this work has been responsibly disclosed to the Kubernetes community. We opened a public issue to discuss the attack and its implications.[8] The issue was acknowledged by the Kubernetes maintainers, but as of the time of writing, a solution has not yet been implemented in the codebase. Still, the maintainers have confirmed that the attack is not a security vulnerability, but rather a resource usage bug, spurring a discussion and several pull requests on how to address it in future versions of Kubernetes.

V. Experiments and Results

In this section, we present a summary of our findings after running a series of experiments. We evaluate the effectiveness of the attack on a real K8s cluster and additionally analyze the effect of the attack on some key system resources.

A. Experimental Setup

To evaluate the attack, we conducted a series of experiments on a K8s cluster. The goal was to assess the attack's impact on resource usage, scheduling delays, and service interference. We also wanted to verify the attack's effectiveness in manipulating the local image cache, and finally, to confirm the attack's applicability in public cloud environments. All necessary files to reproduce the experiments presented here are available in the MAGI GitHub repository [17].

1) Local Testbed: We set up a local testbed on our premises to evaluate the attack. On a machine equipped with an Intel Xeon Silver 4112 CPU (2.60 GHz) and 64 GB of RAM, we created three virtualized nodes with 2 vCPUs, 4 GB of memory, 120 GB of storage capacity (HDD), a 1 Gbit/s network interface, and Ubuntu 22.04 LTS as the operating system, to run as nodes in a K8s cluster: one for the control plane and two as worker nodes, using K8s v1.27 and `containerd` v1.6.8 as the container runtime. We also used a separate physical machine hosting a private Docker registry.

2) Public Cloud Testbed: To further confirm our findings in bigger and more realistic scenarios, we also conducted some experiments in a public cloud environment. We used Google Cloud Platform's GKE service, allowing us to assess the attack's impact on a larger scale and in a more production-like environment. We used `e2-standard-2` machines with 2 dedicated AMD Epyc vCPUs, 8 GB of RAM, a 60 GB balanced persistent disk, and a 10 Gbit/s network interface. We deployed a total of ten worker nodes, while the control plane was provided by GKE. The nodes were configured to use Ubuntu 24.04 LTS as the operating system and `containerd` as the container runtime. No particular settings were enabled during the creation of the cluster.

3) Image Sets: To ensure the experiment's effectiveness and consistency, we decided to build our own set of Docker images for the attack.

TABLE I
Summary of Layers, Size, and Other Characteristics of the Two Image Sets Used in the Experiments. ∗The Total Size Includes the Base Image Only Once, as It is Not Duplicated When Generating an Image. The Compressed Size Takes Into Account `pigz`'s Compression Factor of 0.504.

Image set	**Variable GB**	**Variable MB**
Tag count	7	40
Lower-bound uncompressed size	2.12GB	0.10GB
Upper-bound uncompressed size	14.40GB	0.90GB
Total uncompressed size*	57.12GB	16.80GB
Total compressed size*	28.88GB	8.47GB

Each image in the set contains a base Ubuntu layer (of around 80 MB uncompressed), which is shared between all the images, and a variable number of layers, created with the `od` command and the `c` option. We performed some experiments on `gzip`'s compression options, and the results are available in a separate report [18].

The first set, called **Variable GB**, contains a total of seven images. Each image contains one to seven additional layers of 2 GB each. As a result, each image in the set ranges in size between 2 GB and 14 GB. The second set of images called **Variable MB**, contains forty images, with each additional layer containing 20 MB of data. This results in images ranging between 100 MB and 900 MB. These sizes refer to on-disk usage, as when fetched from a registry, they will be compressed, decreasing their size by roughly half. Table I shows a summary of these two sets.

B. Denial of Resources and Scheduling Delay

We first focused on assessing the denial of resources capabilities of the attack. In particular, we investigated to what extent the Pod scheduling can be delayed and how the CPU and I/O are affected. These tests were performed on the local testbed.

1) Scheduling Delay Metric: To properly compare the different lengths of the attack and the actual impact on the node image pulling, we created a metric that would normalize the results. We call it **Scheduling Delay**. It is measured in s/GB and is calculated as follows:

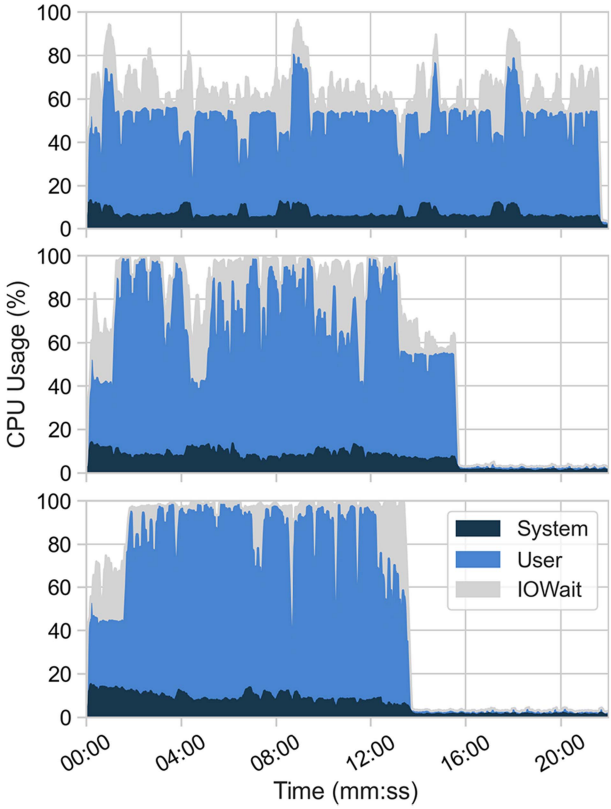
$$SD = \frac{D}{S_c}$$

Where D is the duration of the attack (in seconds), and S_c is the total amount of compressed data that the node had to withstand from the beginning of the attack. This metric is unrelated to the impact on system resources and gives us a rough idea of how efficient is a certain attack configuration in preventing a node's downloading of new Pod images.

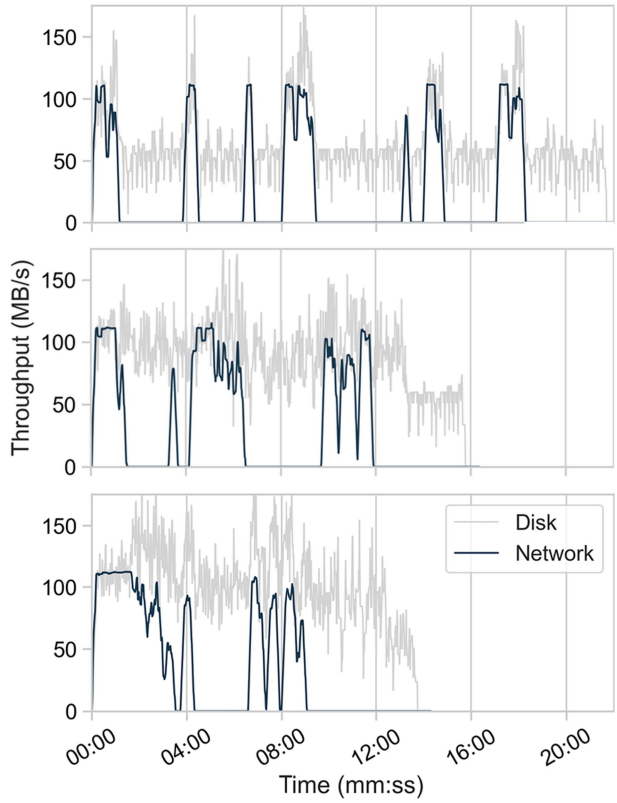
2) Max Parallel Image Pulls: Starting from version `v1.27`, K8s allows users to specify the maximum number of images to be pulled in parallel;[9] indeed, in the past, images were

[8]GitHub issue available at https://github.com/kubernetes/kubernetes/issues/122905

[9]GitHub issue available at https://github.com/kubernetes/enhancements/issues/3673



(a) CPU usage by type with mp set to 1 (top), to 2 (middle) and to 4 (bottom).



(b) Disk write and network read with mp set to 1 (top), to 2 (middle) and to 4 (bottom).

Fig. 2. CPU usage by type (on the left); disk write and network read (on the right) of the system during the variable GB scenario. Data were taken from a randomly sampled trial.

exclusively pulled serially. While this feature helps improve the efficiency of the cluster by making full use of the available bandwidth, we want to investigate its effects on the attack since it would prevent the cluster from being stuck in a given download. However, this option would probably increase resource consumption.

In the experiment, we use the values 1, 2, and 4 of the `MaxParallelImagePulls` flag. This flag will be shortened to mp in the upcoming sections for brevity.

3) Results: Variable GB set: We first run the experiments of the Variable GB image set on the local testbed. Before each experiment, we reshuffle the images. Then, for each image, we deploy it, wait two seconds, and then delete it with the `--force` flag. Additionally, to verify the attack's behavior with the parallelism provided by the mp flag, we run each experiment with the values described before. We repeat this process thirty times for each setting; the final result is an average of these runs. To better visualize the results, we chose a random trial from the thirty and plotted the CPU usage, broken down by type, and the disk write and network read. These plots are available in Fig. 2(a) and (b).

With mp set to 1, the results are staggering. Even with a small attack, no parallelism, and just seven images, the impact on both worker nodes is substantial, and clearly could be categorized as a DoS. Every new download on the node is blocked for a total of 1352 seconds, almost 23 minutes; the CPU usage averages 65% with spikes of almost 90%, and the disk is kept constantly busy, with I/O write spikes of 150 MB/s. From Fig. 2(a) (row 1), we can see how these CPU peaks are noticeable and correlate with the download of the images (Fig. 2(b)), and thus with a high bandwidth and I/O usage. On the other hand, when images are decompressed, the CPU usage drops significantly but remains steadier, as does the disk usage.

As we increase mp, the total time of the attack decreases, and the resource usage increases accordingly. For example, setting mp to 2 reduces the attack time to 909 seconds, while with $mp = 4$, it further reduces to 887 seconds. On the other hand, the CPU usage increases, while the difference in the peaks becomes less and less apparent as the various parallel downloads compete for CPU and I/O time with the ongoing unpacking tasks. With four parallel images, the CPU usage reaches almost 100% for most of the attack time.

By calculating the Scheduling Delay for these three tests, we see how it also drops from 46.82 ($mp = 1$) to 31.48 ($mp = 2$) and 30.72 ($mp = 4$) s/GB. It is important to observe the difference between the former two settings, where the SD drops 33%, and the latter, where the SD drops again but by a smaller margin (only 2.4%). While introducing parallelism strengthens the attack and puts an increased load on resources, at a certain point, the returns start diminishing as contention is introduced, and the cluster's overall performance starts degrading. Thus, we believe that increasing mp without implementing a solution to this threat could be even more detrimental.

TABLE II
SCHEDULING DELAY (S/GB) AVERAGED OVER 30 TRIALS WITH STANDARD DEVIATION. THE HIGHER, THE MORE DISRUPTIVE THE ATTACK.

Set	Scheduling Delay		
	mp= 1	mp= 2	mp= 4
Var. GB	46.82 ± 0.37	31.48 ± 1.02	30.72 ± 0.72
Var. MB	56.93 ± 1.59	41.10 ± 1.16	39.92 ± 1.60

TABLE III
AVERAGE CPU USAGE OVER ALL CORES OF A WORKER NODE DURING THE ATTACK FOR EACH SCENARIO. FACTOR OF 100, AVERAGED OVER 30 TRIALS.

Set	CPU Usage		
	mp= 1	mp= 2	mp= 4
Var. GB	67.31 ± 0.81	91.67 ± 2.24	93.22 ± 2.23
Var. MB	64.14 ± 0.35	84.35 ± 0.62	85.78 ± 2.69

Variable MB set: We now repeat the same experiments on the Variable MB set. Such results are similar to the ones we obtained with the Variable GB set, but with a much lower runtime due to the overall smaller size. Thus, we calculate the Scheduling Delay metric, presented in Table II, and the average CPU usage over all cores, presented in Table III, to properly assess the consequences of using smaller-sized images in the attack.

From the Scheduling Delay results, the immediate impression is that using smaller images is indeed more disruptive. With $mp = 1$, the Variable GB set hangs the worker nodes for 47 seconds per GB. On the other hand, the Variable MB manages to block it for 57 seconds. Thus, we can block the downloads for 18% more time with small images. With $mp = 2$ and $mp = 4$, this becomes more evident, arriving at 23% more with small images. We attribute this increase to the overall overhead that *containerd* must withstand when downloading images. Indeed, each new image implies fetching its manifest, obtaining the list of layers, and manually downloading each of them: all operations repeated many times start piling up.

On the other hand, Table III presents almost opposite results. While with no parallelism, the difference is almost unnoticeable, as mp is increased, the Variable GB set is far more CPU-intensive, using almost 10% more CPU on average than the Variable MB set. We attribute this lower average CPU usage to the relative weight of downloading and unpacking a smaller image. As a consequence, the possibility of saturating the CPU is lower unless the parallelism is way higher than in our experiments.

C. Service Interference

Another goal of the attack is to affect the Quality of Service (QoS) of other running services in the node. We performed two experiments on the local testbed with the same attack, but this time, we verified how this affects CPU and I/O usage of other Pods.

To do so, we employ the Phoronix Test Suite [19], an automated benchmark suite that allows users to perform a wide range of performance assessments on their system. From the wide variety of benchmarks available, we choose the `build-linux-kernel`[10] and `stress-ng`[11] benchmarks, two of the most popular of the suite. Table IV presents the results and the number of trials for each of them. The first assesses the time needed to compile the Linux kernel (version 6.1) on the machine, putting an I/O and CPU workload on the system. The latter evaluates the amount of operations per second the CPU can perform. To ensure the statistical relevance of its results, Phoronix automatically performs additional trials if it encounters performance issues or suspiciously high variability between different runs.

First, we run the `build-linux-kernel` benchmark with no attack running on the cluster. In this case, it completes in 853.25 seconds, averaged over three runs. Then, we apply the attack as described in Section V-B with the Variable GB set and mp set to 1 (as parallelism is not a concern in this setting). We chose this particular scenario as it is the longest and, thus, the most likely to last the entirety of the benchmark. We therefore re-run it, applying the attack in the background and cleaning up after every trial. With this setup, the build time increases to 1630.04, a 91% increase over the non-disrupted one.

The other benchmark also reports significant results, with the `stress-ng` measuring 2185.83 bogo-ops/s[12] with no attack running over three trials. Repeating it with the attack, Phoronix takes 15 trials to obtain 1463.18 bogo-ops/s, a drop of 33% with regards to the non-attacked scenario. Additionally, the standard error is 12.52 seconds, almost ten times higher than the 1.29 seconds of the non-attacked benchmark. This 33% drop is less severe than the one from `build-linux-kernel`, mainly due to the fact that `stress-ng`'s CPU-bound behavior clashes more with the attack, introducing additional contention.

D. Cache Manipulation

We then verified the attack's capabilities in interfering with another kind of system resource: storage. In particular, we analyze how the attack can manipulate the local image cache on the local testbed.

As previously described in Section IV-D, K8s is equipped with a local image cache which is periodically expunged by a GC should the used disk space surpass the `ImageGCHighThresholdPercent` (85%) threshold. Thus, we aim to investigate whether the attack can induce cache evictions as a side effect. Such a scenario would mean that the attacker can have a way of indirectly controlling the cache. In addition, it would further disrupt the node's operations as the I/O and CPU overhead of the cache eviction is added to the underlying attack.

We therefore rerun the Variable GB experiment with mp set to 1, as we are not interested in seeing the effects of parallel downloads in this case. With a simple script, we monitor the images present in the cache and the disk space. However, this

[10]Specifications available at https://openbenchmarking.org/test/pts/build-linux-kernel-1.15.0. The "defconfig" option was used.

[11]Specifications available at https://openbenchmarking.org/test/pts/stress-ng. The "cpustress" option was used.

[12]Bogus operations per second, a measure used by `stress-ng` to evaluate processor performance

TABLE IV
SUMMARY OF THE RESULTS OF THE BENCHMARK ON THE NON-DISRUPTED AND ON THE ATTACKED CLUSTER. bo/s STANDS FOR BOGO-OPERATIONS/SECONDS.

Benchmark	No attack		With the attack	
	Results	Trials	Results	Trials
`kernel`	$823.25 \pm 0.41\ s$	3	$1630.04 \pm 5.47\ s$	5
`stress`	$2185.83 \pm 1.29\ bo/s$	3	$1463.18 \pm 12.52\ bo/s$	15

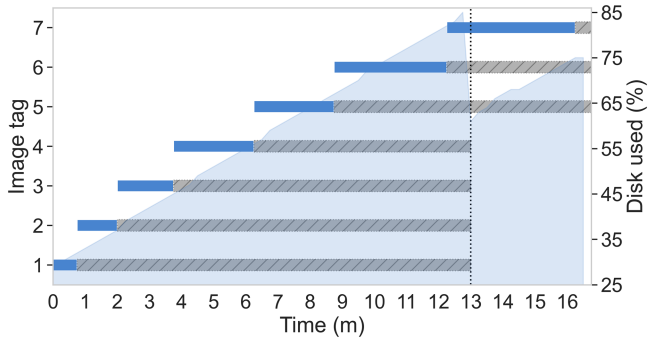


Fig. 3. Sequence diagram of downloaded images (left axis) and used disk space (right axis, shaded area) during the sequential image download. Dark lines represent active downloads; grey lines represent image locality (i.e. image finished downloading and present on disk).

time, we run the attack sequentially rather than randomly to eliminate variability. In other words, first we deploy the smallest image (2 GB), delete it with `--force`, deploy the second smallest, and so on. We repeat the experiment several times, obtaining varying results. An example result is shown in Fig. 3.

Indeed, we discovered that the GC is extremely aggressive, and once the 85% threshold has been reached (at $t = 13$), it starts deleting all qualifying images that are eligible for deletion. In our case, during the download of the seventh image the GC deleted images numbered 1 to 4 at once, dropping the disk usage to 62%. Since the minimum Time to Live (TTL) of images downloaded to disk is two minutes, the sixth image was not removed. On the other hand, the fifth image was spared for no reason, and in other trials, it was also deleted.

Either way, this greedy behavior by the GC, while beneficial for the disk pressure, further bolsters the capabilities of the attack. Since in the background more images are being downloaded, saved to disk and unpacked, triggering the deletion of several images at once causes further impact on system resources and potentially deletes images that may have been needed for short jobs by other tenants.

On the other hand, we obtained a slightly different result by avoiding to delete Pods at all. Doing so prevents the GC from deleting any image — as they are all in use — and forces it to wait for the eviction manager to delete at least one Pod. Consequently, the disk usage eventually reaches 90%, triggering the eviction manager. As it evicts Pods, the GC also deletes their image shortly after. This overlap of both the eviction and GC causes even more mayhem on the node, heavily impacting CPU and storage I/O.

E. Applicability to Public Cloud

To further confirm our findings, we conducted additional experiments in the GKE public cloud scenario, aiming to determine if the attack was feasible in larger and more realistic scenarios. First, we replicated all the experiments described so far in this section. The results were consistent with those obtained in the local testbed, but due to space constraints, they can be found in the repository. Second, to investigate whether increasing the number of nodes affects the attack's capabilities, we crafted a new scenario using a set of real, Machine Learning (ML)-oriented images and a slightly different attack procedure. Additionally, information about the setup can be found in the repository.

In this scenario, instead of creating and deleting Pods, we first create a Deployment. The Deployment is configured to have eight replicas, one for each node in the cluster, using affinity features to achieve so. Since the images are large, we wait several seconds after the first creation, and then we patch the Deployment, changing the image. As a result, the underlying ReplicaSet is substituted, triggering a process in which old containers are deleted, and new ones are spawned with the new image, thus requiring a new download from the container registry. This process realizes the same effect as the previous attack with the `--force` flag, but without actually using it. We also validate whether Falco can identify this new approach. Yet, with the default set of rules, the only `INFO` alert printed is the one related to the first Deployment creation, since ReplicaSets are not observed by Falco.

This attack proves, in a production scenario, that all the characteristics explored in the local testbed (the scheduling delay, resource usage, and cache manipulation) are still valid. With just a single alert triggered, an attacker could perpetually edit the Deployment, keeping the node(s) under constant pressure. The CPU usage over all nodes is still high, averaging 91% over the attack duration. Each node reacted differently to the various download requests: unfortunately, some images were never downloaded by some nodes. Yet, all of them were overall delayed for 18 minutes on average. The GC could have also been triggered if we had increased the number of images. Thus, we can conclude that the attack is indeed feasible in a public cloud environment. Attackers wishing to make it stealthier could reduce the amount of replicas or use specific nodeSelectors, achieving a similar effect. The only required permission is the ability to create and patch Deployments, and the number of nodes that can be affected is limited by the amount of replicas one can create.

VI. MITIGATION DISCUSSION

This section presents the rationale about how the problem can be solved in the Kubernetes, *containerd*, and CRI-API code, the difficulties in implementing the solution, and, as an alternative, we present a proof-of-concept solution, using eBPF until all

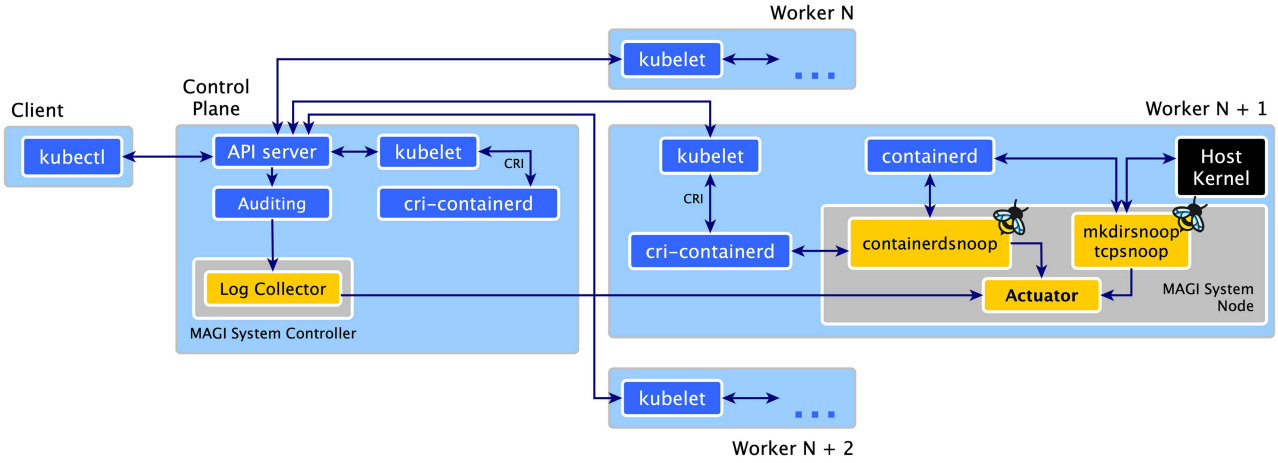


Fig. 4. Architecture of the MAGI system. Dark components are part of K8s; light components are part of MAGI. Arrows represent communications and data flow.

the necessary implementations have been released inside the respective projects.

A. Kubernetes Status and Best Approach

To some extent, the limitations of the CRI-API in controlling/reporting the image downloading process are already known by the community. As mentioned in Section III-B, KEP-3542 has already been assigned to implement an image progress notification. Open since 2019, the reasoning behind this proposed enhancement is to bring more information to the administrator, reporting the current speed and download status from each image. Nonetheless, the threat previously described will not be solved by this implementation since the progress notification will not enable the termination of the downloading process if a container has been marked for deletion.

On the other hand, KEP-3673 outlines the Parallel Image Pulls option (outlined in Section V-B2) and its role in preventing cluster saturation due to prolonged image downloads. We show in our experiments how, in reality, this feature can actually increase the effectiveness of the attack: with more images to be downloaded concurrently, the attack will consume more resources.

As discussed in Section III-B, the problem is the lack of communication status routes in the CRI-API, because there is no status message between the user's request (create or update) and its completion. The difficulties to completely solve this issue are threefold. First, ensure that the *containerd* maintains the context open during all the command executions. Second, implement a series of new API routes to guarantee that the CRI can cancel the submission or the download of each requested image to the runtime. Finally, the Pod controller must also be adapted to include these new routes, maintaining the status of the requests, validating if the image download has already started, and deciding if it needs to stop after a delete request.

Unfortunately, these implementations are not trivial and depend on several interactions with different Special Interest Groups (SIGs) and projects with distinct schedules and contribution methodologies. We are already collaborating with them to solve these issues. However, the implementation process is usually very long and laborious. Therefore, we developed a temporary mitigation solution that is presented in the following subsection.

B. MAGI System

As a stopgap and proof-of-concept solution, we propose the **MAGI System** (Mitigation AGainst Images) [17], whose architecture is shown in Fig. 4. MAGI is fully deterministic and is written in eBPF, Python, and Bash. It monitors the cluster status and image downloading requests, and uses kernel-level tools to terminate offending image downloads whether an attack matching our pattern is detected.

The MAGI System is composed of two main components: the **controller component** and the **node component**. The controller component, as the name suggests, must run on the control plane node of the cluster; node components must be run on each node of the cluster. This includes the control plane itself, if users wish to schedule Pods on it.

MAGI can be deployed in any K8s cluster, regardless of the container runtime being used. It requires minimal effort to be set up, with the only requirement being the enabling of K8s' auditing feature and root access to the nodes. Once deployed, it runs in the background and does not require any user interaction.

Controller component: The controller component leverages Kubernetes's auditing capabilities to keep track of the Pod creation and deletion requests, which are handled by the API server. As a pre-requisite, thus, it requires enabling Kubernetes's Auditing feature,[13] which by default is disabled. The Auditing features lets cluster administrators specify with great flexibility what to monitor, for example user activity, API usage, and control plane events. For our purposes, we limited the scope and verbosity of the logging to only the `pods/*` resource group. More precisely, we captured all API calls related to the Pods group and subgroups, logging event metadata. In the case of elaborate topologies with multiple control plane nodes, it is

[13]https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/

sufficient to place the controller component in a node in which the auditing capability has been enabled.

Once the Auditing is properly set up, the controller component can be started. During regular usage, it will monitor the API server's calls for Pod creations, keeping a cluster-wide track where each Pods have been scheduled and if any image pull is necessary for them. Once the image has been downloaded and the Pod has been successfully deployed, the component discards the collected information.

When the component instead detects a Pod deletion request, it checks if the Pod had previously requested an image pull which has yet to complete. Such an event could potentially signal an attack attempt. In this case, the controller component promptly contacts the node in which the Pod was scheduled, informing it that the tracking image should not be downloaded anymore.

Node component: The node component is responsible for several concurrent tasks, and works with three threads. The first is in charge of keeping track of the images being downloaded by *containerd*; the second monitors the API calls being made by the Kubelet to *containerd*; the third listens for the alerts emitted by the controller component, eventually performing corrective actions. Each instance of the node component operates independently from the others.

Image download queue monitoring: The first thread keeps track of `crictl`'s local image download queue. This information is fed to the script by `containerdsnoop` [20], an eBPF program capable of intercepting gRPC calls to and from *containerd*; in particular, calls from the ImageService (`crictl`) and the RuntimeService (Kubelet).

Within the implementation, gRPC calls are not encrypted, but the `HTTP2` `HPACK` header compression mechanism requires access to the initial packets of connections. Thus, before the startup, the Kubelet instance is deliberately restarted, allowing our tool to fully parse the calls. While the program can parse almost every API call made to *containerd*, we narrowed down its scope for our tool, deliberately snooping only requests and responses to the PullImage API route.

When the tool detects a request, the payload of the call is inspected and the requested image is appended to the queue. When a response is detected – signaling a successful download – the said image is removed from the queue. The tool is able to distinguish between Docker Hub and other-registries images, and maintains the ordering of the download queue even when the `MaxParallelImagePulls` parameter is set to a value greater than 1.

System call log monitoring: A second thread is tasked with monitoring the system calls being made by *containerd*. Once we know that a certain image is being downloaded, we also need to identify which connection is being used to perform the download of which layer. Unfortunately, *containerd* does not expose such information natively. However, we fortunately noticed that *containerd*'s behavior is predictable.

When *containerd* downloads an image, it first opens a connection to the registry, retrieving the manifest file and obtaining the list of layers that compose the image. Then, for each layer, *containerd* spawns a thread that creates a folder on the disk for storing the downloaded files, then immediately after binds to a network socket. By default, *containerd* opens a maximum of four sockets at a time[14] when downloading images.

To properly log this information, we used a custom-made eBPF-based tool [21] that specifically monitors two selected system calls. In particular, we attached eBPF kprobes to the `mkdirat` and `tcp_connect` system calls being performed by *containerd*. The first is related to the creation of folders containing the layer data on disk, and the latter to opening the connections to download the data in question. Since the two steps are performed by the same *containerd* thread, by correlating the two calls, we can identify the mappings between *containerd* threads, layers, and open ports. This effectively allows us to understand which connection is being employed to download which layer. The two tools continuously fetch new data and push it to userspace using a `perf_buffer`, and do not introduce any exploitable functionality since they only trace events and forward that data to userspace.

Alert listening and actuator: Finally, a thread is tasked with listening for the controller's alerts. When such an alert arrives, the component promptly reviews its local image pull queue: if the requested image is currently being downloaded, the component terminates the connection between the node and the registry, interrupting the download and clearing the local cache.

This is done first by querying the `bbolt` database[15] and obtaining the list of layers from the name of the image. Alternatively, the component can also query the registry directly, obtaining the list of layers from the manifest file. From each layer in the list, the port number of the connection downloading that particular layer is obtained. Finally, each of the offending sockets is terminated using `ss -K`. Since the Kubernetes API has already deleted the Pod and freed up its resources, it will not attempt again to download the image, thwarting the malicious actor's attack. Clearly, if the attacker were to try again to download said image (or any other one), the system would get triggered again, thus being able to continously take down attack attempts over time.

To account for pending downloads, we put offending images signalled by the controller component in a blocklist. This behavior is triggered when an image download is requested but one or more images are in the queue before it. As the queue empties, the script automatically detects if a new download matches an image in the blocklist and terminates it accordingly.

C. Evaluation

To evaluate the effectiveness of our mitigation approach, we devised a series of tests that evaluate the functionality of the system in different scenarios: without the attack, with the attack, and with the attack and the mitigation running.

Each scenario comprises the deployment of a series of images in a K8s cluster at pre-defined intervals. We selected some common images with different sizes, to represent a common

[14]This option is customizable within *containerd*, and is unrelated with the `MaxParallelImagePulls` K8s flag, which instead refers to *parallel images* rather than *parallel layers*.

[15]`bbolt` is a fast on-disk database used by *containerd*.

cluster scenario. While in real clusters workload might be scheduled more irregularly and unpredictably, we devised a series of ordered commands with a constant wait time between each other, allowing us to accurately assess the behavior of the cluster and our solution while minimizing external interferences. Since the mp parameter is not the focus of this experiment, we set it to 1.

In particular, in each scenario we perform the following steps. First, to emulate the deployment of a large-sized image, we deploy `sagemathinc/cocalc` in background and wait 20 seconds to give K8s sufficient time to complete the deployment and propagate the changes. Then, depending on the setting, we either start the attack (using the variable GB image set) or do nothing, and we wait 20 further seconds. Finally, to simulate the download of small images, we deploy `jupyter/scipy-notebook`, `nginx`, and `online-boutique` [22], waiting 20 seconds between each other. In particular, `online-boutique` is a collection of twelve lightweight microservices deployed as separate containers.

We repeat these steps for each of the aforementioned scenarios. In each one, we measure the total length of the test, the moment at which every deployment was requested and the moment at which the container was actually started. This information is collected using the in-built K8s APIs.

In the first scenario, the total time spent by the test is around eleven minutes. Of these, eight of them are spent downloading the `cocalc` image, which is extremely large; once the download is completed, the other images follow suit quickly. On the other hand, in the second scenario we deploy the attack after the first twenty seconds, thus queuing it after the `cocalc` download. Once `cocalc` finishes downloading, all the following requests are brought to a grinding halt as the attack starts saturating the CPU and bandwidth of the worker nodes. This increases the length of the test to around twenty-nine minutes. It is important to note that the attack does not affect the net download time of the images; rather, it makes them stall and wait until their turn arrives. Finally, in the third scenario, the MAGI System successfully intercepts and blocks six out of seven images, i.e., all images but the smallest one. As the downloads get interrupted almost instantly, the total length of the test drops back to eleven minutes, the same as the first scenario.

These tests highlight the capabilities and low impact of our solution. First, we observed how MAGI has almost no footprint. The CPU usage in nodes spikes at 0.8% during detection and almost 0% when sitting idle. The memory footprint was more evident, ranging from 380 MB idle to almost 420 MB during the attack, but remaining mostly stable nevertheless. We attribute this usage to Python's memory management and the need of storing status for several different metrics at once. On the other hand, both CPU and memory usage were insignificant on the control plane node, with the only discernible overhead being K8s' auditing feature.

Wishing to further investigate why MAGI failed to block the smallest image of the attack, we performed a series of tests with a set of smaller images, starting from 83 MB (42.5 MB uncompressed) and adding a new 5 MB layer at each step. With MAGI running in the background, we deployed and deleted a Pod with the image, waited for the download to complete, purged the node's image cache, and then repeated the process with the next image.

We observed that the cutoff at which MAGI was able to block the download was around 250 MB, which on a 10 Gbit/s network takes around 2 seconds to download. MAGI's eBPF code identified all image downloads and their corresponding layers. However, validating all the steps required to delete only rogue downloads takes a non-negligible amount of time. This delay does not depend on the image size or bandwidth, but rather on the exchange of information between various distinct services, resulting in an almost linear relationship with both the image size and bandwidth. Finally, since MAGI can identify all downloads, even if it cannot always delete them, future improvements could include generating alerts for the administrator when the number of image downloads on a node surpasses a given threshold within one minute.

D. Discussion and Limitations

While MAGI demonstrates a significant improvement over the current state of affairs, it is does not come without drawbacks. First and foremost, the attack surface of the cluster increases significantly when MAGI is deployed. This is due to several factors.

First, the K8s auditing feature – which is not enabled by default – exposes sensitive information about the cluster's state and operations. Second, MAGI needs to monitor the gRPC calls between the API server and the container runtime. Doing so requires restarting the Kubelet, which can disrupt the cluster's operations. Finally, the MAGI client requires root access to the nodes in order to intercept the system calls and kill the offending sockets. All of these extra requirements increase the attack surface.

In addition to the security risks, MAGI also has some limitations in terms of performance and usability. As shown above, MAGI struggles in blocking the download of very small images. Indeed, the time required to perform heavy tasks such as the domain resolution, manifest retrieval, and socket termination can exceed the time required to download the image itself, if it is small. MAGI is also inherently fragile, relying on code and requirements that may break between Linux, K8s, and *containerd* versions. In our opinion, the sheer amount of information required for gracefully terminating a download without disrupting any service, the inherent unpredictability of the system scheduler and the relative opacity of *containerd*'s download management render any further attempts in increasing MAGI's speed better spent in improving the main problem instead.

VII. Related Work

Denial of Service based on Resources Usage: Resource-based DoS in Cloud Computing was previously covered in previous works. In [23], the authors present a novel class of attacks called Resource-Freeing Attacks (RFAs) and create a methodology to prevent the DoS by increasing the amount of resources available to the victim VM. Fang et al. also discussed resource and co-location attacks in [24] and [25], where they proposed respectively one new metric, called Heteroscore, and a threat method,

REPTTACK. In the latter, a mitigation technique is also proposed, based on introducing randomness in the scheduling of the Virtual Machine (VM)s. However, both works do not consider container-based deployments, such as K8s. Finally, in [26], Zhan et al. show a framework that aims to detect CPU-exhaustion DoS attacks in containers called Coda. However, it does not consider the node resources, just the container ones. Finally, some CVEs related to resource-based DoS attacks in K8s have been reported, such as CVE-2022-1708,[16] and CVE-2025-0426[17]. Both CVEs deal with node resource exhaustion, the first one exploiting a CRI vulnerability causing large files to be written to disk, the second exploiting unauthenticated requests to cause a DoS condition in the node.

Kubernetes and Container Security: Ahmed [27] shows that the container deployment process can generate significant resource usage, mainly memory and CPU. The author also tested several alterations to the container deployment's decompress process, focusing on the parallelism of several simultaneous layers. Such an approach decreases the total time to deploy a given image but inherently increases the resources used in this process.

Additionally, two studies investigate vulnerabilities in the configuration of various services within the Kubernetes orchestrator. He and Guo [3] show a threat in the resource configuration, where using highly privileged configurations allows eBPF applications to escape containers and access sensible information. Xiao [4], instead, found several threats based on the communication between microVMs solutions — such as Firecracker and Kata Containers — and *containerd* to escape attacks and resource-based DoS.

Another important area that has received recent attention from the container orchestration community is anomaly detection in cluster resource usage. Almaraz-Rivera [5]'s study highlights the importance of implementing monitoring and alerting systems to track containers' performance and resource utilization. He and Guo also implement a demo solution using an ML auto-encoder algorithm. Lastly, Yolchuyev [28] proposes an eXtreme Gradient Boosting (XGBoost) based model for anomaly detection on the Kubernetes orchestration platform. Either way, to the best of our knowledge, no prior work has been conducted on exploiting the asynchronous design of the K8s API communication.

Monitoring and Security in Kubernetes: While the academic community has reserved little attention in the field of monitoring and security in K8s, several tools in the market have been developed to address these issues.

OPA (Open Policy Agent) [29] is a tool that provides policy-based control for K8s. It allows users to define policies in a high-level language and enforce them before the K8s API server processes requests. For example, the use of the `--force` flag can be actively blocked using OPA. However, OPA only performs its checks before the API server, which means that container image downloads cannot be controlled in this way.

Another popular tool for monitoring K8s environments is Falco [30] with its plugin `k8saudit`. In addition to its runtime monitoring capabilities, Falco can ingest K8s audit logs and generate alerts based on predefined rules. However, it works on a line-by-line basis and does not have built-in status tracking or counting features, which makes it unsuitable for tracking container image downloads. Using Falcosidekick [31] and a visualization tool like Grafana, this limitation can be overcome, but it requires additional setup and configuration efforts that may not be feasible in all environments. Still, Falco only performs monitoring and alerting, and does not provide any mitigation capabilities.

Finally, other tools such as Tetragon [32] and Tracee [33], provide runtime monitoring of containerized applications. Both tools are based on eBPF and can monitor system calls, network activity, and other events in real-time. They can also detect anomalous behavior and generate alerts based on predefined rules. Like Falco, both Tetragon and Tracee focus on monitoring. However, to make them usable for identifying this type of weakness, several new rules would need to be created. Even with well-configured settings, they would still generate a large number of false positives, since they cannot correlate information from distinct sources. Finally, beyond their monitoring capabilities, they provide no means of actively mitigating the issues they may detect.

[16]CVE-2022-1708: https://nvd.nist.gov/vuln/detail/cve-2022-1708

[17]CVE-2025-0426: https://nvd.nist.gov/vuln/detail/cve-2025-0426

VIII. Conclusion

This paper demonstrates that the asynchronous implementation of some routes from the CRI-API can be exploited to create a resource-based DoS. We illustrate this threat through three distinct attacks. First, we focus on resource usage, where the attack used up to 95% of the CPU. Second, we show how this attack could interfere with the performance of other applications that run on the affected worker nodes of the cluster. Finally, we investigate how the cache could be manipulated by the constant stream of requests performed by the attack. We further experimentally prove the effectiveness of each attack in a local testbed and on GKE. Finally, we provide a guideline to solve the problem in all the related projects, and are currently collaborating with the working groups to solve this issue as soon as possible. As a stopgap, we present an eBPF-based proof-of-concept to monitor the activity in the cluster, stopping useless image downloads.

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016, doi: 10.1145/2890784.

[2] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source Kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 99:1–99:36, May 2023, doi: 10.1145/3579639.

[3] Y. He et al., "Cross container attacks: The bewildered eBPF on clouds," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 5971–5988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/he

[4] J. Xiao et al., "Attacks are forwarded: Breaking the isolation of MicroVM-based containers through operation forwarding," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 7517–7534. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/xiao-jietao

[5] J. G. Almaraz-Rivera, "An anomaly-based detection system for monitoring kubernetes infrastructures," *IEEE Latin America Trans.*, vol. 21, no. 3, pp. 457–465, Mar. 2023, [Online]. Available: https://latamt.ieeer9.org/index.php/transactions/article/view/7408

[6] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–37, Jul. 2023, doi: 10.1145/3539606.
[7] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Secur. Privacy*, vol. 19, no. 5, pp. 46–56, Sep./Oct. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9497237
[8] Kubernetes, "Kubernetes - Production-grade container orchestration," 2023. [Online]. Available: https://kubernetes.io/
[9] D. Balla, M. Maliosz, and C. Simon, "Open source FaaS performance aspects," in *Proc. 43rd Int. Conf. Telecommun. Signal Process.*, 2020, pp. 358–364. [Online]. Available: https://ieeexplore.ieee.org/document/9163456
[10] gRPC Authors, "gRPC," 2023. [Online]. Available: https://grpc.io/
[11] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes," *Cluster Comput.*, vol. 25, no. 2, pp. 1497–1513, Apr. 2022, doi: 10.1007/s10586-021-03517-8.
[12] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, "Impact of etcd deployment on Kubernetes, Istio, and application performance," *Softw.: Pract. Experience*, vol. 50, no. 10, pp. 1986–2007, Oct. 2020, doi: 10.1002/spe.2885.
[13] Z. Li et al., "Robbery on DevOps: Understanding and mitigating illicit cryptomining on continuous integration service platforms," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 2397–2412. [Online]. Available: https://ieeexplore.ieee.org/document/9833803
[14] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini, "Computing without borders: The way towards liquid computing," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2820–2838, Third Quarter, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9984946/
[15] Sysdig, "CloudNative security and usage report 2023," Sysdig, San Francisco, CA, USA, 2023. [Online]. Available: https://sysdig.com/2023-cloud-native-security-and-usage-report/
[16] Liqo Contributors, "Liqo," 2019. [Online]. Available: https://liqo.io/
[17] L. A. D. Knob, M. Franzil, and D. Siracusa, "MAGI system," 2023, Accessed: Sep 20, 2023. [Online]. Available: https://github.com/daisyfbk/magi
[18] L. A. D. Knob, M. Franzil, and D. Siracusa, "On exploiting gzip's content-dependent compression," 2023. [Online]. Available: https://github.com/daisyfbk/gzip-compression-tests
[19] Phoronix Media, "Phoronix test suite," Sep. 2023, Accessed: Jan 12, 2014. [Online]. Available: https://github.com/phoronix-test-suite/phoronix-test-suite
[20] L. A. D. Knob, M. Franzil, and D. Siracusa, "containerdsnoop," 2023. [Online]. Available: https://github.com/daisyfbk/containerdsnoop
[21] L. A. D. Knob, M. Franzil, and D. Siracusa, "Imagesnoop," 2023, Accessed: Jan 19, 2024. [Online]. Available: https://github.com/daisyfbk/imagesnoop
[22] Google Cloud Platform, "Microservices demo," Sep. 2023, Accessed: Aug 3, 2018. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo
[23] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proc. 2012 ACM Conf. Comput. Commun. Secur.*, Raleigh, NC, USA: ACM, 2012, pp. 281–292, doi: 10.1145/2382196.2382228.
[24] C. Fang et al., "HeteroScore: Evaluating and mitigating cloud security threats brought by heterogeneity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA: Internet Society, 2023. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f996_paper.pdf
[25] C. Fang et al., "Repttack: Exploiting cloud schedulers to guide co-location attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA: Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2022-149-paper.pdf
[26] M. Zhan, Y. Li, H. Yang, G. Yu, B. Li, and W. Wang, "CODA: Runtime detection of application-layer CPU-Exhaustion DoS attacks in containers," *IEEE Trans. Services Comput.*, vol. 16, no. 3, pp. 1686–1697, May/Jun. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9842371
[27] A. Ahmed and G. Pierre, "Docker-pi: Docker container deployment in fog computing infrastructures," *Int. J. Cloud Comput.*, vol. 9, no. 1, pp. 6–27, Jan. 2020, doi: 10.1504/IJCC.2020.105885.
[28] A. Yolchuyev, "Extreme gradient boosting based anomaly detection for kubernetes orchestration," in *Proc. 27th Int. Conf. Inf. Technol.*, 2023, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/10078576/
[29] CNCF, "OPA gatekeeper," 2025. [Online]. Available: https://github.com/open-policy-agent/gatekeeper
[30] Falco Security, "Falco," 2025. [Online]. Available: https://falco.org/
[31] Falco Security, "Falcosidekick," 2025. [Online]. Available: https://github.com/falcosecurity/falcosidekick
[32] Cilium, "Tetragon," Sep. 2023, Accessed: Mar 23, 2022. [Online]. Available: https://github.com/cilium/tetragon
[33] Aqua Security, "Tracee," Apr. 2025. [Online]. Available: https://github.com/aquasecurity/tracee

Luis Augusto Dias Knob received the MSc degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), Brazil, the PhD degree from the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil, and he is a researcher with Fondazione Bruno Kessler, Italy. His research interests include network management, cloud computing, virtualization and containerization, and network security.

Matteo Franzil received the BSc and MSc degrees in computer science from the University of Trento, Italy. He is currently working toward the PhD degree with the University of Trento, Italy, with a grant funded by Fondazione Bruno Kessler. His research interests include network management, monitoring, and observability, virtualization and containerization, and network security.

Domenico Siracusa is associate professor with the University of Trento. Previously, he was the head of the RiSING research unit with Fondazione Bruno Kessler (FBK). His research interests include infrastructure security and robustness, service orchestration and management, cloud and fog computing, and SDN/NFV and virtualization. Domenico authored more than 100 publications appeared in international peer reviewed journals and in major conferences on computing and networking technologies.