Resiliency as a Service Infrastructure (RaaSI)

Michael Coffey, Michael Donahoo, Shaun Hutton

July 25, 2022

Abstract

Ever-increasing dependency on infrastructure drives the need for operational resiliency of both physical and digital infrastructure. A standard system approach to resiliency involves customized solutions that are both brittle and constrained by expense. Computation and Communication (CNC) commonly combine to provide generalized solutions. Devices with these capabilities are now widely and often densely dispersed for everyday operations (e.g., wired/wireless routers are everywhere). We propose a platform that marshals spare CNC resources to provide services in response to events threatening system resiliency. Specifically, our platform monitors existing primary services and invokes a secondary service that operates over a mesh of devices to augment the diminished primary service if such services suffer degradation. As the platform offers spare CNC resources to counter degradation, we call our solution RaaSI (Resilience as a Service Infrastructure).

1 Introduction

The world's dependence on physical and digital infrastructure continues to grow while concurrently, threats to these infrastructures from cyber-attacks, natural disasters, and political instability also increase. Such increases quickly escalate the costs of compromise, going beyond economic to endangering life, as in the case of unregulated water treatment [12] or unmonitored traffic lights. Simply securing infrastructure is an inadequate approach to solve the problem of compromise. Regardless of how secure we make an infrastructure, there will always be some risk of exploitation. Due to the increased integration of digital services in everyday infrastructure (including industrial control systems (ICS) [19]), systems must prepare for compromise from both digital and physical adverse events. With the rise in infrastructure dependency, the need for resilient solutions is more critical now than ever and will only continue to increase.

On May 7th, 2021, gas pumps were closed in many parts of the United States due to a ransomware attack against the Colonial Pipeline. In this attack, hackers successfully compromised the pipeline's billing system, leading to the complete shutdown of the Colonial Pipeline for six days [9]. This six-day halt meant that the channel could not pump its average of 2.5 million barrels of gasoline per day to the eastern United States [9], leaving a lasting effect on gas prices [36]. The ransomware attack on the Colonial Pipeline might have been unavoidable. However, the Colonial Pipeline could have avoided the sudden halt of 2.5 million barrels of gasoline per day through resiliency strategies. There are many other examples of undesirable consequences resulting from premature infrastructure failure in an enterprise (i.e., [11, 22]); these occurrences can be lessened by careful planning and through graceful degradation. Administrators might use other techniques

to prevent these attacks from being successful, such as defense-in-depth. Still, these techniques do nothing after the compromise has already occurred in an enterprise.

Regardless of how much effort administrators put into keeping them alive, all services in an enterprise will eventually fail. To offset such risk, systems must increase their resiliency, i.e., the ability of systems to "anticipate, withstand, and recover from adverse events" [23]. What separates successful enterprises from more brittle ones is their ability to respond to these failures in services. When these failures occur, it is essential to have a backup plan that enterprises can use to augment the services experiencing failure. However, resilience solutions are often either too expensive or only applicable to a single service. For example, the UPS is just for the servers it is assigned to and is often unable to assist nearby systems even when it is unneeded by the servers themselves. In the case of a physical perimeter, one might add a second layer such as moat; however, this is often too expensive or impractical.

All enterprises consist of computation, communication, and sensor elements. The form of these elements in individual enterprises might differ; however, their fundamental responsibilities remain the same. Computation & communication (C&C) provide the capabilities to solve a range of problems. In fact, C&C solutions are solving problems previously addressed by physical solutions (e.g., Shop from home with Amazon). With the explosion of the networked devices (e.g., routers, WAPs, smart appliances), devices capable of computation and communication are becoming ubiquitous. Often, these devices contain various sensing capabilities such as communication (e.g., see bluetooth devices), video/audio, temperature, motion, etc. Systems are responsible for receiving data through sensors, mutating it into processable chunks through computation, and sending the processed data to its intended target through communication.

The ability of C&C to solve a range of problems and the near ubiquity of C&C devices gives us a mechanism and platform for incident response. When compromise occurs, response services may use spare communication/computation resource or even replace the normal services in order to provide infrastructure replacement. Instead of relying on a service specific resiliency solution, we can employ C&C and sensing resources to provide response services. Such response services replace normal operation services, providing graceful degradation. For example, switches on military bases could house a facial-recognition service that monitors the base for intruders when the perimeter fence is down. In the case of the Colonial Pipeline incident, individual gas pumps could store their transaction history locally when the overall transaction service becomes compromised.

We propose a system that leverages the ubiquity of C&C and sensing to provide incident response services. We can create a range of response services capable of utilizing the computation, communication, and sensing capability of this vast network of devices to approximate normal operation services. As these capabilities are general in nature, this creates a resilience service. We call this solution RaaSI (Resilience as a Service Infrastructure).

In order to address the problem of brittle critical systems, what is needed is a service that can provide enterprises with sufficient resiliency in order to continue delivering mission critical services in the face of adversity. Resiliency as a service can be achieved if a number of requirements are met. First, to accomplish resiliency as a service in an already established infrastructure, a solution needs to be able to provide detection and awareness of the status (failed or operational) of monitored infrastructure so that we can identify the set of potential responses. This action will give the solution knowledge of the entities used during graceful degradation. The solution must also be able to dynamically discover available resources (i.e., cameras, lights, speakers, memory, etc.) in the environment so we know what is usable. The solution should handle management of resources such as computation, communication, storage, and sensory resources, including the ability to co-

opt normal services and replace with another service so that we can respond to an incident. Such management includes intelligent utilization of scarce resources such as power (e.g., in the event of switching to battery backup). The solution should provide the development and deployment of response services over managed resources in response to failure. Also, the ability to switch between normal and replacement services in response to infrastructure failure/restoration must be achieved, along with the identification and ability to isolate failed or compromised systems and services.High-Availability (HA), "a failover feature to ensure availability during device or component interruptions" [5], is essential for nodes in the infrastructure to provide the continued and correct operation of services. Devices connected to this resilient solution must be able to communicate securely, and new devices should be allowed to integrate into the resilient solution easily. Secure communication is required to prevent malicious actors from intercepting sensitive information, that the resilient solution might transmit between nodes, in plain text. Security must be a focus for the solution by enforcing authentication, authorization, integrity of data, the principle of least privilege, and communication encryption. Nodes should be able to integrate into the resilient solution easily to allow for scalability. These integrated nodes must also have the ability to be maintained and updated with new versions of a resilient service. Finally, the quality of this solution must be considered and maintained through quality assurance techniques, such as static code analysis and unit testing. These requirements can be shown in the bulleted list below.

Resiliency Solution Requirements:

1. Detection and Awareness of the Status of Monitored Infrastructure
2. Development and Deployment of Response Services
3. Dynamic Discovery of Existing Resources
4. Efficient Management of Existing Resources
5. Statefulness of Response Services Preserved
6. Ability to Switch Between Normal and Replacement Services
7. Identification and Isolation of Failed or Compromised Systems and Services
8. Secure Distributed Communication
9. Scalability of Solution
10. High-Availability of all Parts in the Solution

The rest of this paper is organized as follows. Related works are described in Section 2. Section 3 provides the architecture and development of RaaSI. Experiments are described and discussed in sections 4 and 5. Section 6 details conclusions and future work on RaaSI. Finally, Section 7 provides a glossary for abbreviations used in this work.

2 Related Works

In the related works portion of this paper, we discuss existing solutions that developers can use to meet the requirements of a resiliency solution. First, we present defense through virtualization, which illustrates solutions that use the advantages of virtualization to provide resiliency. These

solutions utilize existing infrastructure in an enterprise and provide resilient communication between nodes. We then discuss techniques used to communicate throughout a distributed environment. This communication is essential in providing a resilient solution that optimally uses a distributed environment. Finally, we discuss scalable and stateful solutions. These solutions integrate new devices into existing systems while collectively maintaining the system's state.

2.1 Defense through Virtualization

One of the critical technologies needed for a resiliency solution is the virtualization of its secondary services [24]. These secondary services are responsible for supporting the essential operations of infrastructure after its primary services fail. Due to the uncertainty in the enterprise design that these solutions could be installed on, they need to be flexible in their deployment of these replacement services and efficient with their resource utilization.

Much research exists on the security benefits, and risks of incorporating virtualization into an infrastructure [33, 37, 30]. Regardless of the form of virtualization used, security risks exist in systems that use this technique. The main risk found through the use of virtualization is due to the added complexity of the system. Virtualization allows users to customize and scale diverse systems to their likeness. This freedom might add complexity to the system since users might populate a system with many different virtual environments running various software. This added complexity to the environment makes it difficult to apply generic security techniques throughout a system. Most security tools and techniques are designed to perform in specific systems. If an enterprise is comprised of many different systems, just as many different security tools might be required to harden the enterprise. Based on this environment complexity, security must be considered in each virtualized environment individually, along with the hosting system, without considering orchestration in virtualization [35]. Due to the separated nature of virtualized systems, security tools that work on the stand-alone system might not think of a virtualized environment running alongside that system [33]. The lack of duality in these tools means developers must consider security methods on the virtualized system and the machine hosting this system. A virtualized system can only be as secure as the machine that hosts it.

Although security issues arise through the complexity of virtualized systems, developers can also find security benefits through this tactic. Virtualization can provide availability and fault tolerance to a system [40, 33]. Virtualized environments can be easily cloned and duplicated throughout an infrastructure, providing this environment with scalability. Due to a virtual system's ability to store states of the system, if the system were to fail, developers could revert the system to a previous state before the failure occurred. Because of these security benefits, many security solutions exist through virtualization. Developers of resiliency solutions should consider virtualization due to its ability to scale solutions, the deployment capabilities that it provides for services, its efficiency in running unique services, its ability to provide secure communications, and the power of virtualization to provide high-availability services.

The article [20] describes a solution, VFence, that defends against DDoS attacks by deploying virtual machine agents to act as a filter layer for traffic coming into the server. These agents check all packets that are inbound and filter out the packets determined to contribute to the DDoS attack. Due to the virtualization of agents, VFence can dynamically change the amount of deployed agents to help with load balancing. This scalability is part of the critical functionality for VFence in that it allows for more filtering when an enormous influx of DDoS traffic exists. The scalability shown through VFence will enable nodes to be created and updated easily. This scalability matches the

resiliency solution requirement, "scalability of the solution."

Another security solution created with the use of virtualization is shown in SecPod [38]. SecPod is a framework that works to provide security to systems in the form of isolation. The author of SecPod describes the vulnerabilities of system kernels and provides SecPod as a virtualized solution to these vulnerabilities for users to work in an environment isolated from the kernel of the system. This isolation offers an environment where secure communication through virtual means can occur. Secure communication exists over virtual networks in [18] where software-defined networking provides resilient communication between virtualized nodes. Secure communication meets the resiliency solution requirement, "secure distributed communication."

Windows 10 even uses virtualization to add a layer of security to its OS. This security comes in the form of Windows Credential Guard [39]. Windows Credential Guard allows users to store their system credentials in an environment separated from the rest of their system through virtualization. This solution has a virtual machine running in the background whose sole responsibility is maintaining credentials. Due to this solution's separation from the rest of the system, the system's credentials are not necessarily compromised if the system gets compromised. This example shows how virtualization can provide an added layer of security to communication between nodes.

ESCAPE [6] is a moving target defense solution that makes use of containerized environments to provide confusion to attackers. The solution makes use of a monitoring software that monitors containers to see if attacks are ongoing on them. Once intrusion is detected, ESCAPE kicks in and changes characteristics about containers used in the system. Due to the dynamic nature of containers, these characteristic mutations are done with ease. This solution shows how simple it is to toggle between containerized systems, matching the resiliency solution requirement of the "ability to switch between normal and replacement services."

One of the downsides to virtualization is the need for extra resources to run these virtualized environments. Different types of virtualization are considered in [32] concerning resource utilization. This article shows containers to be less expensive on resources than virtual machines due to containers using their host OS kernel and not having to spawn a new one upon creation. Due to this minimization of resource utilization, developers use containers when efficiency is required.

In [2] the author describes containers as the ideal form of virtualization for High-Performance Computing (HPC) systems due to their efficiency in resource utilization, isolation capabilities, and ease of packaging for extensibility purposes. HPC systems present the need for compact solutions to maximize the solutions' computational ability. Containerization is key to the success of an HPC system due to its ability to maintain VM-like isolation while requiring relatively low resource utilization. This meets the resiliency solution requirement, "efficient management of existing resources." The use of containers also gives the benefit of ease in packaging applications. This ability to easily package applications makes systems that use containerized solutions more extensible and adaptable.

A solution that uses containers to provide resiliency to a system can be found in Flooid [4]. Flooid places a service in a containerized environment and maintains the healthy state of this environment. Once the service becomes corrupted, Flooid reverts the container to a healthy state. It does this through an orchestration software that the author designed. The container's healthy state is stored through files away from the container. This way, even when the container becomes corrupted, a version of its healthy state is preserved. This healthy state replica can revert a system to a previous state; however, if an enterprise has experienced a failure in the original state of a system, this solution might not be able to perform correctly due to missing resources from the fault. A resiliency solution should be able to maintain some form of functionality even if resources become depleted. This is where Flooid falls short; however, through Flooid, we can see an example of

virtualization used to provide a service with high-availability, which is a requirement for a resiliency solution. Flooid also demonstrates how simple it is to create a containerized image from a service. This ease of creation can be used to meet the resiliency solution requirement, "development and deployment of response service."

A resiliency solution should require many elements that virtualization provides, such as; scalability, secure communication, and efficient resource utilization. Due to this requirement match, developers should use virtualization when developing a resiliency solution. Also, developers should use containerized virtualization to ensure optimal efficiency, as shown in this research.

2.2 Distributed Communication

Developers should design a resilient solution for graceful degradation of an infrastructure to handle communication across nodes in the infrastructure so that the secondary services run on these nodes can communicate with each other. However, distributed communication over a network can produce security risks, add confusion to the system, and cause computational problems with concurrent reading and writing of data. Two forms of distributed communication are primarily used by nodes in a network: distributed file systems and distributed databases.

Distributed file systems allow users to connect endpoints and store data in a non-relational way [41]. These storage points resemble file systems on OS to give users an easier way of accessing the data. Two of the most popular distributed file systems are GlusterFs and CephFS. Due to their popularity, they have been well documented and will be discussed as the examples of distributed file systems. GlusterFS is designed with ease of access in mind [10]. Once installed, this file system looks identical to any Linux file system and allows users to integrate any Linux-based software with the stored data. GlusterFS also enables users to encrypt data flow between nodes with TLS encryption, giving systems that use it secure data communication. CephFS [3] is another distributor file system that acts similar to GlusterFS; however, CephFS uses binary object storage to store data. This binary object storage allows for some relationships in data to be created, allowing users to do operations on the data with ease. CephFS also enables users to encrypt data transmission through TLS. When comparing the two, GlusterFS performs better at scale than CephFS and should be used for a resiliency solution on a scalable enterprise. Users should use a distributed database when storing large amounts of data.

Distributed databases allow users to store bulk data in an interrelated way that uses resources found in multiple nodes across a network [29]. These databases help process data of immense size and mainly exist for machine learning and data science. While developers could use them to provide secure communication between nodes in a distributed environment, this would require a lot of setup time and not result in added benefit for the resiliency solution when compared to distributed file systems.

Developers of a resiliency solution should choose to use a distributed file system for node communication rather than a distributed database due to the setup cost that a distributed database adds with no added benefit. GlusterFS should be used as the distributed file system for this resiliency solution because of its ease of implementation and ability to scale. Solutions placed on an enterprise environment should quickly scale with the environment.

2.3 Scalability & Statefulness

Environments will not always be static; nodes might be added, taken away, or replaced. Due to this, a solution's ability to scale with its environment should be considered an essential piece of functionality. This scalability allows solutions to adapt to changes in the infrastructure they run on. The subsection "Defense through Virtualization" shows that solutions can meet this scalability by introducing containerized systems. However, a new problem arises from scalability in the form of statefulness. The state of containers does not persist in a scalable system by default; however, by providing an orchestration framework, statefulness can be kept. In researching the subject, we considered case studies of scalable systems to determine best how to provide a scalable solution while still keeping the state of the software for users to monitor. The key to the problem of scalability and statefulness is found in the form of virtualization orchestration solutions such as Kubernetes [17] and Docker Swarm [26]. Both scalability and statefulness are requirements for developing a resiliency solution.

The article [7] shows a scalable image repository. The authors of this paper discuss the current state of medical image processing through the use of cloud infrastructure. Although this allows for collaborative work on medical image processing, working with a cloud infrastructure alone can become frustrating due to the lack of scalability for extensive data. To solve this scalability problem, the authors introduce Kubernetes to manage the distributed computing associated with medical image processing. Due to this ability to maintain a scalable environment, solutions can use Kubernetes to run containerized systems that scale to their environment. Kubernetes addresses this scalability problem; however, Kubernetes does not maintain the state of these containers by default.

In [1] scalability for microservices is shown through the use of Kubernetes. This paper describes how microservices can maintain their distributed properties through Kubernetes. However, the article also considers a statefulness problem when implementing microservices across a Kubernetes cluster. The solution proposed in this paper is to use Persistence Volumes (PV) and Persistence Volume Claims (PVC) to mount volumes on the pods deployed in a Kubernetes cluster to maintain the state of these pods. PVC lets users directly connect a distributed file system to the Kubernetes instance. This connection provides nodes with a central communication point that the nodes can use for the stateful storage of data used in the nodes. However, the PVC can also act as a single point of failure, and stateful storage might best be mounted directly to the nodes that will be using the storage. Kubernetes also allows virtualized services to mount existing directories from the node they live on. By combining stateful storage mounting on a node with directory mounting in Kubernetes, developers can provide a solution to stateful storage in Kubernetes without causing a single point of failure.

A problem arises in using Kubernetes to maintain the networking configuration of the distributed nodes used in the cluster. This problem comes when new pods are pushed to these nodes, requiring the network routing to change. Overlay networks provide a Container Network Interface (CNI) to manage the network of these scalable pods, resolving this problem. The authors of [31] discuss two types of overlay networks used for Kubernetes, Calico and Cilium, and perform experiments on these two overlay networks to determine which one performs best when Kubernetes networks are scaled. In the experiments, the authors create a Kubernetes cluster with the Calico overlay network and a Kubernetes cluster with the Cilium overlay network. The authors then extended the number of pods the clusters were using and recorded the results through throughput to the pods. The results showed that Calico allows more throughput than Cilium when scaling is concerned due to the small amount of overhead it uses to route network traffic with IPIP [25] and BGP [8].

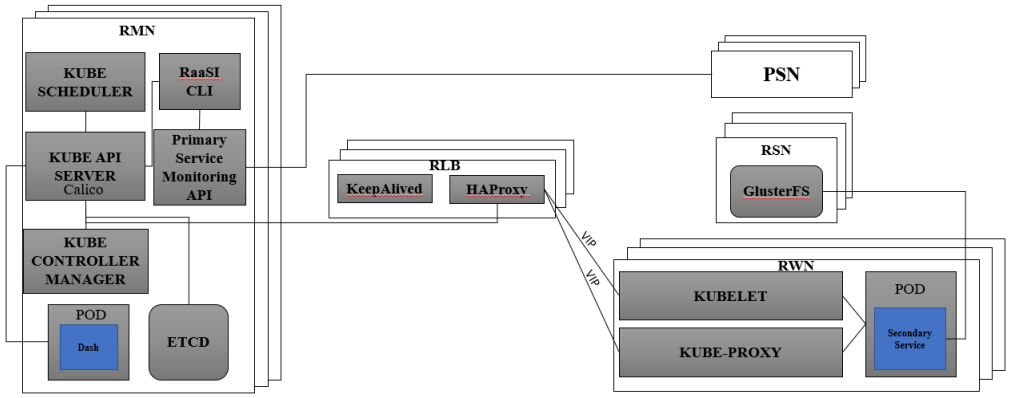
Developers should use Calico in environments where scaling is likely to occur; however, Cilium outperforms Calico in throughput in small-scale networks. Regardless of which overlay network is used, Kubernetes uses these overlay networks to provide secure communication throughout its cluster. For the development of a resiliency solution, this secure communication must be considered.

A resiliency solution should be able to scale to its environment while maintaining its services' state. Through research done in this section, we see that developers should use Kubernetes to provide orchestration to a scalable virtual environment. This orchestration allows tools to be added to the solution to ensure statefulness. One of these tools is the PVC; however, this might cause a single point of failure due to the PVC going directly through the control plane of the Kubernetes cluster. Instead, Kubernetes can directly mount the stateful storage onto the virtualized services themselves in Kubernetes. Finally, this resiliency solution should use the Calico overlay network to provide secure communication between the virtualized services in the Kubernetes cluster.

3 Architecture

In this section, we discuss the components of RaaSI, what they do, and how they meet the requirements needed for a resiliency solution as stated in the introduction. The architecture of RaaSI is split into five distinct parts; the RaaSI Master Nodes (RMN), RaaSI Worker Nodes (RWN), RaaSI Load Balancer (RLB), RaaSI Storage Nodes (RSN), and Primary Service Nodes (PSN). The integration of these parts make up the RaaSI cluster. This cluster enables RaaSI to monitor primary services, deploy and execute secondary services, maintain the state of secondary services, and provide high-availability throughout the RaaSI cluster. The interconnection of the RaaSI cluster is shown in 1.

Figure 1: RaaSI Architecture Diagram



3.1 Primary Service Nodes (PSN)

Primary services are the essential services of an infrastructure that RaaSI is providing resiliency for. These services are actively monitored by the RMN in the form of a heartbeat to check their health. Due to RaaSI being a generalized solution, these health checks are customizable by means

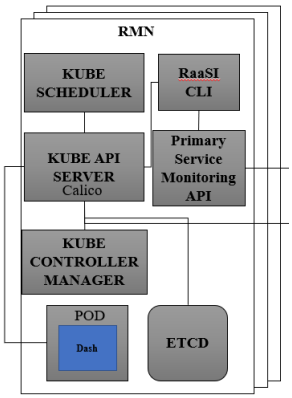
of the Primary Service Monitoring API. The Primary Service Monitoring API is created from a python solution designed for recording the up-time of services [14]. The health check of the primary services is split evenly between all RWN through a bash script. For more information on how this works, visit the Users Guide. This Primary Service Monitoring API meets the resiliency solution requirement of "detection and awareness of the status of monitored infrastructure."

When the heartbeat check on the primary service returns a failure, the RMN will execute the secondary services, associated to that primary service, on the RWN. To determine what secondary services should be executed, the user must tell the monitoring script what device should be found on the secondary service. Once the secondary service is executed, the health monitoring script begins to check if the primary service becomes healthy again. The purpose of RaaSI is not to provide a permanent replacement for the primary service, but rather to provide a secondary service as a temporary fix until the primary service becomes healthy once more. Once the primary service is determined to be healthy again, the secondary services is given a halt command that stops the secondary services associated with that primary service. This meets the resiliency solution requirement, the "ability to switch between normal and replacement services."

3.2 RaaSI Master Nodes (RMN)

In the RaaSI cluster, the Master Nodes are labeled as RMN and the Worker Nodes are labeled as RWN. The RMN is responsible for controlling the RaaSI cluster. Due to the responsibility of the RMN, services that are essential to the monitoring and management of the RaaSI cluster are set up on these nodes. These services are the Kube API Server, Kube Controller Manager, Kube Scheduler, ETCD, Primary Service Monitoring API, and RaaSI CLI. A diagram of the RMN is shown in 2

Figure 2: RaaSI Master Node (RMN)



The Kube API Server acts as an ambassador for the entire RaaSI cluster. All traffic processed by the cluster flows through this service. Due to this, this service is of great importance and is considered essential to the health of RaaSI. In order to alleviate interruptions in the Kube API Server, high-availability (HA) for the RMN is introduced, which meets the resiliency solution requirement of "high-availability of all parts in the solution." This HA uses the RLB to direct traffic from the RWN to healthy RMN. In order for this to properly work, three nodes are created for

the RMN, assigned a priority, and Round Robin Scheduling is done by the RLB to elect a primary RMN to send traffic to. The RLB is described in more detail in the RLB subsection.

The Kube API Server is also responsible for managing the Calico Overlay Network run on the RaaSI cluster. In order to provide a scalable and secure network for the secondary services on the cluster to communicate on, the Kube API Server manages a Calico Overlay Network. This network uses BGP and IPIP to route traffic to the secondary services on the network. It also sets up iptables rules on each secondary service to filter communication between secondary services on the network. A further discussion on choosing Calico over other Overlay Networks and the need for Overlay Networks is shown in section 2 under the "Scalability and Statefulness" subsection. This meets the resiliency solution requirements, "secure distributed communication" and "scalability of the solution."

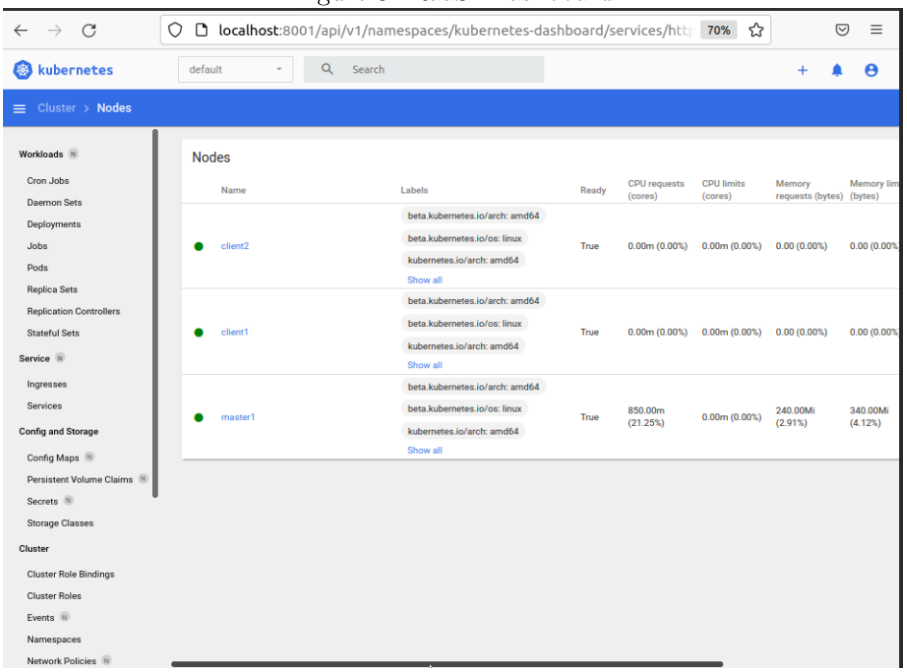
Another key service run on the RMN is the Kube Controller Manager. The Kube Controller Manager acts as a controller for the Kubernetes cluster. It manages deployments and balances the state of the system by viewing the state of the cluster from the Kube API Server and sending commands back to the Kube API Server to be executed throughout the cluster.

To deploy secondary services in the RaaSI cluster, the RMN uses the Kube Scheduler. This service monitors what nodes in the cluster match the deployment requirements and marks them to receive the secondary service. The deployment requirements are set when creating a secondary service and are in the in the form of RAM, CPU, and physical resources. Physical resources are devices that are connected to the node (i.e., cameras, speakers, lights, etc.). Secondary service creation occurs by the user setting these required resources and providing a containerized image of their secondary service, matching the resiliency solution requirement, "development and deployment of response service." Once these secondary services are created and deployed, the Kube Scheduler dynamically finds the nodes that match the resource requirements specified in the secondary service deployment, meeting the resiliency solution requirement, "dynamic discovery of existing resources." The Kube Scheduler will also find the most efficient node to place the secondary service. This meets the resiliency solution requirement, "efficient management of existing resources."

In order for users to monitor the health of nodes in their cluster and view the status of secondary service deployments, the RMN provides them with the RaaSI dashboard. This dashboard is hosted locally on the RMN.Through the dashboard, the user can identify nodes that have failed. Once these failed nodes are identified, the RMN can remove them from the cluster through kubeadm commands, which matches the resiliency solution requirement, "identification and isolation of failed or compromised systems and services." A open source repository was used to setup the dashboard [21]. The dashboard runs as a pod on the RMN and is installed during the execution of the master node installation script. An example of the dashboard is shown in 3. In order to view the GUI shown in 3, the user must first login. This authentication is used to ensure that not everyone on the network can monitor specifics about the cluster. To login, the user must provide the authentication service with a token generated by the RMN. This token is very complex which should deter any actor from attempting to brute force it; however, even if a brute force attempt is successful, the user can generate a new token which makes the old token useless. An example of this token authentication is shown in 4.

The ETCD service acts as a state store for the Kubernetes cluster. All configuration done to the Kubernetes cluster is stored on the ETCD. The ETCD is then accessed through the Kube API Server for the other services to view specifications relating to the Kubernetes cluster. These specification can be in relation to the state of nodes on a cluster, the location of secondary services deployed on the cluster, configuration of the Overlay Network, etc. Users can set a backup job for

Figure 3: RaaSI Dashboard



the ETCD through the RaaSI CLI.

The final piece of the RMN is the RaaSI CLI. This CLI acts as an API for all of RaaSI. In doing so, it allows users to install RaaSI components, set up primary service monitoring, and configure the RaaSI cluster. The entire CLI is written in bash and discussed more in the "RaaSI: Developers Guide" [27]. Due to the CLI being run on the RMN, the RMN acts as a centralized controller for the entirety of RaaSI. This way users do not need to move to other devices for RaaSI to configure them, installation and configuration are executed from the RMN.

3.3 RaaSI Worker Nodes (RWN)

The RWN are nodes that existed on the infrastructure prior to RaaSI that RaaSI is using to run secondary services. These nodes serve some other C&C purpose for the original infrastructure. These nodes are added to the RaaSI cluster through the CLI run on the RMN. There are three essential services related to the Kubernetes worker node that are being run on each of the RWN; Kubelet, Kube-Proxy, and the secondary service pod. A diagram of the RWN is shown in 5.

The Kube-Proxy service is where the RWN networking occurs. To do so, the Kube-Proxy calls on the Kube API Server run on the RMN to get its network configuration from the Calico Overlay Network. With this networking configured, the Kube-Proxy fetters communication between itself and all other nodes in the Kubernetes cluster.

On the RWN, Kubelet is used as an ambassador to the execution of secondary services on the node. Kubelet gets the information of what secondary services to place on the node and how to configure those secondary services from the Kube API Server on the RMN. This helps to complete

Figure 4: RaaSI Dashboard Login Page

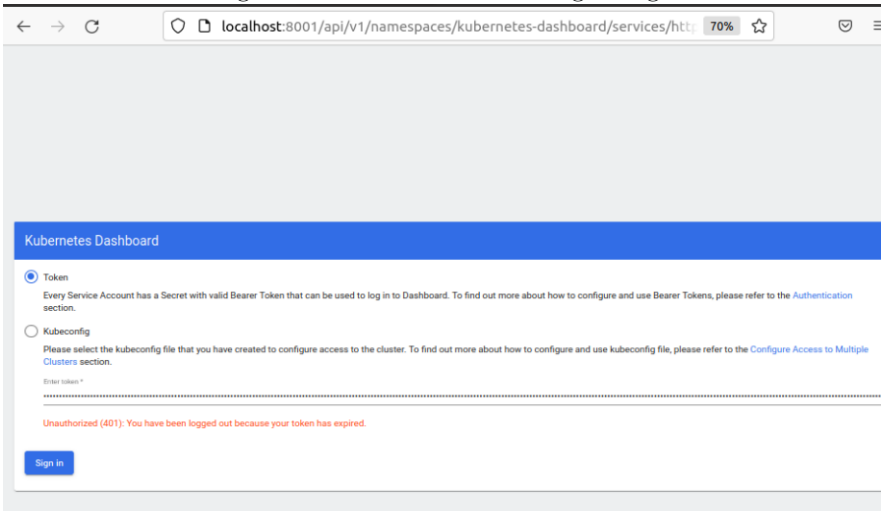
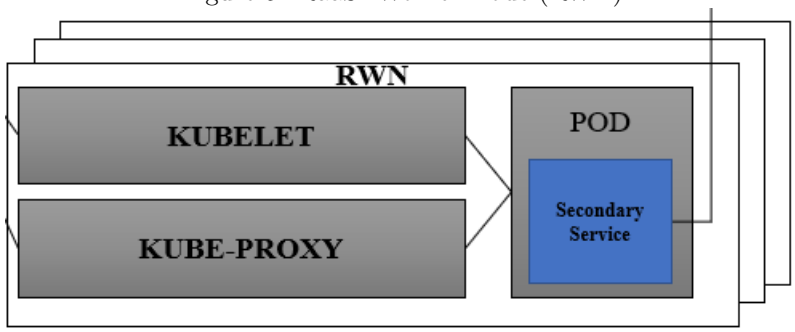


Figure 5: RaaSI Worker Node (RWN)



the resiliency solution requirement, "development and deployment of response services." It then monitors the secondary service and communicates with the Kube API Server its status. The Kubelet service acts as a control point governed by the Kube API Server.

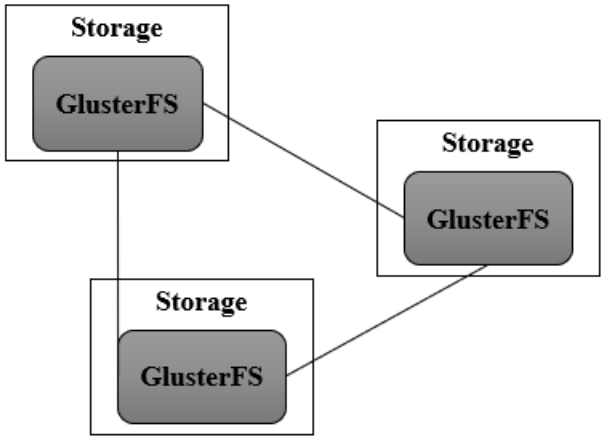
The final part of the RWN is the secondary service itself. This secondary service is given to the RWN by the Kube API Server on the RMN based on the resources available on the RWN. The secondary service run on the RWN is deployed during configuration of RaaSI through the CLI. This secondary service lays dormant on the RWN until the RMN tells it to activate (either due to a primary service failing its health check or through manual means). Once executed, the secondary service stores its state in a mounted file, maintained by a distributed file system on the RSN.

3.4 RaaSI Storage Nodes (RSN)

When running secondary services on RaaSI, statefulness of these secondary services is crucial. If the state of theses services is not preserved, the information that they gathered while they were

running is useless. For statefulness of secondary services to be achieved RaaSI uses a GlusterFS server. This server is setup on the RSN. A diagram of the RSN is shown in 6

Figure 6: RaaSI Storage Node (RSN)



GlusterFS was chosen as the data-store for preserving the state of the secondary services due to the peer-to-peer architecture of it's servers, ability to allow only specified clients to connect to it, integration with TLS to ensure encrypted data movement, and ease in integration. The state of secondary services is stored on the RSN, allowing secondary services to persist information recorded during their execution and giving them the ability to communicate with other secondary services. This matches the resiliency solution requirement, "statefulness of response services preserved." The peer-to-peer architecture of the GlusterFS servers gives the RSN built-in high-availability by storing a mirror of the data on each node in the peer-to-peer network. RaaSI is set up with three nodes in the RSN. So if one node fails, two others can support the stateful data, which meets resiliency solution requirement, "high availability of all parts in the solution."

The RSN uses TLS in the GlusterFS instance to encrypt data during transit. This encryption also allows the RSN to manage who is connect to it as a client. Clients of the RSN are limited to nodes that have a certificate that matches the certificate authority (CA) on the RSN created in setting up TLS encryption. This meets the resiliency solution requirement, "secure distributed communication." RaaSI enables this encryption between the RSN and secondary services through TLS 1.3. This form of data encryption is used because the same form of data encryption is used throughout the Kubernetes cluster. TLS 1.3 is used instead of 1.2 due to a vulnerability found with TLS 1.2 in CVE-2011-3389 [28].

3.5 RaaSI Load Balancers (RLB)

The RLB provides high-availability (HA) to the RMN by managing its controller nodes and providing a virtual IP that the RWN can use as a proxy to communicate with the RMN. To ensure the

RLB does not act as a single point of failure, it must also be set up with HA. For high-availability to be maintained in the RLB and the RMN keepalived and haproxy are run on the RLB. This matches the resiliency solution requirement, "high availability of all parts in the solution." Three instances of Load Balancers are created to provide HA on the RLB. A diagram of the RLB is shown in **??**.

Keepalived is used by the RLB to provide the RLB cluster with a virtual IP (VIP). This VIP is the advertised IP for the RLB and is used to route traffic between the RMN and the RWN. A node in the RLB is elected to house the VIP through a health check done on all nodes in the RLB. This health check works by checking if the Kube API Server can be reached through the VIP. If it can be reached, the node hosting the VIP is healthy; however, if the Kube API Server is non-responsive, the RLB cluster elects a new node to host the VIP by looking for the node with the highest priority.

Along with keepalived, haproxy is run on each node in the RLB. Haproxy is used to do the actual load balancing for the connected Kube API Servers found on the RMN. It does this by storing the IP addresses of the nodes on the RMN and running a Round Robin Scheduling algorithm to determine which traffic goes to specific nodes in the RMN.

After the RLB nodes have been created and configured, the RMN must connect to the RLB. This is done by the first node created in the RMN initiating with the RLB specified as its load balancer. The remaining two nodes in the RMN will then join the Kubernetes cluster through this first node [15]. Once this is done, HA has been setup on the on the RMN and RLB.

3.6 Architecture Decisions

We made many decisions regarding how RaaSIs' architecture would look throughout its development. These decisions made RaaSI more secure, failure resistant, and accessible to users. We considered the stateful storage of secondary service data through Kubernetes PVC before moving to a node-by-node approach. We secured GlusterFS by making a whitelist of acceptable IP address connections until a more secure process in certificates was found. We first designed the Primary Service Monitoring API to be a bash script that connected to the PSN and checked its health before RaaSI adapted a python based API. Finally, we developed the RaaSI CLI to give users a better experience than manually executing each bash script.

GlusterFS preserves the stateful storage of secondary service data. We used this type of storage due to its ease of integration into most applications and ability to perform well in scaled environments. This type of distributed file system has worked great with RaaSI in all stages of development; however, its connection to the secondary services has changed. Initially, RaaSI used the Kubernetes features, Persistence Volumes (PV) and Persistence Volume Claims. These features allowed for GlusterFS incorporation into the Kubernetes cluster through the RMN. The RMN would record what nodes housed the GlusterFS server and connect all secondary services to these nodes as GlusterFS clients. This method helped the scalability of secondary services in RaaSI by not needing any manual configuration of GlusterFS on the nodes that host these secondary services. However, PVC caused RaaSI to have a bottleneck point in the RMN. Due to the way RaaSI ran this stateful storage through the RMN, every time a read or write occurred on a secondary service, it needed to travel through the RMN first before it connected to the RSN. This bottleneck would prove to be a problem when dealing with large networks. Since all traffic flowed through the RMN, this also provided RaaSI with a single point of failure. If the RMN went down with PVC implemented, the secondary services would no longer be able to read or write data to the RSN. Due to these problems, we decided to make each RWN a GlusterFS client and mount the shared directory on

the secondary services. This way, we do not need to route all traffic through the RMN to the RSN.

Since we decided to make each RWN a GlusterFS client, we needed to have a way of limiting the creation of GlusterFS clients to only include nodes registered to be RWN. GlusterFS can whitelist IP addresses to join its cluster as clients; this is what we chose to do at first. After reassessing this, we discovered that a malicious user could exploit the whitelist by spoofing their IP address to match one of the RMN. To resolve this issue, we used TLS certificates to verify RMN to add as GlusterFS clients. With these certificates, malicious users could not join the GlusterFS cluster unless they stole the certificate from an RMN. An added benefit of this certificate verification is that it allowed us to use TLS encryption across the GlusterFS cluster.

Initially, we also used certificates to monitor the health of the primary service. Before RaaSI adapted the Primary Service Monitoring API, we designed a bash script to periodically sign in to the PSN with SSH through SSL certificate authentication and query the node for the status of the primary service. RaaSI could do this query by checking systemctl or through the user writing a more detailed health checking script. We eventually decided that using SSL certificate authentication over SSH was not secure enough to monitor a primary service's health due to a malicious user's ability to steal this certificate. Instead of this health monitoring script, we adopted the Primary Service Monitoring API. This API allows users to write Python code to check the health of a primary service without having to access the PSN on which this service is located. The Primary Service Monitoring API also allows users to associate a set of secondary service deployments to replace each primary service.

Finally, we created the RaaSI CLI to give users a more user-friendly experience when installing, configuring, and running RaaSI. Before we created this CLI, users were required to run each installation and configuration script on the node in question. This requirement meant that RaaSI could not scale well and caused frustration for users. The RaaSI CLI stitched these scripts together and allowed users to direct everything from one centralized RMN.

The Developers Guide shows the management and configuration of RaaSIs' through code analysis and testing [27]. It also gives users an in-depth look at how they can configure RaaSI to match their use cases more closely. Users should consult the Developers Guide if they want to manipulate the use cases of RaaSI or if they want to consider how to run quality assurance tools on RaaSI.

4 Analysis

After we completed the development of RaaSI, we conducted experiments to demonstrate its performance. These experiments utilized different secondary services and RaaSI node count to provide various results that generalize performance metrics. After describing these experiments and their results, we will provide an overall analysis of RaaSIs' performance based on the experiments. Once the performance of RaaSI is discussed, we show the vulnerabilities that malicious actors could exploit in RaaSI by walking through the threat model. Finally, the use cases of RaaSI are shown by walking through two real-world scenarios.

4.1 Experimentation

We ran experiments on RaaSI to demonstrate how well it performs when presented with various secondary services. For these experiments, we created a secondary service, deployed the secondary service for the first time, halted the secondary service, and deployed it after its initial deployment. During the initial deployment, the assigned RWN will pull that image from Docker Hub and store it

locally. When a deployment of a secondary service happens after its initial deployment, the image is already stored locally on the RWN, causing its time to complete to be significantly less than the initial deployment. We recorded the time it took to perform each task in Table 1.

For experiments 1, 2, and 3, we deployed a facial-recognition secondary service to RaaSI. For this facial-recognition service we used dlib face_detector [16]. Due to the nature of facial-recognition solutions, we required many resources to run this service. The facial-recognition secondary service required 5 GB of RAM and 2 CPUs to run on each RWN. The initial deployment time for this secondary service was much larger than that of any other experiment due to the size of the image. The initial deployment time did not increase much as we added RWN nodes to receive this secondary service. The average initial deployment time per node for the facial-recognition service was 31.5 minutes. Once RaaSI completed the initial deployment, subsequent deployments were much quicker. The average deployment time per node for the facial-recognition service was 14.7 seconds. Finally, the halt time of the facial-recognition service was similar to the deployment time, with an average of 16.41 seconds per node.

We deployed a Snort secondary service for experiments 4, 5, and 6. The Snort secondary service was created and deployed by the developers of RaaSI. This service takes an Ubuntu 14.04 docker image, installs Snort 2.9.19, and runs the Snort instance. The Snort service required less than 1 GB of RAM and 1 CPU to run. The initial deployment of this secondary service is much quicker than that of the facial-recognition service, with an average initial deployment time of 10.75 minutes per node. The subsequent deployment time average is 12.92 seconds per node and the halt time average is 14.98 seconds per node.

For the remainder of the experiments, we deployed a Rsyslog secondary service. The Rsyslog secondary service was also created and deployed by the developers of RaaSI. It works similarly to the Snort service by taking an Ubuntu 14.04 docker image, installing Rsyslog, and running the Rsyslog instance. The Rsyslog service also requires less than 1 GB of RAM and 1 CPU. The initial deployment average for this service was 7.34 minutes per node. After the initial deployment, subsequent deployments took an average of 12.59 seconds per node, and halting took an average of 13.84 seconds per node.

From these experiments, we can see that the size of the secondary service image directly affects the image's initial deployment time. The Rsyslog service had the quickest initial deployment time, at an average of 7.34 minutes per node, while only requiring 1 GB of RAM and 1 CPU. By comparison, the facial-recognition service had the most considerable initial deployment time at an average of 31.5 minutes per node while requiring 5 GB of RAM and 2 CPUs. These experiments also show that a secondary service's deployment and halting time after its initial deployment is much smaller than the initial deployment time. These times are much closer to each other than the initial deployment times. From this data, we can conclude that pulling the images from Docker Hub provides a bottleneck in the initial deployment process. This bottleneck is directly correlated with the size of the image pulled. Finally, the secondary services can deploy and halt quickly at about 15 seconds per node after this initial pull.

Table 1: RaaSI Experiments

Experiment ID	**Secondary Service Name**	**RWN Nodes Effected**	**Initial Deployment Time**	**Deployment Time**	**Halt Time**
1	Facial-Recognition	1	51 min	16 sec	20 sec
2	Facial-Recognition	2	60 min	21 sec	32 sec
3	Facial-Recognition	4	54 min	1 min 10 sec	53 sec
4	Snort	1	16 min	14 sec	17 sec
5	Snort	2	21 min	24 sec	26 sec
6	Snort	4	23 min	51 sec	49 sec
7	Rsyslog	1	9 min	17 sec	19 sec
8	Rsyslog	2	16 min	20 sec	22 sec
9	Rsyslog	4	20 min	43 sec	46 sec

4.2 Performance

This section discusses RaaSIs' performance throughout installation, secondary service creation, PSN monitoring, and device failure. We show the performance of RaaSI installation for each component of RaaSI. Secondary service creation, execution, and halting times are given. This section then goes over the time it takes for the Primary Service Monitoring API to trigger a secondary service execution. Finally, we present what happens when RaaSI experiences device failure.

The RaaSI CLI initiates the installation of RaaSI components. After their initialization, each RLB takes an average of five minutes to install, each RMN takes an average of seven minutes to install, each RSN takes an average of twelve minutes to install, and each RMN takes an average of seven minutes to install. The RSN takes the longest to install due to the time it takes to install GlusterFS, join the GlusterFS servers together in a cluster, and configure TLS encryption across the server. Most of the installation time for each component is caused by the installation of technologies required for RaaSI. Configuration of these technologies does not take very long.

Secondary services can be created and executed after RaaSIs' installation. Once these secondary services are created, it can take up to 60 minutes to push the created service to each RWN on its first push. This time delay is caused by the RWN pulling the containerized image from Docker Hub. This pull time is directly correlated to the size of the containerized image. If developers use smaller images, this time will be significantly reduced. Once RaaSI pulls these images, it takes an average of 15 seconds per node to execute the secondary service on the RWN. Once the system gives the halt command to these secondary services, it also takes an average of 15 seconds per node to halt the secondary service on the RWN.

The Primary Service Monitoring API monitors the health of each primary service configured in RaaSI. These health checks are split between all RMN to provide parallel monitoring. On each instance of the Primary Service Monitoring API, monitoring of each primary service occurs synchronously. This means that the length between monitoring primary services depends on the number of primary services that need to be monitored and the efficiency of the monitoring script used. Due to the amount of configuration users can do to the Primary Service Monitoring API, it is difficult to measure this API's performance accurately.

High-Availability (HA) is one of the requirements of RaaSI. This HA means that RaaSI should be resilient to device failure, which it is. It would take all nodes of a given component to experience failure before RaaSI started to degrade. If all RWN failed, RaaSI would have no nodes to run the secondary services on; however, it could still monitor the health of the primary services. If all RMN failed, RaaSI would be unable to execute any more secondary services, and it would not be able to monitor the primary services anymore. However, RaaSI would still be able to run the secondary services that already existed on the RWN and store the data they collect on the RSN. If all RSN failed, RaaSI would not be able to store or retrieve any stateful data on the GlusterFS cluster. However, the secondary services would still be able to execute, and the RMN would still monitor the primary services. Finally, if all RLB failed, the RWN would no longer be able to communicate with the RMN over the overlay network. This failure would prevent the RMN from executing any new secondary services. However, the RMN would still monitor primary services, and secondary services that RaaSI already deployed would be able to run on the RWN and store their state on the RSN. Due to the way RaaSI is partitioned, some form of graceful degradation will be maintained even if entire component clusters fail.

4.3 Threat Model

RaaSI is a software solution that works alongside critical infrastructure to provide resiliency for essential primary services. This responsibility paints a target on RaaSI for malicious actors to attack and devastate the resiliency of an enterprise. Due to the inevitability of an attack on RaaSI, it is prudent to consider the attack surface and to address how RaaSI reacts to these attacks. To do so, we must consider the attackers' goals, how the attacker will exploit RaaSI to achieve these goals, the risk level of each of these exploits, and how RaaSI remediates these exploits. The threat model showing these categories is shown below in Table 2. This threat model first discusses the goal that the attacker is planning to achieve. It then shows the target for this specified goal, the exploit against this target, and how to exploit the target. After this, we offer the risk of this exploit. For this paper, risk "is a function of the likelihood of a given threat source exploiting a potential vulnerability and the resulting impact of a successful exploitation of the vulnerability" (or put, risk = likelihood x impact) as defined by NIST [34]. Finally, we give a brief description of the remedy for these exploits. Due to the variety of systems that can run RaaSI, some of these remedies might seem vague. For example, in a lot of the exploit remedies system hardening is used. This system hardening is the responsibility of the user of RaaSI to do on their systems before RaaSI is installed. Users who are not sure on how to harden their system should consult the CIS Benchmark [13]. These benchmarks provide details on how many different OS should be secured to prevent intrusion.

4.4 Case Study

We explored two scenarios to demonstrate the use cases and analyze the performance of RaaSI. In the first scenario, RaaSI works with a military base as it continues to defend itself during a disaster. During the second scenario, RaaSI is protecting infrastructure against cyber threats by migrating defenses after a primary defense has failed. Proxmox VE was used to test both of these scenarios. Proxmox is a virtual machine repository that allows users to create and connect these virtual machines. For these scenarios, we used a Ubuntu 20.04 machine with 8GB of RAM and 4 CPUs for all nodes in the RaaSI cluster.

4.4.1 Military Base: Perimeter Breach

The first scenario shows an actual use case and how RaaSI provides graceful degradation to replace an essential primary service. For this scenario, there is only one secondary service associated with the primary service; however, this secondary service requires special physical hardware to run. This scenario also shows what happens when the primary service comes back online.

Military bases are designed to keep unauthorized personnel out. They accomplish this partly by erecting an electric fence around the base's border. This fence is a deterrent to any curious actor who wants to venture to the other side. However, what happens if a large storm comes through, knocking out the power and destroying the fence? These military bases have highly durable CNC solid switches used to provide communication throughout the base during times of disaster. These switches also have ports where administrators can plug surveillance cameras into them. Suppose RaaSI exists on the solid switches in the military base, and the solid switches have connections to surveillance cameras. In that case, these switches can become a virtual fence through a facial recognition secondary service when the perimeter fence gets knocked out. The primary service in this scenario is the electric fence guarding the perimeter of the military base. The secondary

service in this scenario is the facial recognition service that, once deployed, provides a virtual fence to replace the down physical one.

For simulating this scenario, the primary service can be any service due to the highly configurable Primary Service Monitoring API. The service chosen for the primary service was an Apache2 website. This website will act as a health monitoring access point for the status of the electric fence. For the secondary service, dlib face_detector [16] was used. We chose this facial recognition software due to its lightweight nature and docker-friendly design. After the secondary service was pushed to Docker Hub (see instruction in the Users Guide on creating a secondary service), we configured the facial recognition secondary service in the RaaSI instance. This secondary service required 5GB of RAM and 2 CPUs to run on each RWN and a virtual Ubuntu camera. Due to the significant resource requirement for this secondary service, it initially took 50 minutes to pull the image from Docker Hub and deploy it on all nodes with the virtual Ubuntu camera. However, once RaaSI completed this initial pull, it took less than one minute for the facial recognition service to run on the necessary nodes. For this scenario, there were seven RWN, and four had the required resources to run the facial recognition service.

The health check done on the primary service was a curl on the website hosted from the PSN IP address and searching the response for the word "Apache." If the curl contained "Apache," the primary service passed the health check; if it did not, the health check would fail. We shut down the Apache2 web server to test the automated deployment of the facial recognition service. After the primary service failed, it took the secondary services two minutes to execute on all four RWN that had the required resources. No RWN ran the secondary service while having insufficient resources. Finally, we turned the Apache2 web server back on, and all secondary services halted within one minute.

4.4.2 Defense Migration

For the second scenario, two secondary services with similar resource needs replace one primary service. This scenario tests RaaSI's ability to evenly distribute secondary services among RWN, that match the requirements needed during primary service failure. It also tests the ability of these secondary services to access the GlusterFS shared storage through their containerized service.

The primary service for the second scenario is an Untangle FW. This scenario migrates defense responsibilities from the Untangle FW to a snort service and a rsyslog service. When the Untangle FW fails, RaaSI should evenly distribute snort and rsyslog services among devices that match the requirements for the services. RaaSI should collect data on the distributed file system that RaaSI manages for secondary services. Once the Untangle FW becomes healthy again, the rsyslog and snort services should halt the execution, and the data in the distributed file system should persist.

Once again, an Apache2 web server was the primary service for this scenario. For the secondary services, docker containers were created that installed and ran Rsyslog for the Rsyslog service and Snort for the Snort service. Once we made these secondary services, we pushed them to Docker Hub to be deployed in the RaaSI cluster. The Rsyslog and Snort service needed less than 1 GB of RAM and 1 CPU to deploy on RaaSI. RaaSI also used a virtual Red Hat device to identify what RWN to place these secondary services on. The initial creation of these secondary services took five minutes.

RaaSI monitored the primary service the same way as it did for the first scenario. Once the primary service failed, the secondary services were split evenly among the four RWN that matched their requirements, with two Rsyslog services and two Snort services choosing unique RWN. It took

less than one minute for the secondary services to begin executing on the RWN. Finally, we turned the Apache2 web server back on, and it took less than one minute for the secondary services to halt.

5 Discussion

RaaSI was run through several scenarios to show how it holds up under its use cases. In this section, we discuss the results of the case study scenarios. From those results, we offer the threats to the validity of RaaSI and highlight the following steps on how developers might remediate these threats.

5.1 Scenario Results

After running RaaSI through the two scenarios discussed in Section 5, it is apparent that RaaSI is effortlessly configurable and can deploy secondary services with great haste when a primary service fails. Users can also use the distributed file system to communicate between secondary services in RaaSI. For more communication ability, users can directly message all nodes running a secondary service or a single node through the kubectl exec command (the Developers Guide shows more information regarding this). The scenarios showed that the speed it takes to execute a secondary service is only slightly impacted by the resource demand of the secondary service. The initial deployment, however, showed a significant difference in performance based on resource demand. The resource-heavy facial recognition secondary service took about five times longer to initially deploy on the RWN than the less heavy, rsyslog and snort secondary services. It is also important to note how simple it was to monitor the health of the primary service (the Developers Guide shows the information on how to configure the Primary Service Monitoring API [27]).

5.2 Threats to Validity

While running RaaSI through the case study scenarios, specific areas for improvement were highlighted. Users who want to run RaaSI with a distributed file system other than GlusterFS need to modify the RSN and RWN installation scripts to do so. This modification process can become difficult for the user and cause problems installing these nodes. It would be preferable for the user to select what distributed file system they wanted instead of manually installing it themselves. Similarly, the Primary Service Monitoring API configuration could also get confusing. Currently, the API only monitors one default service, HTTP. If users want to monitor other services, they need to write their python code (shown in the Developers Guide). This modification gives the user more opportunities to inject incorrect code into RaaSI and cause problems with its functionality. Finally, RaaSI uses a simple installation of OSSEC to check if its' files are modified. However, OSSEC monitors the files to see if they are being changed; it does not prevent users from modifying them in the first place. For the continuation of RaaSI, developers should investigate better essential file reconnaissance.

5.3 Next Steps

The next steps of RaaSI should be to give the user a more automatic experience in the configuration of RaaSI. RaaSI follows a specific installation and configuration procedure. Due to this, users attempting to fill different use cases will have to do some manual configuration, which can lead to

faults in functionality. For the future work of RaaSI, these manual configurations should be made automatic with a configuration wizard. RaaSI should also have a wizard to help users set up the Primary Service Monitoring API. Finally, RaaSI should watch over its' essential files with more security than what OSSEC alone can provide.

6 Conclusion

In conclusion, we are increasingly more reliant on IT systems for a large number of key national functions. Our systems are growing in complexity, and as something becomes more complex the greater the likelihood there is for flaws and vulnerabilities. Our systems are not engineered sufficiently to handle system failures whether they be the result of physical failures or cyber attacks. All enterprises should prepare for their services to eventually fail. A great way to ensure that your enterprise does not die with its services is to provide resiliency through graceful degradation. RaaSI uses all enterprises' computation, communication, and sensor elements to run secondary services that offer graceful degradation for primary services. RaaSI has been tested and run through case study scenarios to show how effective it is at providing resiliency for an infrastructure. Through these scenarios, RaaSI has been demonstrated as reliable, accurate, and fast. Finally, RaaSI should continue in development with more user-friendly configuration wizards and more focus on essential file security.

7 Glossary

Abbreviations:

The following abbreviations are used in this manuscript:

RaaSI	Resiliency as a Service Infrastructure
RMN	RaaSI Master Node(s)
RWN	RaaSI Worker Node(s)
RLB	RaaSI Load Balancer
RSN	RaaSI Storage Node(s)
PSN	Primary Service Node(s)
PV	Persistence Volume
PVC	Persistence Volume Claim
FS	File System
HA	High-Availability
DoS	Denial-of-Service
DDoS	Distributed-Denial-of-Service
MTD	Moving Target Defense
IoT	Internet of Things
HPC	High Performance Computing
ICS	Industrial Control System
CRF	Cyber-Resiliency Framework
NIST	National Institute of Standards and Technology
BGP	Border Gateway Protocol
CNI	Container Network Interface
CLI	Command Line Interface
API	Application Program Interface
TLS	Transport Layer Security
VIP	Virtual IP
CNC	Computation and Communication

Table 2: RaaSI Threat Model

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
1.1	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Becoming a RWN to access pods that use GlusterFS, gaining access to the GlusterFS mounted volume	**Likelihood:** Not likely for an outside node to become a RWN (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	Joining the RaaSI cluster requires a generated token from the RMN. Brute forcing can be used to attack this, but the token will expire before that time, due to its complexity. This token can only be accessed by breaking into the RMN.
1.2	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Becoming a RWN to access pods that use GlusterFS, gaining access to the GlusterFS mounted volume	**Likelihood:** Not likely for an outside node to become a RWN (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	Users should make sure the RMN system is secured against intrusion. This can be done through general hardening of the system which can vary based on the OS used. To harden, users should follow the CIS Benchmark for their OS [13]

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
2	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Become GlusterFS client	**Likelihood:** Not likely for an outside node to become a GlusterFS client (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	Certificate authentication through TLS 1.3 is setup between the GlusterFS servers and their clients. This prevents unwanted clients from joining the GlusterFS.
3	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Break into RWN machine	**Likelihood:** If the RWN system is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	The RWN nodes should be secured to prevent malicious actors from breaking in. This system security should be conducted by the user before RaaSI is installed. This can be done with general hardening of the system which may vary depending on the OS used. To harden, users should follow the CIS Benchmark for their OS [13].

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
4.1	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Break into GlusterFS server	**Likelihood:** If the RSN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	The nodes that house the GlusterFS server should be secured to prevent malicious actors from breaking in. To secure the GlusterFS server, users should follow the CIS Benchmark for their OS [13].
4.2	Read/Modify/Destroy data stored by secondary services	GlusterFS	Access stateful data from GlusterFS	Break into GlusterFS server	**Likelihood:** If the RSN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If the attacker can access GlusterFS data they could compromise sensitive information that secondary services are recording (High) **Overall:** Medium	Due to the ability of users to make any node a RSN, the responsibility of securing the RSN falls on the user of RaaSI. This can be done with general hardening of the system which can vary based on the OS used. To harden, users should follow the CIS Benchmark for their OS [13]

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
5	Force secondary service triggering	PSN	Interrupt health monitoring of PSN	DoS on RMN	**Likelihood:** DoS attacks are simple to enact and common by attackers (High) **Impact:** If one RMN goes down from a DoS attack, there are still at least two more active (Low) Overall: Medium	High-availability of RMN means that if one RMN goes down, there are at least two more that can replace it. However, if all RMN go down from a DoS attack, the only defense is to block the IP of the DoS traffic.
6	Force secondary service triggering	RMN	Access manual trigger in RMN	Break into RMN	**Likelihood:** If the RMN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If the secondary service can be manually triggered resources could be over utilized (Low) **Overall:** Low	RMN is kept away from RWN and should be secured by the user of RaaSI. This can be done with general hardening of the system which can vary based on the OS used. To harden, users should follow the CIS Benchmark for their OS [13]

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
9.1	Halt secondary service execution	RWN	Destroy RWN	Break into RWN machine	**Likelihood:** If the RWN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If secondary services are forced to halt, resiliency through graceful degradation cannot occur (High) **Overall:** Medium	The secondary service is deployed to all RWN that match the resource requirements. If some RWN are destroyed that are meant to house a secondary service, the secondary service will still run on the RWN that have not been destroyed.
9.2	Halt secondary service execution	RWN	Destroy RWN	Break into RWN machine	**Likelihood:** If the RWN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If secondary services are forced to halt, resiliency through graceful degradation cannot occur (High) **Overall:** Medium	There is no remediation for this exploit if all RWN are destroyed. If execution has already begun and is halted the statefulness for secondary service run on the compromised RWN is stored outside of RWN on GlusterFS server so that data persists.

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
10.1	Halt secondary service execution	RWN	Destroy RWN	Physical disruption	**Likelihood:** Attackers would need physical access to the RWN to physically destroy it which is not likely (Low) **Impact:** If secondary services are forced to halt, resiliency through graceful degradation cannot occur (High) **Overall:** Medium	The secondary service is deployed to all RWN that match the resource requirements. If some RWN are destroyed that are meant to house a secondary service, the secondary service will still run on the RWN that have not been destroyed.
10.2	Halt secondary service execution	RWN	Destroy RWN	Physical disruption	**Likelihood:** Attackers would need physical access to the RWN to physically destroy it which is not likely (Low) **Impact:** If secondary services are forced to halt, resiliency through graceful degradation cannot occur (High) **Overall:** Medium	There is no remediation for this exploit if all RWN are destroyed. If execution has already begun and is halted the statefulness for secondary service run on the compromised RWN is stored outside of RWN on GlusterFS server so that data persists.
11	Prevent secondary service from halting	PSN	Block communication between RMN and PSN	DoS on PSN	**Likelihood:** DoS attacks are simple to enact and common by attackers (High) **Impact:** If the secondary service does not halt when intended, resources could be over utilized (Low) **Overall:** Medium	If the PSN is being attacked by a DoS, the PSN should not be considered healthy and the secondary services supporting this PSN should not halt. This is the intended functionality for RaaSI.

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
12	Prevent secondary service from halting	PSN	Block communication between RMN and PSN	DoS on RMN	**Likelihood:** DoS attacks are simple to enact and common by attackers (High) **Impact:** If the secondary service does not halt when intended, resources could be over utilized (Low) **Overall:** Medium	User managing the RMN should setup firewall rules to block traffic coming from the malicious IP conducting the DoS.
13	Malicious modification of secondary service	RMN	Modify secondary service on RMN	Breaking into RMN	**Likelihood:** If the RMN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If attackers are able to modify secondary services, they could push out anything to the nodes across the enterprise (High) **Overall:** Medium	The RMN should be made secure by users of RaaSI. This can be done though general hardening of the system which can vary based on the OS used. To harden, users should follow the CIS Benchmark for their OS [13]

Continued on next page

Table 2: RaaSI Threat Model (Continued)

ID	Attacker Goal	Target	Exploit Against RaaSI	How to Exploit	Risk Level	Exploit Remedy
14	Malicious manipulation of RaaSI's code	RMN	Manipulating RaaSIs' code	Breaking into RMN	**Likelihood:** If the RMN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If RaaSIs' code is compromised, attackers could push malicious code that could threaten the enterprise (High) **Overall:** Medium	RaaSI has an OSSEC attachment that monitors all files related to RaaSI on the RMN. If any of these files are modified the user of the RMN system will be notified of the change and can revert it before running the code. However, there is still the risk that the code will run before the manipulation is found. This is why the system should be secured.
15	Breaking Kubernetes	RMN	Kubernetes services are halted	Breaking into RMN	**Likelihood:** If the RMN is hardened it is not likely for attackers to break in. To harden, users should follow the CIS Benchmark for their OS [13] (Low) **Impact:** If Kubernetes is broken, RaaSI will no longer work for users (High) **Overall:** Medium	The RMN should be made secure by users of RaaSI. This can be done though general hardening of the system which can vary based on the OS used. To harden, users should follow the CIS Benchmark for their OS [13]

References

[1] Leila Abdollahi Vayghan et al. "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019, pp. 176–185. DOI: 10.1109/QRS.2019.00034.

[2] Subil Abraham et al. "On the Use of Containers in High Performance Computing Environments". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 284–293. DOI: 10.1109/CLOUD49709.2020.00048.

[3] Luca Acquaviva et al. "Cloud Distributed File Systems: A Benchmark of HDFS, Ceph, GlusterFS, and XtremeFS". In: *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2018, pp. 1–6.

[4] Merino Aguilera and Xavier J. "Managed Containers for Increased Cyber-Resilience". In: 2017.

[5] Emmanuel Aroms. *NIST Special Publication 800-113 Guide to SSL VPNs*. 2012.

[6] Mohamed Azab et al. "Smart Moving Target Defense for Linux Container Resiliency". In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 2016, pp. 122–130. DOI: 10.1109/CIC.2016.028.

[7] Tibério Baptista, Luís Bastião Silva, and Carlos Costa. "Highly scalable medical imaging repository based on Kubernetes". In: *2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2021, pp. 3193–3200. DOI: 10.1109/BIBM52615.2021.9669559.

[8] *BGP (Border Gateway Protocol) - Linux*. https://netref.soe.ucsc.edu/node/11. Accessed: 2022-03-29.

[9] Christopher Bing and Stephanie Kelly. "Cyber attack shuts down US fuel pipeline 'jugular,'Biden briefed". In: *Reuters. Accessed* (2021), pp. 05–25.

[10] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. *Glusterfs one storage server to rule them all*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012.

[11] Elisabeth Cardis and Maureen Hatch. "The Chernobyl accident—an epidemiological perspective". In: *Clinical Oncology* 23.4 (2011), pp. 251–260.

[12] Andrew Chaves et al. "Improving the cyber resilience of industrial control systems". In: *International Journal of Critical Infrastructure Protection* 17 (2017), pp. 30–48. ISSN: 1874-5482. DOI: https://doi.org/10.1016/j.ijcip.2017.03.005. URL: https://www.sciencedirect.com/science/article/pii/S1874548217300501.

[13] *CIS Benchmarks*. https://www.cisecurity.org/cis-benchmarks/. Accessed: 2022-07-23.

[14] Ryan Cobb. *Judge*. Version 0.1. Mar. 2022. URL: https://github.com/cobbr/judge.

[15] *Creating Highly Available Clusters with kubeadm*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/. Accessed: 2022-03-29.

[16] davisking. *dlib C++ library*. Version 19.24. Mar. 2022. URL: https://github.com/davisking/dlib.

[17] Lily Puspa Dewi et al. “Server Scalability Using Kubernetes”. In: *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)*. 2019, pp. 1–4. DOI: 10.1109/TIMES-iCON47539.2019.9024501.

[18] Nikki John B. Florita et al. “IoT Resiliency through Edge-located Container-based Virtualization and SDN”. In: *2021 IEEE Asia Pacific Conference on Wireless and Mobile (APWiMob)*. 2021, pp. 39–44. DOI: 10.1109/APWiMob51111.2021.9435269.

[19] Md Ariful Haque et al. “Cyber Resilience Framework for Industrial Control Systems: Concepts, Metrics, and Insights”. In: *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*. 2018, pp. 25–30. DOI: 10.1109/ISI.2018.8587398.

[20] A H M Jakaria et al. “VFence: A Defense against Distributed Denial of Service Attacks Using Network Function Virtualization”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2016, pp. 431–436. DOI: 10.1109/COMPSAC.2016.219.

[21] *kubernetes-dashboard*. https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml. Accessed: 2022-03-29.

[22] Allan Liska. “Early findings: Review of state and local government Ransomware attacks”. In: *Recorded Future* 10 (2019).

[23] Logan O. Mailloux and Michael Grimaila. “Advancing Cybersecurity: The Growing Need for a Cyber-Resiliency Workforce”. In: *IT Professional* 20.3 (2018), pp. 23–30. DOI: 10.1109/MITP.2018.032501745.

[24] Yaser Mansouri and M. Ali Babar. “A review of edge computing: Features and resource virtualization”. In: *Journal of Parallel and Distributed Computing* 150 (2021), pp. 155–183. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2020.12.015. URL: https://www.sciencedirect.com/science/article/pii/S0743731520304317.

[25] *Manual:Interface/IPIP*. https://wiki.mikrotik.com/wiki/Manual:Interface/IPIP. Accessed: 2022-03-29.

[26] Nikhil Marathe, Ankita Gandhi, and Jaimeel M Shah. “Docker Swarm and Kubernetes in Cloud Computing Environment”. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. 2019, pp. 179–184. DOI: 10.1109/ICOEI.2019.8862654.

[27] Michael Donahoo Michael Coffey and Shaun Hutton. “RaaSI: Developers Guide”. In: (2022), p. 10. URL: https://baylor.box.com/s/m4bm4y3gwjtu33t8e2zzkskmvt86geoa.

[28] *MS12-006 MS12-006 - Important : Vulnerability in SSL/TLS Could Allow Information Disclosure (2643584) - Version: 1.3*. https://www.cvedetails.com/microsoft-bulletin/ms12-006/. Accessed: 2022-03-29.

[29] M Tamer Ozsu and Patrick Valduriez. “Distributed database systems: Where are we now?” In: *Computer* 24.8 (1991), pp. 68–78.

[30] Edward Ray and Eugene Schultz. “Virtualization Security”. In: *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. CSIIRW ’09. Oak Ridge, Tennessee, USA: Association for Computing Machinery, 2009. ISBN: 9781605585185. DOI: 10.1145/1558607.1558655. URL: https://doi.org/10.1145/1558607.1558655.

[31] Sara Shakeri, Niek van Noort, and Paola Grosso. "Scalability of Container Overlays for Policy Enforcement in Digital Marketplaces". In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–4. DOI: 10.1109/CloudNet47604.2019.9064090.

[32] Prateek Sharma et al. "Containers and Virtual Machines at Scale: A Comparative Study". In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. Trento, Italy: Association for Computing Machinery, 2016. ISBN: 9781450343008. DOI: 10.1145/2988336.2988337. URL: https://doi.org/10.1145/2988336.2988337.

[33] Federico Sierra-Arriaga, Rodrigo Branco, and Ben Lee. "Security Issues and Challenges for Virtualization Technologies". In: *ACM Comput. Surv.* 53.2 (May 2020). ISSN: 0360-0300. DOI: 10.1145/3382190. URL: https://doi.org/10.1145/3382190.

[34] Keith Stouffer, Joe Falco, and Karen Scarfone. "NIST special publication 800-82: Guide to industrial control systems (ICS) security". In: *Gaithersburg, MD: National Institute of Standards and Technology (NIST)* (2008).

[35] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. "Container-Based Orchestration in Cloud: State of the Art and Challenges". In: *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. 2015, pp. 70–75. DOI: 10.1109/CISIS.2015.35.

[36] Tsvetan Tsvetanov and Srishti Slaria. "The effect of the Colonial Pipeline shutdown on gasoline prices". In: *Economics Letters* 209 (2021), p. 110122. ISSN: 0165-1765. DOI: https://doi.org/10.1016/j.econlet.2021.110122. URL: https://www.sciencedirect.com/science/article/pii/S0165176521003992.

[37] Steven J. Vaughan-Nichols. "Virtualization Sparks Security Concerns". In: *Computer* 41.8 (2008), pp. 13–15. DOI: 10.1109/MC.2008.276.

[38] Xiaoguang Wang et al. "SecPod: a Framework for Virtualization-based Security Systems". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 347–360. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang.

[39] *Windows 10 Device Guard and Credential Guard Demystified*. https://docs.microsoft.com/en-us/archive/blogs/ash/windows-10-device-guard-and-credential-guard-demystified. Accessed: 2022-04-06.

[40] Chao-Tung Yang et al. "On improvement of cloud virtual machine availability with virtualization fault tolerance mechanism". In: *The Journal of Supercomputing* 69.3 (Sept. 2014), pp. 1103–1122. ISSN: 1573-0484. DOI: 10.1007/s11227-013-1045-1. URL: https://doi.org/10.1007/s11227-013-1045-1.

[41] Yanliang Zou et al. "User-level parallel file system: Case studies and performance optimizations". In: *Concurrency and Computation: Practice and Experience* 34.13 (2022), e6905.