Design and Evaluation of Security-Focused Service Meshes for Management of Microservice Deployments

Dalton A. Brucker-Hahn

M.S. Computer Science, University of Kansas, 2020
B.S. Computer Science, Kansas State University, 2018

Submitted to the graduate degree program in Electrical Engineering and Computer Science Department and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Chair: Alexandru G. Bardas

Andrew Davidson

Fengjun Li

Bo Luo

Huazhen Fang

Date defended: April 26, 2023

The Dissertation Committee for Dalton A. Brucker-Hahn certifies
that this is the approved version of the following dissertation :

Design and Evaluation of Security-Focused Service Meshes for Management of Microservice Deployments

Chair: Alexandru G. Bardas

Date approved: 04/27/2023

Abstract

Shifting trends in modern software engineering and cloud computing have pushed system designs to leverage containerization and develop their systems into microservice architectures. While microservice architectures emphasize scalability and ease-of-development, the issue of *microservice explosion* has emerged, stressing hosting environments and generating new challenges within this domain. Service meshes, the latest in a series of developments, are being adopted to meet these needs. Service meshes provide separation of concerns between microservice development and the operational concerns of microservice deployments, such as service discovery and networking. However, despite the benefits provided by service meshes, the security demands of this domain are unmet by the current state-of-art offerings.

Through a series of experimental trials in a service mesh testbed, we demonstrate a need for improved security mechanisms in the state-of-art offerings of service meshes. After deriving a series of domain-conscious recommendations to improve the longevity and flexibility of service meshes, we design and implement two proof-of-concept service mesh systems, *Anvil* and *ServiceFRESH* to facilitate frequent, automated rotation of security artifacts (keys, certificates, and tokens), within service mesh deployments. *Anvil* is a novel, standalone service mesh with automated, zero-downtime artifact rotations, while *ServiceFRESH* is a best-effort service mesh rotation module for use alongside a current state-of-art service mesh, Consul. The next prototype designed and developed as part of this work, *ServiceWatch* leverages these frequent security artifact refreshments and introduces a novel access control monitoring scheme. *ServiceWatch* effectively provides holistic monitoring and management of the microservice deployments it hosts. Further, *ServiceWatch* automatically isolates and removes microservices that violate the defined network policies of the service mesh, requiring no system administrator intervention. Extending this proof-of-concept environment, we design and implement a prototype workflow called *CloudCover*.

CloudCover introduces a *verification-in-the-loop* scheme and leverages modern service mesh tools, allowing easy adoption of these novel security mechanisms into state-of-art deployments. Under a realistic and relevant threat model, we show how our design choices and improvements are both necessary and beneficial to real-world deployments. By examining network packet captures, we provide a theoretical analysis of the scalability of these solutions in real-world networks. We further extend these trials experimentally using an independently managed and operated cloud environment to demonstrate the practical scalability of our proposed designs to large-scale software systems. Our results indicate that the overhead introduced by *ServiceWatch* and *CloudCover* are acceptable for real-world deployments. Additionally, the security capabilities provided effectively mitigate threats present within these environments.

Acknowledgements

I would like to thank everyone who has been involved with the research that I've conducted during my time as a graduate student. Specifically, my lab mates Kailani Jones, Yousif Dafalla, and Ron Andrews, of which I have had the pleasure of participating and contributing to their research as well. Also, I would like to thank the entirety of the Undo-Lab research group and the guidance and support of my advisor Dr. Alexandru Bardas. Alex has been a massive support, mentor, friend, and teacher for me through both my undergraduate and graduate research experiences. Without Alex's help and guidance, this work would not be possible. I would also like to thank Dr. Drew Davidson for playing an active role throughout my Ph.D. research and providing significant advice and support for this research. Lastly, I would like to thank all of the members of this committee: Dr. Alex Bardas, Dr. Drew Davidson, Dr. Bo Luo, Dr. Fengjun Li, and Dr. Huazhen Fang for their consideration of this work and the comments and advice that have helped refine this work.

The Madison and Lila Self Graduate Fellowship has played an important role in shaping me as a well-rounded graduate student and Ph.D. candidate. Without their financial support to focus on research activities and pursue research questions I find interesting, this work would not be what it is today. Additionally, the training and support provided throughout the four years of the program and the guidance from the Self Graduate Fellowship staff has been tremendous and has afforded me many opportunities and freedoms during my time as a Ph.D. student.

I would like to thank my family for the lifetime of support and confidence that they have shown me and my education. I'm so incredibly grateful to them and they have played a monumental role in the creation of this work. I thank my parents, Mark and Susan, for giving me every resource available and for fostering my interest in computers from an early age. The guidance and help from my brother, Quentin, was extremely important throughout my undergraduate career, and the friendship and support I have received helped encourage a pursuit of graduate education. I'd like

to thank my friends for their continued belief in me and an interest in my pursuits.

Lastly, I would like to thank my wife, Meagan Brucker-Hahn. Meagan has been the greatest support I could ever imagine and has pushed me to be more than I could have ever achieved on my own. Through long nights of stress, questioning, and struggle Meagan has been the support that has provided me strength to pursue this research and educational goal. Since the very beginnings of my research endeavours, Meagan has never failed to be interested, supportive, and curious of my work. The unending support, encouragement, and faith in my abilities and work have made the following document possible. Without Meagan, I would not be the person or researcher who I am today. For today and everyday, I am forever grateful for Meagan's love and support.

Contents

1 Introduction **1**

1.1 Research Goals 4

1.2 Contributions 5

2 Background and Related Work **6**

2.1 Enabling Factors for Microservices 7

2.2 Microservice Architectures 10

2.3 Service Meshes 12

2.4 Security Issues in Microservice Deployments 15

2.5 Related Work 16

3 Security Artifact Rotations in Service Meshes **18**

3.1 Introduction 18

3.2 Background and Motivation 21

3.2.1 Microservice Architectures 21

3.2.2 Service Meshes Overview 22

3.2.3 Research Gaps in Existing Service Meshes 24

3.3 Threat Model and Security Capabilities 25

3.3.1 Threat Model 26

3.3.2 Security Capabilities 27

3.4 Anvil Design 30

3.4.1 Core Service Mesh Functionality 31

3.4.2 Novel Security Features 32

3.5 Implementation 34
3.5.1 Anvil Development 34
3.5.2 Experimental Evaluation Setup 35
3.6 Evaluation 38
3.6.1 Consul (ServiceFRESH) Downtime Costs 43
3.6.2 ServiceFRESH Downtime Results 44
3.6.3 ServiceFRESH Microbenchmark of Downtime Costs 46
3.6.4 Overlay Service Mesh Performance Costs 47
3.7 Discussion and Limitations 49
3.8 Related Work 52
3.9 Summary 54

4 Mitigating Microservice Compromise in Service Meshes 55

4.1 Introduction 55
4.2 Background and Motivation 58
4.2.1 Microservice Architectures 58
4.2.2 Service Meshes Overview 59
4.2.3 Security Incident Handling in Existing Service Meshes 61
4.3 Threat Model and Security Capabilities 62
4.3.1 Threat Model 62
4.3.2 Security Capabilities 64
4.4 ServiceWatch Design and Evaluation 69
4.4.1 Malicious Service Mitigation 69
4.4.2 Attack Window 70
4.4.3 Access Control Verification 71
4.4.4 Experimental Evaluation – Access Control Verification 72
4.4.5 Theoretical Evaluation – Access Control Verification 73
4.5 Related Work 74

4.6 Summary 75

5 Multi-Hop Access Control in Service Meshes **77**

5.1 Introduction 77

5.2 Background and Related Work 80

5.2.1 Service Meshes and Microservice Architectures 80

5.2.2 Static Analysis of Source Code 81

5.2.3 Related Work 82

5.3 Threat Model 83

5.3.1 State-of-Art Shortcomings 84

5.3.2 Novel Attacks 84

5.4 CloudCover 86

5.4.1 Architecture and Design 87

5.4.2 Implementation 88

5.5 Evaluation 89

5.5.1 Network Overhead Calculation 89

5.5.2 Load Testing Experiments 92

5.6 Discussions and Limitations 94

5.7 Summary 97

6 Conclusions and Future Work **98**

Chapter 1

Introduction

The domain of software engineering and development has been revolutionized by the capabilities and novel methods provided by the advent of cloud computing. Namely, *microservice architectures* [63], a novel paradigm of software design, have emerged as a promising approach to capitalize upon the benefits and features of cloud infrastructure and containerization. Service meshes, a subset of DevOps, have become a leading solution to address the management and security challenges of the microservices domain [14].

The state-of-art in service mesh tools provide the key contributions of service discovery and management for microservices [90], load balancing for replicated and distributed microservices, and critically, security, such as mutual-TLS (mTLS) and authorization for microservice requests [62]. Requiring no microservice code modifications, service meshes deploy service proxies to facilitate the network communication and security benefits to deployed microservices [9, 46, 55]. This property has proven extremely beneficial to microservice developers, allowing them to decompose large software *monoliths* into independently operating *microservices* [63]. In this way, service meshes have allowed developers to abide by the principle of "separation of concerns" when it comes to constructing and designing microservices, focusing upon the business functionality of source code rather than the security and networking concerns of these systems.

Interest in service meshes has been growing in recent years, following a trend of adoption relative to other popular portions of the DevOps toolset, such as Kubernetes [69], a container orchestration platform. Additionally, the annual developer survey conducted by the Cloud Native Computing Foundation (CNCF) [17] has shown that Kubernetes and service meshes have increased in both adoption and popularity across 2020 and 2021 [18, 19]. As adoption of these

tools increases, so too does their maturity and the size of microservice deployments in production environments.

Despite the significant benefits and features that service meshes provide to microservice applications, the current state-of-art falls short of matching the dynamic and strong security needs of microservice deployments. Critically, modern service meshes lack many of the mechanisms needed to maintain fresh security artifacts (encryption keys/certificates and access control tokens leveraged in security mechanisms) [45]. In this way, the highly dynamic nature of microservice architectures and the frequent redeployment and configuration changes made in these environments are unmatched by the current offerings in state-of-art service meshes. Additionally, previous work has identified numerous security flaws and latent vulnerabilities in containerized applications and container networks. As attackers develop methods to exploit and leverage these threats against existing systems, the microservices domain faces significant threats to the security of entire deployments. Among these are: software supply chain vulnerabilities, closed-source third-party code, and underlying container vulnerabilities [77, 92]. With large software systems developed and deployed following the microservices paradigm, solutions to detect, alert, and mitigate compromise of service meshes is vital to maintaining control and the overall health of these systems. To make matters worse, while microservices are designed to be transient and replaced or modified regularly, the underlying service mesh instances are designed to manage and secure these environments for long periods of time. Lastly, the challenge of microservice explosion [64, 125] and highly complex dependency relationships between microservices have created novel attack vectors for adversaries that are able to gain a foothold presence within deployments. As real-world incidents and mishaps have shown [23, 92, 112], misconfigurations and misunderstandings regarding the dependencies between microservices are possible and are difficult to fully understand and capture.

The contributions presented in this work aim to address these security issues and challenges within the microservice domain. Through novel design choices and implementation, we present a series of prototype systems that improve upon the state-of-art in microservice security. Primarily, these prototype systems target the issues of (1) longevity of service mesh systems and maintain-

ing security artifact freshness for service mesh components; (2) detecting, alerting, and mitigating microservice compromise threats in operational service meshes; (3) enhancing service mesh access control mechanisms to defend against unaddressed threats in service meshes due to complex microservice dependency relationships.

1.1 Research Goals

This work attempts to address the following research goals:

1. Improve the long-term security and freshness guarantees in service mesh deployments.

2. Address the issue of microservice compromise in service mesh deployments.

3. Address vulnerabilities in service meshes caused by complex dependency relationships between microservices.

To achieve the research goals posed above, this work presents a series of prototype designs and system implementations that provide novel capabilities to the service mesh domain. These prototype systems are then evaluated with respect to performance, network overhead, and network latency incurred as a cost of operating these additional security measures within microservice environments. Evaluations and performance analysis were conducted both in cloud computing environments and in experimental testbeds that emulate the hardware and technology present within cloud computing infrastructure. In this manner, we attempt to provide realistic scenarios and environments to measure the feasibility of these designs for real-world application.

1.2 Contributions

By addressing the research goals presented above, this work makes the following contributions to the field of cyber security and network security:

1. *Anvil* / *ServiceFRESH* – Two initial, proof-of-concept frameworks for facilitating security artifact rotations in service meshes. *Anvil* is a standalone service mesh with security artifact rotation implemented as a key feature of the system and *ServiceFRESH* is a best-effort implementation to include security artifact rotations in a state-of-art service mesh, Consul. These proof-of-concepts are evaluated for effectiveness and efficiency with respect to performance in large-scale systems.

2. *ServiceWatch* – A novel design and implementation for detecting, alerting, and isolating microservice instances that are suspected to be malicious or misbehaving. The system is then evaluated based upon the network overhead imposed by the novel design.

3. *CloudCover* – A framework for improved access control in modern service meshes that considers novel threats to microservice deployments that are previously unaddressed by previous work or state-of-art implementations. We evaluate its feasibility for adoption in real-world systems by considering network overhead and latency imposed upon service-to-service requests in a realistic deployment.

Chapter 2

Background and Related Work

A growing rate of adoption and availability for cloud computing offerings have allowed developers and architects to restructure their systems to make use of the powerful abstractions and resources present within these environments. With global distribution of datacenters and resources [54], cloud computing infrastructure has paved the way for software engineers to construct highly available and resilient systems that can service a global audience while remaining online during some of the strongest denial of service attacks observed [82, 135]. However, these capabilities and a desire to redeploy and update software at increasingly higher frequencies have strained the management and deployment mechanisms that were used by administrators in the past [1]. Ushering in new challenges of infrastructure management, scalability, and monitoring within large-scale software engineering, cloud computing has not only brought dramatic benefits to the domain, but significant challenges as well.

A primary source of strain within these environments is an increasing interest in *microservice architectures* [63]. Traditional software, often developed and deployed as a single-server application, presents many issues and difficulties when attempting to utilize the available resources and features of cloud computing. This software model is often referred to as the *monolithic architecture*. However, the *microservices* approach decomposes the monolithic software application into atomic, singular-purpose application elements that collaborate with one another to achieve the same high-level capabilities and features. While this practice has enabled administrators to capitalize on the benefits of cloud computing and scalability, it has also increased the pressure and challenges of managing such systems at a global scale.

DevOps, a collection of modern tools and methodologies, has emerged to address these chal-

lenges and to streamline the *development* and *operations* tasks for modern software and microservice deployments [118]. Heavily emphasizing automation and "pipelines" for the process of software engineering and deployment, DevOps tools have appealed to administrators to answer the issue of *microservice explosion* [64, 125] that has come about due to the transition of systems from monolithic to microservice architecture. The remainder of this chapter explores the key enabling factors for this new software paradigm, the basic architecture of microservice applications, and the tools that are used to manage these deployments in operational environments. Additionally, we explore the security challenges and issues present within deployed microservice environments and the previous research that has examined this domain with efforts to improve the security and reliability of these systems.

2.1 Enabling Factors for Microservices

The changing landscape of cloud computing, and software engineering as a whole, has given rise to a higher level of distribution and parallelization of modern software applications. Key contributions within virtualization and novel services within cloud computing infrastructure have provided developers and system architects new methods and techniques to develop, package, distribute, and deploy software at rates dramatically higher than previous capabilities [63, 120]. These mechanisms and techniques have ushered in new software paradigms, namely microservice architectures, bringing significant benefits to system availability, reliability, and scalability.

In an effort to increase utilization and effectiveness of hardware platforms and servers, virtualization has been a key factor in the evolution of cloud computing. With the ability for cloud computing infrastructure to facilitate many developers and organizations on singular hardware platforms, virtualization technologies have improved and changed shape to meet the needs of these consumers [121]. With virtual machines and containerization becoming commonplace offerings in cloud computing infrastructure, the ability to provide high availability, replication, and global distribution to developers and system architects has reshaped the way consumers interact with software applications [136]. A key difference between virtual machines and container technolo-

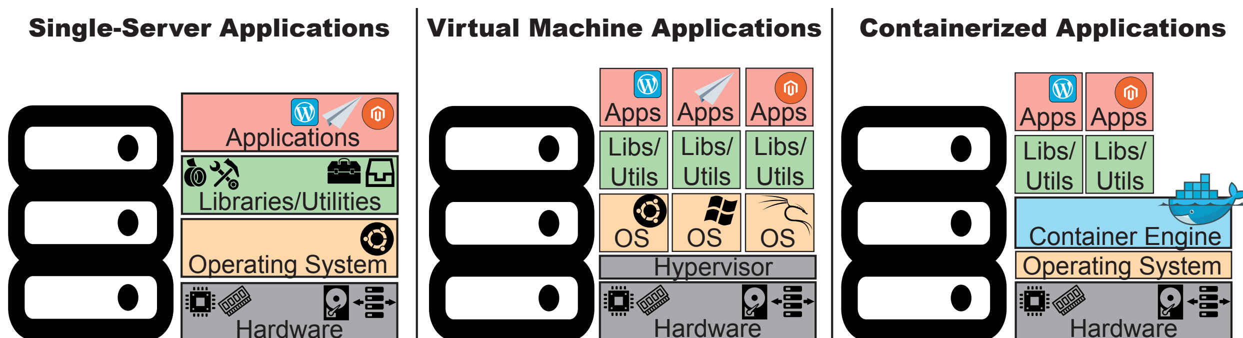

Figure 2.1: **Comparison of Virtualization Architectures**

gies is the depth of abstraction that is presented to software applications. In the case of virtual machines, a thin hypervisor is installed either atop the hardware itself (type-1/native hypervisor), or atop an existing, end-user operating system (type-2/hosted hypervisor) [87]. Virtual machines that are deployed into the hypervisor environment then have a dedicated operating system installed that contains all of the necessary kernel packages and libraries needed to facilitate the applications and features needed by software engineers. In contrast to this approach, containerization tools, such as Docker [27] or containerd [21], and operate similarly to a hypervisor, but are installed atop an existing operating system as "container engines" [87]. Containers themselves are not deployed with their own kernels, as is the case in virtual machines, but rather, share the kernel of the underlying operating system [99]. In containerized applications, requests to hardware features or support are first intercepted and interpreted by the container engine and are then passed to the operating system kernel to be handled [99]. Within a virtual machine hypervisor environment, such as a type-1/native hypervisor, virtual machines that request hardware features or support first make requests to the virtual machine kernel, which are then translated by the hypervisor before ultimately being fulfilled with hardware resources [87]. Figure 2.1 illustrates an abstract view of the differences between traditional, single-server applications, virtual machines, and containers. With novel platforms for hosting and deploying software applications, such as containers, the next key factor for enabling the microservice paradigm are the emerging deployment and management mechanisms that interact with cloud infrastructure.

With a desire to maximize the efficiency and rate at which software can be deployed and updates can be served in operational environments, new tools for creating, modifying, and destroying infrastructure have been developed. In recent years, the idea of Infrastructure-as-Code (IaC) [24, 107] and Software-as-a-Service (SaaS) [123] have become common terms in the domain of cloud computing and containerization. Tools such as Hashicorp's Terraform [53] have provided domain-specific language abstractions for cloud computing platforms that allow system administrators to create, modify, and destroy virtual machines, containers, and even cloud-specific resources, such as load balancers. Through these mechanisms, administrators can quickly take updates and feature additions to software source code and integrate them into operational environments at increasingly high speeds [123]. Additionally, SaaS offerings through cloud computing vendors, such as AWS's Aurora database [3] or ElastiCache [4] caching service provide ready-to-run configurations of commonly used tools to system administrators for easy deployment and integration into their software architectures [123]. Once infrastructure is reserved or deployed to the cloud environments, configuration management tools, such as RedHat's Ansible [109], Puppet [102], or Chef [104] are used to install, configure, and modify software atop deployed containers or virtual machines. These tools provide a high degree of automation to the workflows of software engineers and system administrators, furthering the capabilities of cloud computing to provide fast and reliable updates to software.

Finally, with the size, complexity, and degree of distribution in these new environments, orchestration management tools, such as Kubernetes [69] have evolved to fill the gaps and address challenges in this domain. Kubernetes provides centralized orchestration, management, and scaling for containerized applications. These key functions enable software engineers to more effectively transition software from containerized source code to globally distributed software systems that are capable of handling 42 million unique desktop users and over 126 million mobile users per month, as observed at Amazon [106]. Kubernetes extends the container abstraction and provides key features such as distributed networking for containers. Acting as an "operating system for containers", Kubernetes has continuously increased in its rate of adoption and deployment in

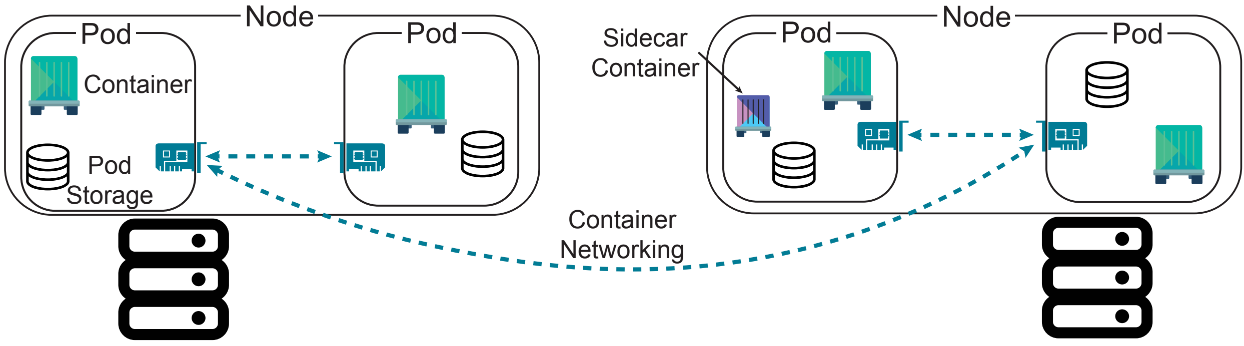

Figure 2.2: **Overview of Kubernetes Pods and Sidecar Containers**

real-world environments [18, 19]. The concept of Kubernetes "pods" is also an important factor for how containers coordinate and collaborate with one another to achieve the high-level business goals of deployed software applications [72]. As shown in Figure 2.2, pods are collections of storage elements and containers that effectively share a common network boundary. Deployed alongside one another on common hardware, containers deployed in pods operate adjacent to one another, commonly referred to as "sidecar containers" [12]. This resource structure provides extremely fast network requests and interactions for containers and storage deployed within the same pod [12]. Sidecar containers have been a key concept and enabling factor for the security of microservices when used within service meshes. Additional details about service mesh structure and design discussed below in Section 2.3. Through this unique feature set and deployment model, software applications have transformed to be highly distributed, independently operating portions of a much larger application that collaborate with one another. This paradigm has been coined the *microservice architecture* paradigm and has grown dramatically in its adoption and popularity in large-scale software systems.

2.2 Microservice Architectures

Leveraging the evolution and changes in hardware abstraction and the support of novel automation and management tools, microservice architectures have emerged as a modern approach to software engineering and deployment. Additionally, the key capabilities provided by containerization and

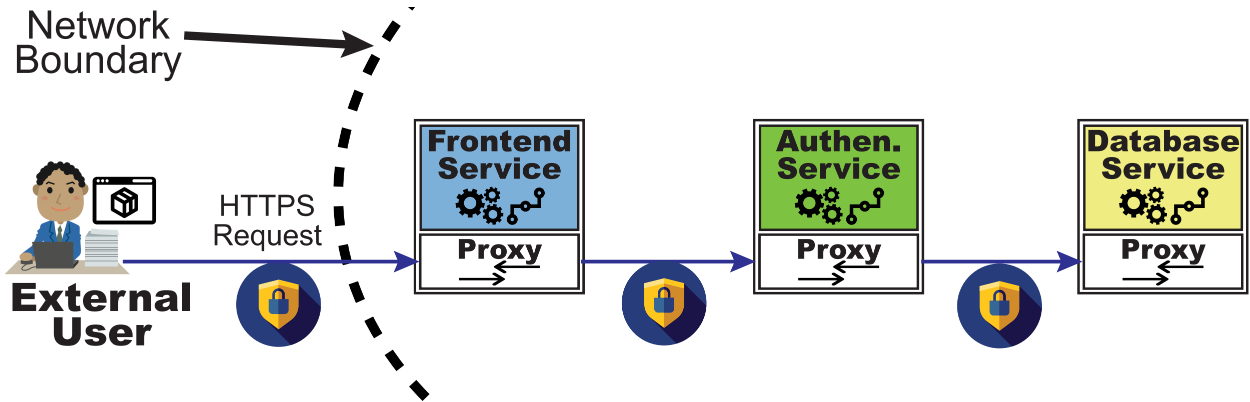

Figure 2.3: **Basic Microservice Deployment with Service Mesh Proxies Installed**

container orchestration platforms have pushed software engineers to develop and deploy their applications in the form of clusters, or "swarms", of unique, atomic microservices. Through network requests and collaboration between these microservices, a high degree of availability, reliability, and scalability are achieved to facilitate the high-level goals of the software application.

The shift in software engineering paradigms from traditional, single-server applications to clusters of distributed and replicated microservices has been a recent development enabled by a long series of novel tools and mechanisms. The decomposition of single-server applications, referred to as *monoliths*, to atomic, often single-purpose components, referred to as *microservices* has derived many new benefits and accompanying challenges for software engineers [63]. These atomic components communicate and collaborate in order to achieve the high-level business goals of the software system. As part of the microservices design, what were once *intra*-service connections have now become *inter*-service connections [110], crossing network boundaries and bringing about new challenges and concerns with respect to security and reliability. Additionally, as software is separated and separation of concerns is conducted upon a system, the issue of *microservice explosion* emerges [64, 125]. Microservice explosion refers to the tendency for large-scale software systems that follow the microservices architecture to experience a dramatic increase in the growth of unique software components that are managed and deployed as part of the system. For example, what may have once been a single-server login application to a payroll system may now involve many microservices that coordinate with one another using network requests to facilitate

the same functionality. A frontend user interface that displays the login page and collects user input may compose one microservice, while the authentication mechanism for checking user credentials comprises another. It is also reasonable to assume that the storage element for storing user credentials in a hashed and salted form comprise a third microservice. This basic microservice model is illustrated in Figure 2.3. In effect, the microservice explosion issue presents new management and complexity problems for this domain, placing additional strain on the current state-of-art tools intended to manage and connect these environments. Accompanying the challenges of scalability and coordination of microservice deployments, the sheer complexity of these environments can grow to degrees that are difficult to understand for single administrators or engineers. As seen in recent mishaps at Twitter [23, 112], the complexity and difficulty in understanding all of the unique dependencies and relationships between microservices can often end in unavailability or system downtime due to configuration errors.

Platform orchestration tools such as Kubernetes attempt to address these challenge of deploying and managing containerized applications for scalable and distributed environments. With features such as auto-scaling, replication, and service abstraction, Kubernetes has been a key technology for the adoption and growth of microservice applications [69]. While providing key mechanisms for deployment and infrastructure support for microservice applications, another key concern for simplifying the development and revision process for microservices is the abstraction of networking and security concerns to other components deployed in the cluster. Namely, these concerns have been accommodated by service meshes [10, 47, 60, 98], the latest iteration in DevOps tools and methods to handle microservice architectures.

2.3 Service Meshes

Service meshes are the latest iteration in a series of advancements made by DevOps tools to address the challenges and needs of the microservices domain. Namely: once microservices and the containers that host them have been deployed to an environment, how are they able to locate and communicate with one another, and how are they secured?

While the current state-of-art service meshes vary in terms of design and deployment strategies, all modern service meshes exhibit similar properties. Additionally, they share a common set of goals and functionality they provide to the microservices they manage. Service meshes deploy service proxies into the microservice environment and operate as “agents” on behalf of the microservices. Figure 2.3 depicts this proxy-service relationship and shows how an external user would interact with the microservice deployment via the entrypoint service’s service proxy. Service proxies operate at the network boundary of a microservice, or “pod” in the case of a Kubernetes cluster, to intercept network traffic passing between microservices. Service proxies also communicate with one another to maintain an accurate view of the access control policies deployed to the cluster and to coordinate identity and authentication among other service proxies [70]. The management operations and tasks within a service mesh occur within the “control-plane” [70] of the service mesh and are separated from the logic and traffic that occurs between microservices. The service-to-service requests between microservices occurs over the “data-plane” [8] of the service mesh. All data-plane traffic first passes through the service proxy of the cluster and may encrypted using mutual-TLS (mTLS) [61]. By applying mTLS for service-to-service traffic in the cluster, the key security properties of confidentiality, integrity, and authentication are provided to microservices without the need for any software implementation within the microservice itself. Additionally, all modern service meshes provide some form of access control to facilitate proper authorization for service-to-service requests in the deployment [51]. While all modern service meshes strive to meet these security goals for microservice deployments, the design and structure of service meshes still varies among the current state-of-art.

Two competing strategies in service mesh design in the current state-of-art tools are *platform independent* service meshes and *overlay* service meshes. Hashicorp’s Consul [47] is an example of a platform independent service mesh in that it does not *require* an underlying Kubernetes implementation to operate, instead allowing system administrators to deploy Consul as a separate binary application that functions alongside the microservice code [48]. However, despite it’s design, Consul may still be used in conjunction with a Kubernetes platform, able to operate as either a platform

independent service mesh, or an overlay service mesh at the administrator's discretion [50]. In contrast to this approach, overlay service meshes, such as Istio [60] and Linkerd [10] heavily depend upon an underlying Kubernetes deployment to facilitate certain service mesh processes and operate correctly. The support of an underlying Kubernetes implementation does provide a series of benefits and capabilities to the service mesh, such as service discovery and management, but limits the applicability of their deployment in certain environments.

Aside from the differences in deployment options, modern service meshes also differ in their management structure. For example, in overlay service meshes, the underlying Kubernetes platform is used to facilitate control-plane operations and tasks such as security artifact rotations of some TLS keys and certificates used within the environment [71]. Alternatively, Consul creates its own "quorum" structure to provide these processes to administrators, but does not automate them. Within a Consul deployment, administrators are responsible for selecting a subset of the deployed service proxies to operate in a "leader-follower" fashion. These nodes comprise the cluster's "quorum" where tasks such as logs for service mesh state are held and coordinated, new certificates and keys are generated, service discovery and management is conducted, and configuration changes for the cluster are possible. Consul's practice of distributed responsibility and management are not unique to the domain however. Kubernetes clusters may also be deployed in "high-availability" mode which provides redundancy and load balancing of control-plane tasks in a similar manner to the Consul quorum.

Providing clear benefits to enhance system administrators' abilities in managing and coordinating tasks in their deployments, service meshes are a clear advancement within the domain of microservices and cloud computing. However, despite the key capabilities and features offered by service meshes to microservice deployments, there exist open challenges and concerns relating to the long-term security of service mesh platforms and the potential for microservice compromise and expansion of attacks within this domain.

Tool	Security Mechanism	Available in Tool?	Enabled by Default?	Default Lifetime	Revocation	Redistribution
Consul	Cluster Message Encryption	Yes	No	∞	No	No
	Service Message Encryption	Yes	No	1 year	Yes	No
	Cluster Access Control	Yes	No	∞	Yes	No
	Service Access Control	Yes	No	∞	Yes	No
Linkerdv2	Cluster Message Encryption	No	No	N/A	N/A	N/A
	Service Message Encryption	Yes	Yes	24 hours	Yes	Yes
	Cluster Access Control	No	No	N/A	N/A	N/A
	Service Access Control	Yes	No	∞**	No**	No**
Istio	Cluster Message Encryption	No	No	N/A	N/A	N/A
	Service Message Encryption	Yes	No	Ext Tool [62]	Ext Tool [62]	Ext Tool [62]
	Cluster Access Control	No	No	N/A	N/A	N/A
	Service Access Control	Yes	No	∞**	No**	No**
Kubernetes	Cluster Message Encryption	No*	No	N/A	N/A	N/A
	Service Message Encryption	Yes	No	1 year	Beta	Beta
	Cluster Access Control	Yes	No	∞	No	No
	Service Access Control	Yes	No	∞	No	No

Table 2.1: **Security Mechanisms in Service Mesh Tools** – A summarized view of the security mechanisms available in each service mesh tool analyzed, which mechanisms are enabled by default, and additional details about the actual implementations. *Pod-to-pod encryption left to third-party implementation [72]. **Inherited from Kubernetes' Role-Based Access Control system [10, 62]. *Table appears in [45].*

2.4 Security Issues in Microservice Deployments

A growing number of vulnerabilities exploited in containerized applications and the challenge of microservice explosion has strained the tools and environments that host and manage microservices. Additionally, as adoption of microservice architectures increases, the potential for these environments to be targeted by adversaries increases.

With the relative youth of service mesh tools in real-world systems, the security capabilities and support in this domain remains relatively untested. Previous work by Hahn, *et al.* [45] explored the state-of-art in service mesh technologies and their security under different configuration settings. Namely, under default settings, many modern service meshes provide limited security to the systems they manage, leaving them vulnerable to exploitation by attackers. Due to the fact that these systems are designed to host and manage clusters of hundreds to thousands of microservices, these environments can be extremely appealing to adversaries. Once compromised, adversaries may

leverage the significant resources of a microservice deployment as a means of launching denial of service attacks, cryptomining operations, and spam campaigns. Additionally highlighted as part of this work, many of the service meshes studied fail to incorporate a number of desirable security mechanisms that other domains have adopted. Namely, infinite or extremely long default lifetimes of employed encryption certificates exist in nearly all modern service meshes. To compound matters, at the time of the study, many service meshes failed to provide meaningful mechanisms to revoke or redistribute fresh certificates should a portion of the microservice deployment need to be taken offline or redeployed. Under these circumstances, stale, or unmanaged, certificates may remain active in the service mesh, providing adversaries the potential for legitimate communication with benign portions of the microservice deployment. The key takeaways from this study are presented in Table 2.1. In addition to the key vulnerabilities and issues presented by Hahn, *et al*. [45], other research works have examined the broader domain from a range of perspectives, including microservice compromise, container networking issues, and access control for different applications in the DevOps space.

2.5 Related Work

Previous work in the microservice and service mesh security domain have sought to address a range of issues and challenges within these environments.

In recent years, growing concern and a range of vulnerabilities and exploits emerging within microservice applications and the container ecosystem have driven a range of research and development to alleviate these issues. With vulnerabilities, such as Log4j [85, 128] making headlines and leading to significant damage [36, 132] work to understand the existing vulnerabilities and security issues in containerized applications has emerged [66, 126]. To address these vulnerabilities and challenges, various works have explored applying principle of least privilege to microservice systems that were constructed from decomposed monolithic applications [108]. Others have examined the potential for utilizing security monitoring in container systems [137]. Additionally, work exploring the dangers of inherent trust within deployed microservices has been considered [124].

Work by Nam, *et al*. [89] explores security of container networks and the vulnerabilities exposed by some container networking systems. They provide a proof-of-concept framework for mitigating these threats and enforcing proper container networking policies. This work includes efforts to improve upon the existing state-of-art in service mesh access control through our prototype system CloudCover. CloudCover examines service-to-service traffic within microservice deployments that operates at higher layers in the networking stack, specifically with service mesh networking rather than container networking. CloudCover provides defenses against existing vulnerabilities caused by complex microservice dependency relationships. However, these methods may be combined to provide a defense-in-depth approach. Additionally, previous work has considered proper access control in a serverless application context [2, 25, 116]. However, this work is limited in scope due to its implementation for serverless applications specifically. Instead, our system, CloudCover, applies system-wide access control for service meshes which may be deployed alongside serverless applications due to their ability to be adopted into service mesh clusters [93]. Work by Li, *et al*. [76] has investigated the issue of inherent trust within microservices and how this practice may be exploited by an insider threat. In a similar manner, throughout this work, we adopt threat models that assume the potential of insider threats or microservice compromise with an attacker that attempts to take malicious actions within microservice deployments. However, we expand upon this work by considering a threat model where the complex dependency relationships and connection maps within microservice deployments enable an attacker to achieve goals if not mitigated. Specifically, in [76] only single-hop service-to-service connections are considered and secured when in actuality, multi-hop paths exist in real-world deployments. Due to this, isolated access control methods are insufficient and a holistic, system-wide access control methodology is necessary. CloudCover expands upon this work by capturing these relationships and providing a proof-of-concept framework to address this issue.

Chapter 3

Security Artifact Rotations in Service Meshes

3.1 Introduction

Adapting to new developments in software engineering design, *service meshes* have emerged as a promising solution to manage and coordinate activities within *microservice architectures* [15, 100, 110]. Modern software engineering has trended towards a microservice paradigm and away from the traditional, monolithic structure of software applications [44, 63, 110]. Through separation of concerns and segregation of software components, the size of deployments has grown dramatically and the issue of *microservice explosion* has emerged [64, 125].

Service meshes [37], a growing subset of tools within the wider collection of methods and practices referred to as *DevOps*, have filled the role of controlling, maintaining, and connecting collections of single-purpose microservices. Microservices, often hosted within deployed containers, but also virtual machines, or even servers, collaborate with one another to achieve high-level business goals and logic [63]. This design paradigm is broadly referred to as the microservice architecture and has seen a dramatic growth in popularity in recent years with a range of enterprise organizations transitioning their infrastructure to this new paradigm [16, 31–34, 67, 129]. With a broad list of responsibilities and capabilities, service meshes bear the weight of providing infrastructure management as well as network coordination and connection. For instance, with the segregation and separation of different aspects of a microservices-based software deployment, what were once *intra*-service connections have become *inter*-service connections. As such, connections between services must now traverse network boundaries. Due to this, service meshes are now responsible for coordinating and securing these connections connections to uphold the

necessary security guarantees.

Despite being charged with responsibilities of network security, current service mesh designs fail to meet the vital needs of this novel domain. Examples of these shortcomings include: lack of meaningful, domain-adapted security features, prevention of cryptographic key and certificate rotation capabilities, and requirement of high-cost underlying container orchestration platforms to function correctly [10, 60, 73, 98]. As such, the current state-of-art in service mesh designs lack the means to effectively ensure the security of deployments that they have been designed for [45]. These drawbacks conflict directly with the appeal of microservice architectures and their goals of rapid deployment, always available resources [63].

The youth of service meshes has allowed for competing designs and structures of tools to exist in competition within the domain (e.g., Consul [47], Istio [60], Linkerd [10], OSM [98], Kuma [73], etc.). However, *all* modern service meshes, fall short in matching the needs of microservice deployments, either in the longevity of platform security or the performance costs associated with underlying platforms [45]. Despite these concerns and challenges, a growing number of service meshes are being labeled as "graduated" according to the Cloud Native Computing Foundation (CNCF) and marked as "ready for production environments" [127]. Linkerd has already been labeled as a CNCF graduated project along with Kubernetes [22, 88]. Additionally, the Istio service mesh has been submitted by Google for inclusion in the CNCF [75]. Growing adoption of service meshes along with security concerns increases the potential threats posed by the use of these tools along with an increase in appeal to potential adversaries to target them.

Our prototype systems, *Anvil* and *ServiceFRESH*, directly address these areas of concern and meet the needs of microservice architectures. Through key longevity features and novel security functionality, Anvil and ServiceFRESH explore the unique capability of security artifact rotations for service meshes. Focusing on feasibility and cost-effectiveness with respect to the security-performance tradeoff, we aim to more closely meet the dynamic and flexible needs of the microservices domain. Anvil is, to the best of our knowledge, the first service mesh that provides automated, synchronous, lively rotation of *all* security artifacts utilized within a service mesh.

ServiceFRESH, in contrast, is a best-effort framework for integrating these capabilities within a current state-of-art service mesh. For the remainder of this work, we use the term *security artifact* to represent any security materials such as access tokens, encryption keys, or encryption certificates that may be utilized within a service mesh. Through a demonstration of these key design decisions, without sacrificing the valuable feature-set present within service meshes, Anvil and ServiceFRESH embody beneficial security design for future integration in state-of-art service meshes.

Due to the diversity in design and implementation architectures of current service meshes (Consul, Istio, Linkerd, OSM, Kuma), a direct lateral performance comparison of Anvil to these tools is difficult and is often unclear with respect to Kubernetes' [69] relationship with overlay service meshes. Our primary goal is to demonstrate the feasibility and benefits of the security features our proof-of-concept system, Anvil, relative to our best-effort implementation of ServiceFRESH. We present a relevant threat model and security analysis of the state-of-art in service meshes, highlighting the design and implementation shortcomings in the current service mesh offerings. We also demonstrate how the superior security design of Anvil more closely matches the needs of microservice architectures.

To evaluate the feasibility of Anvil's design features in practice, we use 29,400 unique CPU and RAM data points collected in a controlled, testbed environment. We find that the security features present in Anvil, such as security artifact rotations, do not incur significant processing costs (up to 1% utilization when comparing median values across experiment trials). Next, we show that Anvil's lively rotation capability, if implemented alongside a current state-of-art service mesh, Consul, creates inherent system downtime. Through a timing and performance analysis of this prototype, ServiceFRESH, we report upon the system downtime incurred, which is undesirable in microservice deployments and is not present in Anvil's operations. Lastly, we analyze the implementation and operation costs of a representative overlay service mesh, Istio, being used with the orchestration platform Kubernetes. Despite bringing Kubernetes to its most lightweight, baseline implementation, the deployed Istio service mesh was at best matching Anvil's performance across

85,200 unique CPU and RAM utilization data points. However, Anvil showed clear benefits to system security through improvements to the long-term health of a service mesh by automatically rotating *all* certificates, rather than requiring manual administrator intervention [11, 59, 74, 96].

Through the key features of lively, automated security artifact rotations and novel, microservice security management functionality with minimal overhead, Anvil demonstrates a key understanding of the environments in which service meshes are deployed and more closely addresses the needs of these areas. As part of this work, we provide the following contributions:

- We analyze service mesh state-of-art designs and identify security shortcomings and limitations that we then accommodate within our prototype solution, Anvil.
- We present Anvil, a service mesh platform with novel security features and lively, automated security artifact rotations.
- We perform experimental evaluations of Anvil and state-of-art service meshes, showing the benefits of Anvil relative to alternative tools with respect to performance and downtime.

3.2 Background and Motivation

Advancements in cloud computing and virtualization, have refined the architectures and design of modern software systems. Current trends in software engineering and cloud computing have encouraged adoption of microservice architectures managed by orchestration tools, such as service meshes.

3.2.1 Microservice Architectures

Traditionally, software has been designed as a singular, whole product. This practice has been referred to as the *monolithic software architecture* [84]. However, while this practice is simple to understand and straightforward in terms of design, the scalability, manageability and speed at which such a system can be deployed and revised is highly limited. Due to this, modern software

engineering practices have trended towards the separation of software components or modules into *microservices* [130]. These single purpose, atomic elements of a larger software system are deployed in tandem with other components and collaborate to achieve high-level business goals such as user authentication or product purchasing [83]. Figure 2.3 illustrates a very simplified user-login microservice deployment with three services handling the "frontend" user-interface, credential "authentication" management, and a credential storage system in the form of a "database".

Bringing an increase in the speed to develop, and availability of, online applications, *microservice architectures* [63] are an emerging software engineering paradigm. Additionally, virtualization has been a key enabling technology for the advent of microservice architectures in that many deployments leverage virtual machines or containers to host the microservice code [64]. With extremely fast deployment times and increased efficiency by leveraging shared resources, containerization played a major role in the adoption and growth of the microservice paradigm [15, 100, 110]. However, as enterprise software expands and the quantity of microservices deployed grows, the management and coordination of these services is increasingly difficult. Service meshes are the latest solution, in a series of developments, to address the issue of microservice explosion in large-scale software. [44, 63, 110]

3.2.2 Service Meshes Overview

Service meshes embody the latest iteration of the DevOps toolset. Namely, due to the aforementioned microservice explosion challenge, managing and coordinating microservice "swarms" becomes increasingly difficult as the size of deployments increases [65, 83]. Figure 2.3 demonstrates this network architecture with deployed microservices operating a "service proxy" alongside their microservice code. The service proxy, a key element of the service mesh handles the network connections and security responsibilities of the microservice deployment, separating these concerns from the microservice code [9, 46, 55].

The current production-ready, state-of-art service mesh tools are Consul, Linkerd, Istio, and Kuma. [10, 47, 60, 73] (OSM, another notable service mesh, is, at the time of writing, in a pre-

Figure 3.1: **Service Mesh Popularity** – Growth in popularity of service mesh tool repositories and Kubernetes plotted over time in GitHub [38] stars. GitHub metrics collected from [13], a tool that accesses the GitHub API to collect historical star values.

release stage, but shares many properties with other overlay service meshes. [98]) Interest in service meshes has been growing in recent years, following a trend of adoption relative to other popular subsets of the DevOps toolset. Figure 3.1 illustrates the growing popularity and increased interest in service mesh tools by measuring the number of “stars” that service mesh repositories have on GitHub [38]. Additionally, Kubernetes [69], a platform for service mesh tools is becoming widely adopted and present in enterprise systems.

While the term service mesh is agreed upon within the community and the feature-set that is provided by service meshes is fairly consistent among the current state-of-art, the implementation and design choices of service mesh technologies vary. For example, Consul is a standalone, platform-independent service mesh that can be utilized as a single binary on a range of operating systems and technologies. In contrast, Istio and Linkerd are both overlay service meshes, reliant upon an underlying Kubernetes infrastructure to provide the necessary structure and capabilities for the service mesh to operate correctly. Without an underlying Kubernetes implementation, all overlay service meshes (Istio, Linkerd, OSM, Kuma, etc.) are unable to provide any of the pur-

ported functionality to a microservice deployment, meaning that expertise with Kubernetes and the associated operational costs of a Kubernetes deployment are inherent when utilizing these tools. Within Anvil and Consul, rather than depending upon an orchestration platform, like Kubernetes, orchestration and management of the cluster is conducted by a specially configured group of higher-privilege nodes referred to as a "quorum" that handles the consensus and management functionality for a deployment. Kubernetes, when configured for enterprise environments, also operates in a "multi-master" mode, where all masters are replicated with identical security artifacts, and replicate state changes amongst the group of "masters". While the design and structure of Kubernetes is quite different relative to platform-independent service meshes, such as Consul and Anvil, these tools can be adapted and utilized within Kubernetes deployments [50], showing a greater flexibility and freedom over the more strict overlay service meshes.

3.2.3 Research Gaps in Existing Service Meshes

While appealing to software engineers and system architects based upon the purported feature set, service meshes introduce new complexity and challenges to these environments as well. Current, state-of-art service mesh design lacks many of the attributes and capabilities necessary to match the needs of the microservice domain.

System Longevity: With the size, complexity, and highly interconnected nature of microservice deployments, the investment of system administrators and software engineers to build long-lasting and functional environments is clear. However, as shown in [45] and mentioned previously, modern service meshes lack many features to maintain these microservice deployments over time. For example, within the Consul service mesh, lifetime, shared encryption keys are enforced as part of system design and there is a lack of any rotation and refreshment mechanisms within the tool [52]. Similarly, due to the fact that overlay service meshes require the use of an underlying Kubernetes environment, these service meshes inherit the limitations present in Kubernetes. One such limitation is the need for manually rotating the certificates that govern the control-plane (communication network for orchestration components [70]) of Kubernetes deployments [11, 59, 74, 96]. To enable

long-lived systems that maintain security over time, greater support for dynamic security methods are needed in service meshes.

Zero Downtime: Similarly to the long-term health of deployed systems, another equally important characteristic of the microservice deployments is the desire to maintain high degrees of system uptime [28, 80, 134]. With the incredibly intertwined nature of microservices and their deployments, losing uptime in a portion of the deployment, or losing uptime across the entire deployment, can be extremely costly. With upwards of millions of U.S. dollars lost within an outage of less than 15 minutes [131], corporations like Amazon depend upon service meshes to maintain uptime and connectivity of services to an extreme degree. Other major retailers see similar losses in revenue due to lost uptime, as reported by [42]. Already present in overlay service meshes, lively rotations of security artifacts within the data-plane (communication network for microservices [8]) are present [11]. Despite this, refreshment of the full set of security artifacts in modern service meshes is lacking from *all* available tools, resulting in inherent system downtime to facilitate this task [11, 59, 74, 96]. In the case of Consul, due to runtime ingestion of access control policies and generation of certificates and policies, this downtime can be upwards of 4 minutes to refresh the artifacts of a 1000 node service mesh cluster. Additional details regarding our best-effort prototype to provide security artifact rotation in Consul and its performance are included in Sections 3.5 and 3.6.

3.3 Threat Model and Security Capabilities

Our proof-of-concept system Anvil considers similar potential threats as other service meshes, but demonstrates novel design choices with respect to security. To combat standard network threats and external adversaries, Anvil adopts a similar "zero-trust" networking approach compared to alternative service meshes. Novel to Anvil, however, is long-term system security through frequent, automated security artifact rotations.

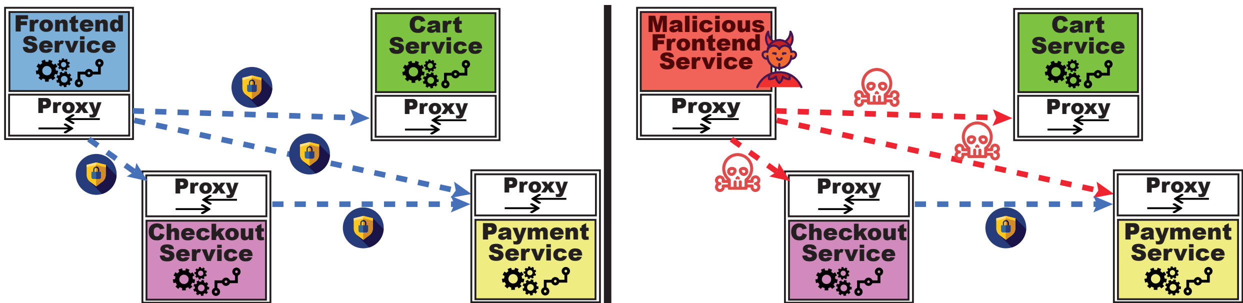

Figure 3.2: **Threat Model Overview** – External network threats are adversaries that have no prior access and no valid permissions within a microservice deployment. They may only interact with public-facing instances and attempt to eavesdrop on shared networks. Insider threats are valid entities within the microservice deployment, but are adversary-controlled.

3.3.1 Threat Model

The highly distributed nature of microservice deployments and the ability to utilize cloud infrastructure to distribute resources globally create opportunities for adversaries to threaten service meshes. We consider the two broad categories of external network threats and insider threats as part of this work.

External Network Threats: One primary category of threats that we account for as part of our representative threat model are adversaries that have no prior presence within the microservice deployment or service mesh environment. Due to the highly distributed, potentially global or multi-cloud, deployments of microservices, network connections initiated between various components can often traverse shared, uncontrolled network boundaries [54]. As such, it is extremely important to consider the potential for skilled adversaries to monitor network traffic in attempts to eavesdrop or manipulate traffic moving across these networks. Additionally, because traffic leaves the controlled network boundaries, public-facing endpoints will inevitably be exposed to the public IP address space in order to accommodate this functionality [58]. This opens the potential for adversaries to target these instances with various network-based attacks. Aside from the range of standard, network-level attacks available to external adversaries, we also consider the more severe potential of insider threats that are the result of a compromised node or container.

Insider Threats: Growing concerns over the potential for container compromise [77, 119], third-

Service Mesh Tool	Artifact Rotations	Automated Rotations	Zero-Downtime Rotations	Flexible Deployments	Attack Window (Data Plane)	Attack Window (Control Plane)
Anvil	✓	✓	✓	✓	1 minute*	1 minute*
Consul	⊗	⊗	⊗	✓	∞	∞
Istio	✓	◐	✓	⊗	24 hours [59]	≥1 year [59]
Linkerd	✓	◐	✓	⊗	24 hours [11]	1 year [11]
OSM	✓	◐	✓	⊗	24 hours [96]	1 year [96]
Kuma	✓	◐	✓	⊗	24 days [74]	≥1 year [74]

Table 3.1: **Security Analysis of State-of-Art Service Meshes –** Current capabilities and features of state-of-art service meshes are compared against the Anvil prototype design. Key features such as zero-downtime, automated security artifact rotations, and platform-agnostic deployments are all present within Anvil. However, the comparable state-of-art *all* lack one or more of these features. **1 minute is the current configured time for rotations within Anvil, however, it is customizable and may be reconfigured as needed for environment or domain needs.*

party software application compromise, and software supply chain vulnerabilities [92], increase the potential for an adversary to gain access to microservice deployments via deployed applications. Within this study, we consider the potential for single-node or single-microservice compromise to be possible for an adversary. From this position, we assume that an adversary may initiate new network connections to other portions of the service mesh, such as other microservices. However, we exclude the quorum instances from potential insider threat compromise because they are known to have higher levels of privilege and responsibility within a service mesh deployment and are not intended to host microservice application code. In this way, we believe that system administrators have a larger incentive for increased scrutiny regarding these components within the service mesh to prevent compromise.

3.3.2 Security Capabilities

To combat these described threats, we make use of some traditional security mechanisms found within state-of-art service mesh design, but we also introduce novel security features unique amongst service meshes currently available. Aiming to uphold strong security guarantees, but with a focus upon long-lived systems, Anvil provides security design that better meets the needs of the microservice domain.

Zero-Trust Networking: The premise of zero-trust networking is one that matches the needs

of microservice deployments quite closely. Rather than having any inherent trust within the deployed components, security mechanisms are used to verify all connections, identities, and content exchanged between microservices within a service mesh [111]. Similar to the current state-of-art, Anvil also adopts these accepted security mechanisms to provide zero-trust networking capabilities. Namely, Anvil makes use of symmetric key encryption for gossip messages exchanged between nodes, x509 TLS certificates to create and secure mutual-TLS connections between services, and access control policies and tokens to ensure proper authorization for requests made between services in the deployment. While the exact implementation of these security mechanisms vary slightly among service meshes, all state-of-art tools either provide a form of each of these mechanisms or depend upon these mechanisms to be provided by underlying tools, such as Kubernetes [52, 62, 71, 78]. With this in mind, it is possible to construct a very secure and effective defense for microservice deployments from modern service mesh offerings. However, as previously mentioned, the long-term support of this security dwindles in the current state-of-art service meshes over time as certificates begin to approach expiration and access control tokens and symmetric encryption keys are exposed for long periods of time. Anvil combats these issues by providing automated, synchronized rotation of security artifacts throughout the lifetime of the deployment.

Deployment Longevity: The highly dynamic nature of microservice deployments and the scaling-up and scaling-down capabilities provided by cloud computing platforms have strained and challenged the current management solutions available to administrators [63]. As such, questions regarding the ability for service meshes to be long-lived platforms arise [45]. Consul, a platform-independent service mesh fails to provide meaningful support for rotating and refreshing security artifacts used within deployments without fully redeploying the entire service mesh infrastructure [52]. Additionally, overlay service meshes that depend upon Kubernetes, while gaining some beneficial security features such as data-plane certificate rotation from the underlying Kubernetes infrastructure, fail to provide complete security artifact rotation [11, 59]. In this way, manual, system administrator intervention is required to rotate and refresh the certificates of this system

overtime, resulting in system downtime. To combat this issue and prevent the need for system downtime, Anvil accommodates complete security artifact rotation as part of its design and provides mechanisms for completing this task in an automated, synchronized manner across all service mesh components. Distributed generation and distribution of security artifacts occur within the Anvil quorum nodes and are then distributed to other components through synchronized file downloads. Lastly, all service mesh components are notified to perform the necessary configuration changes to rotate to the new set of artifacts and this changeover occurs using a lively certificate rotation mechanism as part of the Anvil service proxy.

Attack Window Limitation: Due to the transformative, adaptive nature of security artifact rotations in Anvil, it is important to consider the security of the system over a period of time. Within Anvil, security artifact rotations of all encryption keys, certificate-key pairs, and access control tokens, are synchronized across the cluster at regular intervals and ensure that artifacts deployed within the environment are never *stale*, or in other words, exposed for extended periods of time. This characteristic of limited exposure provides a constrained window of time for an attacker to attempt to discover and make use of any of the security-sensitive artifacts within the cluster. As defined in [6], we see an *attack window* as a continuous time interval an attacker may leverage without being interrupted by system changes. In the context of Anvil:

> **Attack Window** – the period of time in which a given set of security artifacts are active within an Anvil deployment.

Below we formalize this concept within our context.

$$W = t + T_r \tag{3.1}$$

$$T_r = T_{gen} + T_{dist} + T_{ch} \tag{3.2}$$

In Equation 3.1, W represents the attack window, or time in which an adversary has the potential to compromise or discover actively used secret keys or tokens within an Anvil cluster. The

variable t represents the time between rotations that is configured by the system administrator of an Anvil cluster. T_r represents the "time to rotate" that is required for the various processes in Anvil to transition the cluster from one set of security artifacts to another. T_r is further expanded in Equation 3.2 where its component times are elaborated.

The variables T_{gen}, T_{dist}, and T_{ch} represent the time required to generate a new set of artifacts, time required to distribute the new set of artifacts to members, and the time required to synchronize a changing of node configurations within the cluster, respectively. However, T_{gen} and T_{dist} may be performed as background processes that do not affect normal Anvil cluster operations. The only component of a rotation that *requires* a synchronized response from cluster members and may incur downtime within the cluster is T_{ch}, however, in our experimentation, we find that changeover time within Anvil is extremely fast and results in near-zero downtime rotations of artifacts. Specifically, a live reload of the node configuration is performed with the new security credentials and the Anvil process is never stopped. Currently, within the default Anvil configuration, security artifact rotation frequency (t) occurs every 60 seconds, resulting in over 1440 rotations in a 24-hour period. This configuration may be altered according to the application needs and the risk that a given system may accommodate, limited only by the time required to generate the set of artifacts within the quorum.

3.4 Anvil Design

By reviewing the current state-of-art in service meshes, Anvil was designed to embody the core service mesh functionality while also introducing novel security features to the domain. Anvil's secure-by-design approach iterates on current service mesh design by alleviating many of the shortcomings and challenges found in present-day tools. Anvil provides beneficial security improvements and features in a feasible manner to be included in modern service mesh design.

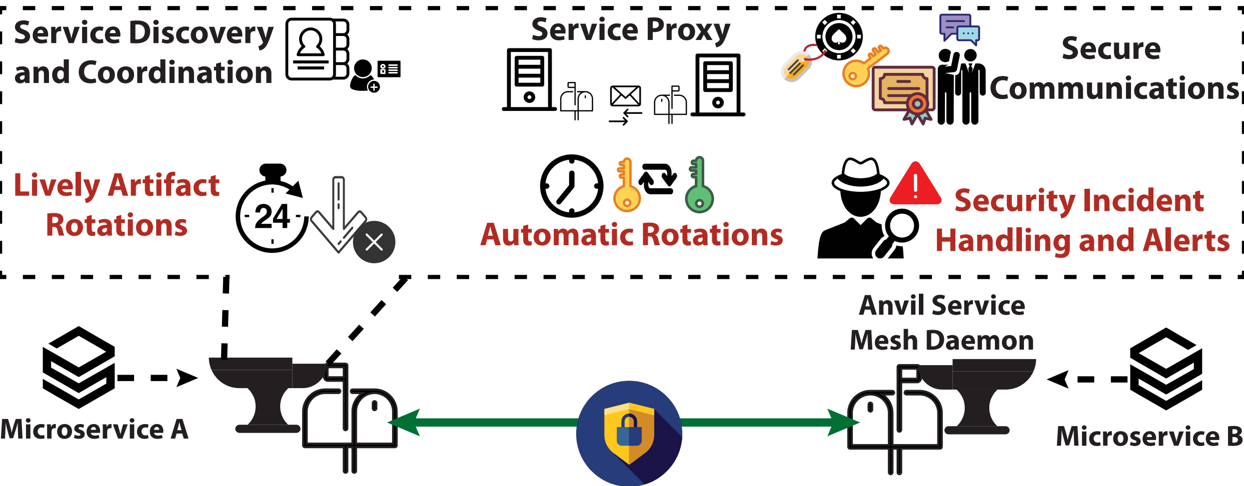

Figure 3.3: **Anvil Service Mesh Features** – Along with the baseline service mesh features set, Anvil introduces lively artifact rotations to avoid system downtime, automated security artifact rotations to ease administrator burden, and security incident detection and alerting, which are novel, beneficial features in the service mesh domain.

3.4.1 Core Service Mesh Functionality

While the designs of modern service meshes vary widely, the overall goals and core functionality remain very similar. Different implementations and additional features differentiate competing technologies from one another, however, the following set of functions are common among all production-ready service meshes and are also part of Anvil:

Service Discovery and Coordination: Discovery and coordination of deployed microservices is a key function in allowing service meshes to simplify the networking logic that must be included within developed microservices. By outsourcing the networking logic to the service mesh and performing service discovery, registration, and management within the service mesh layer, Anvil enables application developers to simply reference the dependencies for a given service as simple network connections and allow the service proxy included with the service mesh to complete the desired network connections on behalf of the microservice.

Service Proxy: All service meshes aim to provide a layer of networking logic between microservices that are deployed as a part of a larger cluster. Service meshes provide this proxy functionality by abstracting the network connections between microservices into a "data plane". Through this

abstraction, microservices do not require exact IP addresses or endpoints to contact, instead, utilizing a simplified network request format. Anvil implements service proxying through a series of firewall rules and routing logic. Microservices deployed within an Anvil cluster are first located via the service discovery and coordination mechanisms described above and then the Anvil proxy re-routes traffic according to the destination service, relaying traffic agnostically on behalf of microservices. Additionally, incoming requests at the service proxy are routed to the correct microservice using network metadata included in a request by the sender-side service proxy.

Secure Communications: Due to the presence of service meshes at network boundaries of deployed microservices, service meshes have significant responsibility to provide appropriate traffic routing, communication security, and speed. Additionally, to simplify microservice development and implementation further, network connections generated by microservices often do not have security associated with them. As such, this responsibility falls to the service mesh to implement and facilitate between microservices. Within Anvil, all microservices, whether creating secure connections as part of implementation, or not, are secured by Anvil between proxy connections. As a connection is created from a microservice hosted by Anvil, the service proxy accesses the request before it leaves the network boundary, TLS is applied along with any relevant access control tokens, and finally the connection is forwarded to the destination service proxy. Upon arrival at the destination service proxy, Anvil processes the TLS connection, extracts the service request from the payload of the connection and forwards the request to the appropriate microservice hosted at the destination node.

3.4.2 Novel Security Features

Automatic Artifact Rotation: Anvil makes use of a dedicated microservice that is deployed along with each of the quorum nodes to facilitate the automated generation, distribution, and synchronized rotation of security artifacts within a deployment. This microservice operates collaboratively on all quorum nodes to distribute the performance and distribution costs of generating a fresh set of security artifacts for all nodes. Initiated by the quorum leader, the rotation microservice ingests a

manifest file of all active nodes, services, and access control policies within an Anvil deployment. Next, each quorum node is assigned a portion of the manifest file to generate and distribute to other quorum members. All quorum members then synchronize and store the full set of generated security artifacts for all components before notifying the remainder of the microservice deployment of fresh artifact availability. All service mesh components then request and collect their necessary security artifact set from the quorum and wait to be notified of the rotation synchronization. Upon receipt of the synchronization, all components perform a lively rotation from one artifact set to another, effectively transitioning the state of the service mesh from one set of security artifacts to another with near zero downtime. In experimentation, the interruption caused by rotation was not measurable and operation of the Anvil deployment continued without failure.

Lively Artifact Reloads: A primary drawback and concern with modern service meshes are design decisions that conflict with the central theme of high uptime in microservice deployments. As mentioned previously, in order to refresh the artifacts present within Consul service mesh deployments, an entire shutdown and reconstruction of the service mesh is required. Additionally, in overlay service mesh instances, the Kubernetes control-plane certificates will eventually expire, requiring manual system administrator intervention to refresh and rotate these certificates. During this time, the Kubernetes deployment will be inoperable due to a lack of valid certificates present within the control-plane. In contrast, Anvil provides generation, distribution, and synchronization mechanisms by default within its deployments to provide seamless rotations from one security artifact set to the next. Anvil makes use of a lively TLS certificate rotation mechanism by "watching" the active certificate for file changes. Upon registering a change in the file, it begins to timeout and complete active network connections and then reinitialize the server with the new configuration in an automated manner. Through experimentation, we find that the interruption to network connections across the entire cluster are negligible in part because these certificate rotations are synchronized across all nodes in the cluster at the same time.

Anvil Module	Lines of Code	Rel. Proportion
Service Proxy	847	21%
Service Catalog	259	7%
Leadership Quorum	1343	34%
Security Mechanisms	290	7%
Artifact Rotation	688	17%
User-Interface	329	8%
Other	204	5%
Total	**3960**	

Table 3.2: **Anvil Prototype Lines of Code** – Written in Golang, this table outlines the major modules and components involved in the Anvil service mesh codebase as of the writing of this work.

3.5 Implementation

A range of tools and platforms have been used as a part of this work in order to express the benefits and features present within Anvil relative to alternative service mesh platforms. Local development, experimentation environments, and major cloud providers, such as AWS, have been used to craft the experimental evaluation of our system. Our implementation and evaluation of Anvil is intended to show the feasibility and beneficial nature of the novel design choices and security features of Anvil relative to the current state-of-art.

3.5.1 Anvil Development

Anvil is a proof-of-concept, service mesh implementation for the purposes of network security research and development of novel design choices in the service mesh domain. Anvil is written in a modular format in the Golang [41] programming language, a common choice for modern DevOps tools [10, 30, 47, 60, 69, 105]. Table 3.2 shows the high-level modules of Anvil and the respective lines of code necessary to implement this functionality. While Anvil borrows heavily from the current designs and implementations of current state-of-art service meshes, the codebase of Anvil is unique and does not include any direct implementation code from alternative service meshes. Many of the design choices, such as a "service catalog" similar to the Consul service mesh and the choice

of implementing the Raft [94] consensus protocol are found in many implementations. The goal of Anvil is not to re-invent the service mesh, but rather to demonstrate feasible and beneficial security designs in an easily modified platform. After examining the repositories and implementations of alternative service meshes, the strict design and highly-integrated implementation decisions made in these service meshes would not easily allow for the development and inclusion of these designs. Anvil is unique in its ability to accommodate the security benefits and features described previously in Section 3.3 that provide a more dynamic service mesh environment. Further, we believe that the security designs made in Anvil are applicable and have the potential to greatly improve the overall design of state-of-art service meshes with the proper engineering effort and time contributed to their integration in the current state-of-art. Due to this, we evaluate these beneficial design decisions within Anvil relative to the increase in performance cost over a baseline operational cost of Anvil and compare the Anvil runtime performance and downtime cost to the current alternative tools within the space.

3.5.2 Experimental Evaluation Setup

For our experimental evaluation, we made use of two controlled, experimental platforms. Our performance-level experiment workloads were deployed to a Dell R540 server configured with 128 GB of RAM, Xeon Gold 5117 processor, and 10 TB of SSD storage. This hardware configuration is comparable to what would be utilized in production environments, both in on-site and remote, cloud datacenters [26]. These resources were utilized for all experiments associated with the development of Anvil and the subsequent evaluation of Anvil and similar state-of-art service meshes. We additionally make use of AWS's Elastic Compute Cloud (EC2) [117] infrastructure to deploy and measure the downtime associated with the Consul service mesh for security artifact rotations.

Anvil Service Mesh Setup: For our experimental service mesh deployments, 15 virtual machines were deployed within the testbed environment. 5 were assigned the role of "quorum members" and were given 4 CPUs and 8 GBs of RAM each. As described in Section 3.2, quorum mem-

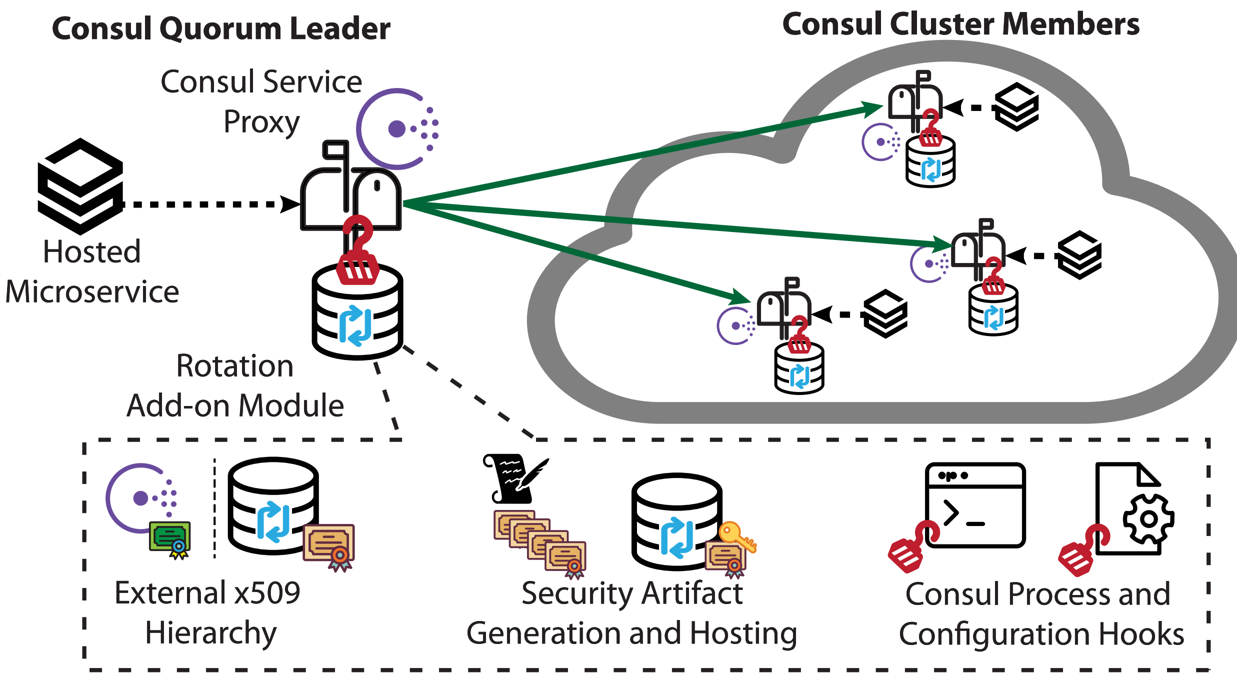

Figure 3.4: **ServiceFRESH Design and Implementation** – Deployed within AWS and using a separate x509 certificate hierarchy from the Consul TLS structure, ServiceFRESH provides generation and distribution of security artifacts without Consul code modifications. Despite this, performance and downtime costs due to Consul design result in the ServiceFRESH framework being unacceptable for use.

bers in Anvil and Consul service mesh deployments represent a higher-privileged subset of cluster members that are responsible for various security and consensus tasks, such as generating TLS certificate-key pairs for cluster members and keeping a consistent system log. The remaining 10 VMs were assigned as "client members" and were given 4 CPUs and 2 GBs of RAM each. In standard microservice deployments, client nodes would be responsible for hosting the various microservices that constitute the business logic of a system. By using similar hardware configurations to those found in a large-scale benchmark test by HashiCorp [86], these configurations are sufficient for our experimental evaluation to achieve a proper baseline of functionality between each type of node in the cluster as well as to gauge system performance overall in a small-scale setting.

ServiceFRESH – Platform Independent Service Mesh Setup: To demonstrate the effects of our "best-effort" prototype, ServiceFRESH, to provide security artifact rotations within Consul, we leverage Amazon Web Services (AWS) as a platform for deployment. We deploy various sizes of Consul clusters within AWS' general-purpose computation nodes, EC2. Our experiments ranged in EC2 hardware configurations from t2.2xlarge to t2.medium [117] providing 8 vCPUs with 32 GB of RAM and 2 vCPUs and 4 GB of RAM, respectively. ServiceFRESH augments Consul service mesh deployments by executing a series of Bash and Python scripts that coordinate the creation, distribution, and configuration of the security artifacts utilized within Consul. Figure 3.4 provides an abstract view of the ServiceFRESH prototype. Our rotation augmentation to Consul does not require any source code changes and operates separately from Consul. With this approach, we highlight the true cost of implementing security artifact rotations in Consul utilizing existing Consul design.

Istio – Overlay Service Mesh Setup: To create an accurate comparison of the performance of Anvil with a representative overlay service mesh, we design and implement an example workload within our testbed environment that attempts to mimic the traffic behavior of both Anvil and Consul quorums as accurately as possible. This example workload generates traffic patterns with the same frequency of the traffic generated within Anvil and Consul quorum members. We then deploy this workload via Docker containers within a Kubernetes environment augmented with the Istio

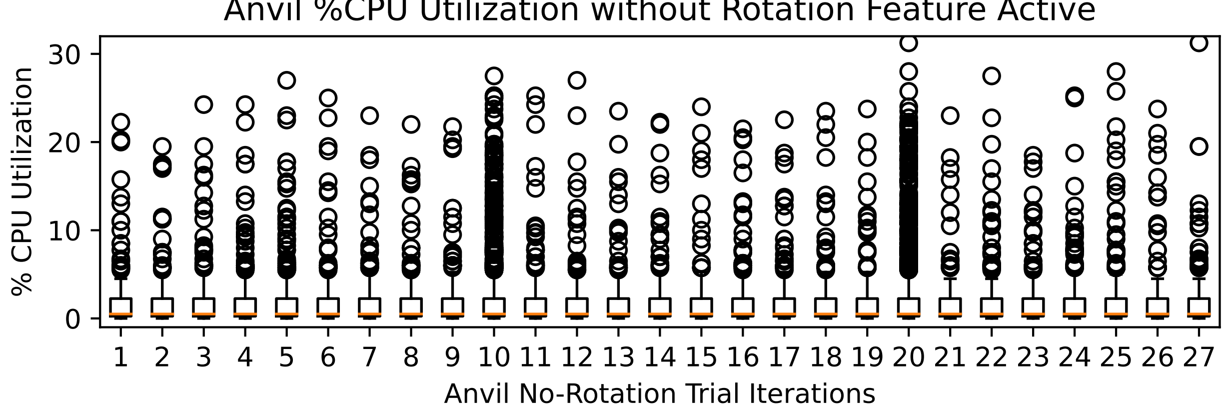

Figure 3.5: **CPU Utilization in Anvil without Rotations** – Box plots show CPU utilization across all experiment trials of Anvil when security artifact rotations were *not* active. Trials shown are those where full data collection was achieved. Median values are heavily focused near 0.5% utilization while outliers extend upwards of 30% utilization, but are highly infrequent.

overlay service mesh. Istio is currently the most widely used service mesh based upon GitHub metrics [49, 57, 79, 97]. This testbed environment involves 5 VMs with hardware configurations that match the "quorum members" deployed for the Anvil experimentation. In this way, we have replicated the quorum environment of Anvil and Consul as closely as possible to properly measure the overhead imposed by Kubernetes and a representative overlay service mesh. Specifically, we aim to install and configure an extremely lightweight overlay service mesh deployment to create as fair a comparison as possible to the Anvil service mesh. Because of this, we choose to deploy only a single pod (container) upon each of the Kubernetes nodes that make up our 5 VM cluster.

3.6 Evaluation

Anvil is intended to fill a gap in the current state-of-art in service meshes by demonstrating beneficial security design without significant costs to system overhead or deployment downtime. Anvil combines the flexibility and performance characteristics of platform-independent service meshes with increased security features of overlay service meshes. Additionally, Anvil provides novel security features and designs not found within *any* state-of-art service meshes. Due to these traits,

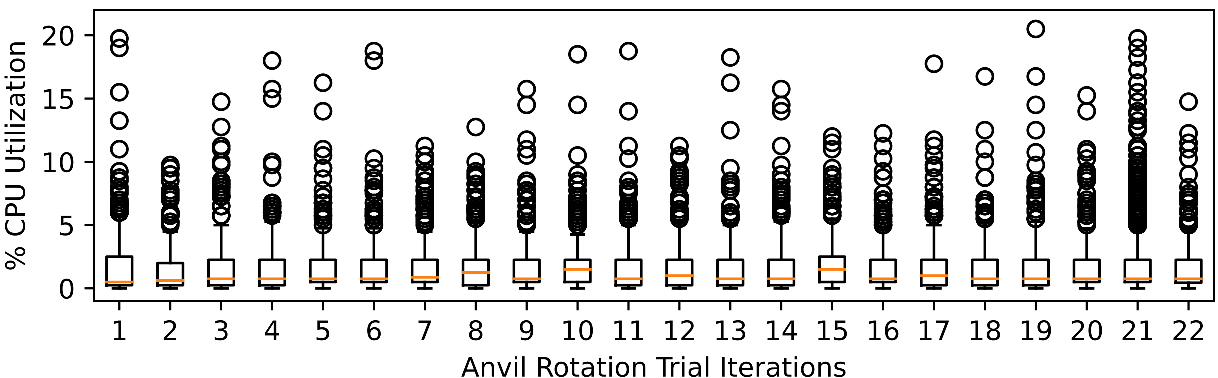

Figure 3.6: **CPU Utilization in Anvil with Rotations** – Box plots show CPU utilization across all experiment trials of Anvil when security artifact rotations were active. Trials shown are those where full data collection was achieved. Median values are heavily focused near 0.7% utilization while outliers extend upwards of 20% utilization, but are highly infrequent.

it is challenging to make a direct lateral comparison of Anvil to alternative options. To properly evaluate and compare the Anvil prototype with its state-of-art counterparts in service mesh tools, we combine three aspects of evaluation to place Anvil within the correct context of comparison for each point of consideration:

- First, we examine the Anvil proof-of-concept and assess the performance cost associated with Anvil's rotation capabilities relative to baseline Anvil without rotations.
- Next, we examine the cost of implementing a "best-effort" security artifact rotation system (ServiceFRESH) alongside Consul due to its lack of rotation capabilities. We evaluate Consul with ServiceFRESH according to the system downtime incurred.
- Finally, we compare the associated CPU and RAM utilization costs of overlay service meshes. We use the Istio service mesh as a representative example of this design to assess performance and operational costs relative to Anvil.

Anvil Experimental Performance Cost: With artifact rotation being a novel contribution within the Anvil service mesh, it is necessary to consider the performance cost of facilitating this feature

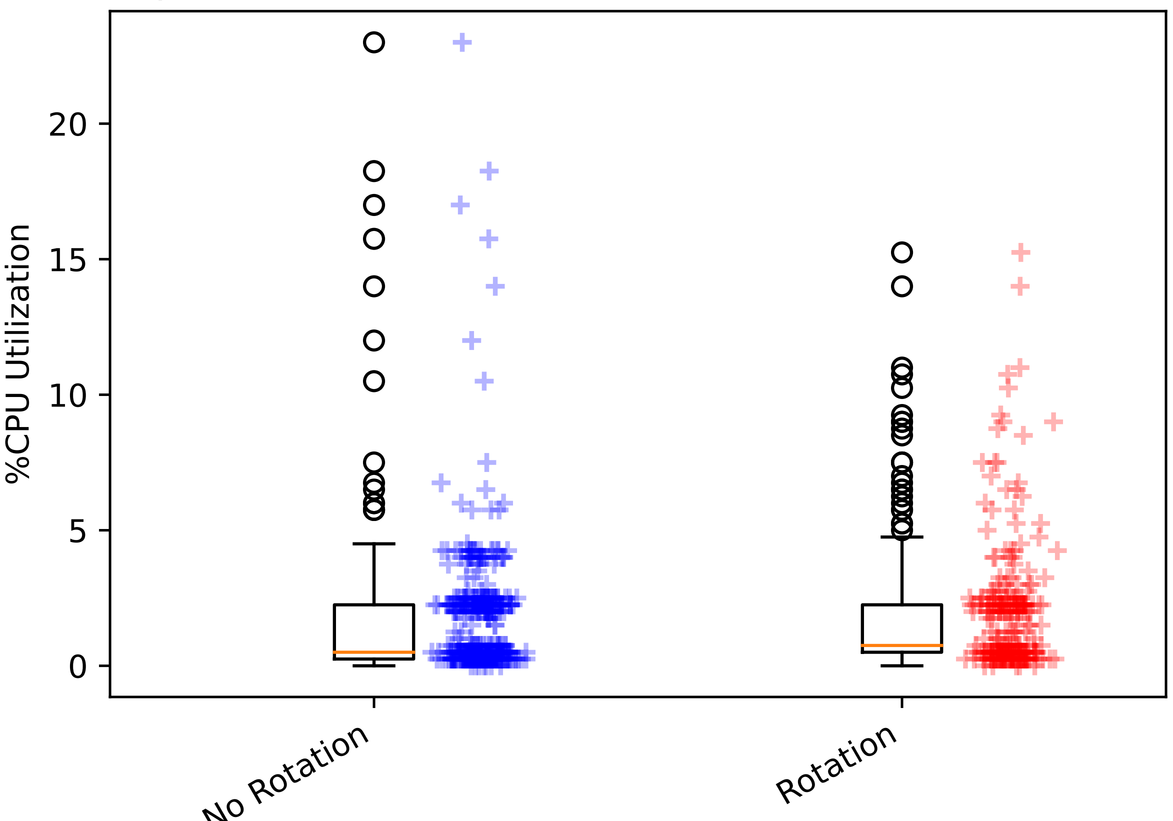

Figure 3.7: **CPU Utilization in Anvil (Single Trial) –** Box plots of CPU utilization in showing one experiment trial of Anvil with and without security artifact rotations active (data point distribution plotted alongside).

within an active Anvil service mesh cluster. As such, we leverage the experimental setup previously described in Section 3.5 to conduct our experiments and measurements. Additionally, we monitor active CPU and RAM utilization of nodes within the cluster by employing the *pidstat* [39] Linux utility. We track each of the system processes that Anvil creates separately and measure their CPU and RAM utilization as a percentage of the system total. Next, the separate processes are combined to form an instantaneous measurement of the "cost" of Anvil at any given moment during the experimentation. Using 29,400 points of utilization data, we plot the CPU and RAM utilization data into box plots to view the spread and overall locality of the data. CPU utilization data across all experiments is presented in its entirety in Figures 3.5 and 3.6 which has been separated into each of the data collection trials performed. As can be seen, the data present within these figures is highly consistent, showing that our collected data is indicative of consistent behavior of Anvil within our experimental deployments. To provide a more concise view of how Anvil performs when rotations are active versus inactive, we plot a single trial of each experimentation within Figure 3.7 and plot the raw data points alongside each box plot to demonstrate the distribution of the data in a more concise fashion. Figure 3.8 plots the RAM utilization observed through all Anvil experimentation trials. Based upon median values across trials, Anvil with rotations active requires only $\sim$1% higher CPU utilization in the worst case trial and in the best case trial, Anvil with rotations active has the same median CPU utilization as Anvil without rotations active. This is due to the cost of rotation events being amortized over the time period in which the new set of security artifacts is active within the cluster. The frequency of rotations is configurable within Anvil and may be modified by system administrators according to their organization's risk. With Anvil's current status, rotations as frequently as every 2 minutes are possible, however, through optimizations and more powerful hardware configurations, this value may be shortened. For high-risk scenarios, more frequent rotations, on the order of a few minutes, may be necessary, while in low-risk scenarios, less frequent rotations, on the order of hours or days, may be acceptable resulting in very low additional overhead imposed by rotations.

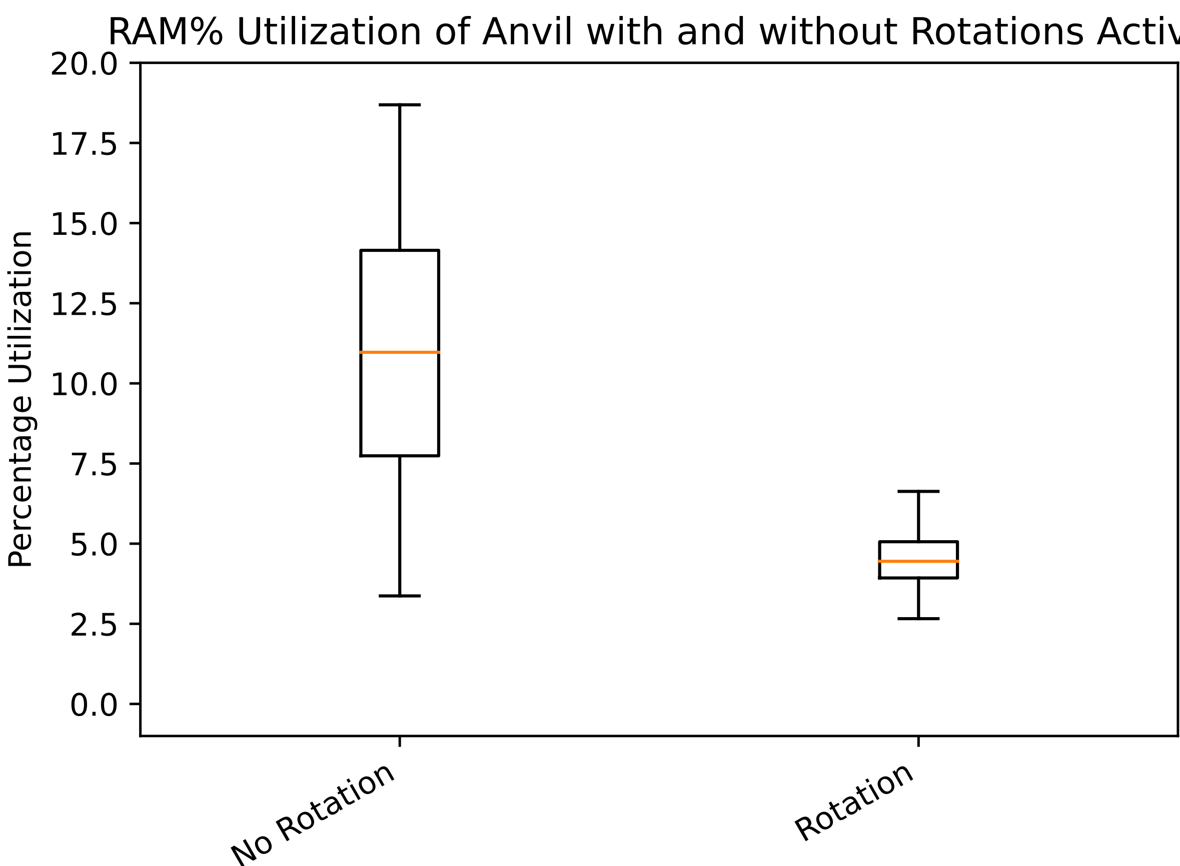

Figure 3.8: **RAM Utilization in Anvil** – Box plots of RAM utilization in Anvil with and without security artifact rotations active. After a careful examination of collected data that included use of Golang's profiling library *pprof* [40], we believe Golang's garbage collection responsible for this behavior and is triggered more often in the case of rotations.

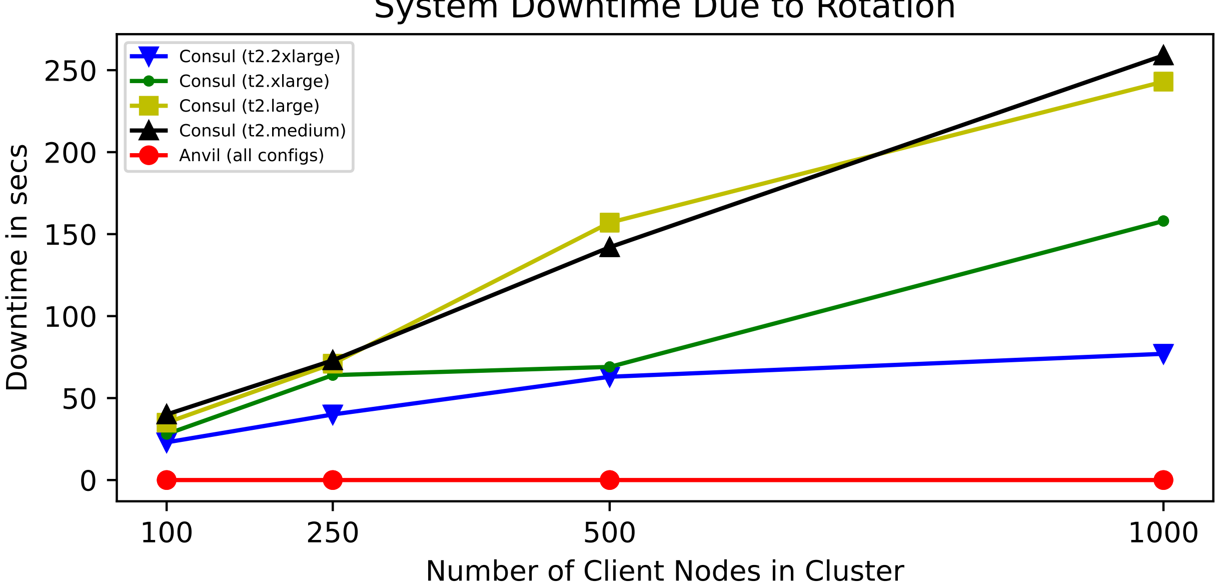

Figure 3.9: **Downtime of Best-Effort Rotation (ServiceFRESH)** – Each line represents a different hardware configuration within Amazon Web Services' EC2 cloud t2-series of virtual machines. The leader of Consul quorums is required to perform the generation and distribution tasks for cluster startup, this figure represents this cost in terms of seconds for the cluster to move from one artifact set to another. In contrast, Anvil requires *no downtime* to facilitate security artifact rotations. For the full dataset, refer to Table 3.3.

3.6.1 Consul (ServiceFRESH) Downtime Costs

While the Consul service mesh provides appealing performance and flexible deployment options that may be desirable to system administrators, it lacks the foundation for effective rotation of security artifacts as part of its functionality.

We frame the comparison of Anvil and Consul in terms of the differences in design choice between Anvil and Consul, primarily the aspect of security artifact rotations and the "downtime" cost of performing rotations. Downtime is a key metric for service meshes due to a desire for high-availability and speed related to request-response latency in microservice deployments. In this scenario, it can be imagined that a compromised node within a deployed Consul cluster has been detected and in order to maintain protection of the system overall, new security artifacts must be generated for all nodes and redistributed to all cluster participants.

Figure 3.9 displays the results of the best-effort system shown in Figure 3.4 that attempts to automate the shutdown, artifact creation, artifact deployment and subsequent cluster recreation of such a scenario. As the Figure 3.9 demonstrates, this cost grows as the system scales in size, resulting in significant cost to large enterprises that make use of Consul. With a *best-case* downtime of 23 seconds and a *worst-case* downtime of 259 seconds, the cost of system downtime in a Consul deployment can result in the unavailability of deployed microservices for significant periods of time.

The downtime cost increases in such a fashion within the Consul service mesh due to highly centralized design choices with respect to security. In other words, the access control mechanisms within a Consul service mesh must be created and distributed while the cluster is operational. This results in significant cluster downtime while nodes wait for security artifacts to be distributed and configured before service requests may be answered securely. Figure 3.12 highlights these costs at a finer granularity, showing specifically the processes involved in a Consul rotation that constitute the total system downtime from a rotation. As the figure shows, the primary sub-processes of ServiceFRESH that incur downtime are in the generation of TLS certificates and the generation and ingestion of access control policies and tokens. TLS certificates are required for Consul nodes to communicate securely with one another and access control policies and tokens are required for Consul nodes to take actions within the deployment. These actions include joining a deployment and becoming a member of the service mesh, a node wanting to update its status in the registries of other nodes to signal that it is online, or a deployed microservice attempting to make a service-level request to another microservice in the deployment. In contrast, Anvil is able to accomplish the task of full-system artifact rotation and refreshment without incurring any system downtime.

3.6.2 ServiceFRESH Downtime Results

An important aspect of a system that performs modification of configuration and rotation of security artifacts in live environments is to measure the effects that these operations have upon the availability and uptime of services. As part of this study, we aim to provide a comparison be-

AWS EC2 Node Type	Cluster Size	Exp. Trial 1	Exp. Trial 2	Exp. Trial 3	Exp. Trial 4	Exp. Trial 5
t2.medium	100	41 secs	33 secs	**40** secs	35 secs	42 secs
	250	73 secs	73 secs	**73** secs	72 secs	72 secs
	500	158 secs	177 secs	125 secs	**142** secs	140 secs
	1000	**259** secs	243 secs	244 secs	264 secs	261 secs
t2.large	100	30 secs	35 secs	42 secs	41 secs	**35** secs
	250	72 secs	71 secs	**71** secs	69 secs	71 secs
	500	122 secs	**121** secs	119 secs	118 secs	133 secs
	1000	261 secs	249 secs	241 secs	**243** secs	223 secs
t2.xlarge	100	24 secs	26 secs	**28** secs	29 secs	31 secs
	250	64 secs	64 secs	**64** secs	64 secs	63 secs
	500	78 secs	68 secs	88 secs	**69** secs	68 secs
	1000	157 secs	157 secs	168 secs	**158** secs	162 secs
t2.2xlarge	100	19 secs	21 secs	**23** secs	25 secs	26 secs
	250	40 secs	42 secs	36 secs	39 secs	**40** secs
	500	65 secs	62 secs	**63** secs	63 secs	62 secs
	1000	77 secs	**77** secs	86 secs	88 secs	77 secs

Table 3.3: **Full Dataset for ServiceFRESH Downtime Analysis –** to measure the downtime cost incurred by the ServiceFRESH, five independent experimentation trials were conducted. Each hardware configuration and cluster size combination were trialed independently of one another across a series of five observations. The median values across the five runs are highlighted and correlate to the values presented in Section 3.6: Figure 3.9.

tween Anvil that provides automated security artifact rotations and Consul that does not provide any support for artifact rotations.

In order to facilitate this functionality within Consul, we design and implement ServiceFRESH, a "best-effort" rotation add-on to Consul. To measure the downtime cost of operating Consul in conjunction with ServiceFRESH, we make use of an internal API within the Consul service mesh to measure the system downtime incurred during a rotation. This internal API, the "consul members" command, is designed to collect and display the current, visible members of a Consul service mesh.

An important note to make about this API is that it requires a notable amount of processing time on the node performing the call. Due to this, our measurements involve periodic calls to the Consul API every 10 seconds. When the API call is made, if the number of "members" observed on the Consul leader is not the same as the number of nodes that were deployed in a given experiment trial, then downtime is recorded. The API calls are repeated every 10 seconds, until the response of the API call matches the expected number of members for a given experimentation trial. When this value is met, the system is then recorded as being "available" and the accumulation of downtime is stopped. Table 3.3 illustrates the results of five independent experimentation trials under varying

EC2 Type	Cluster Size	UDP Key Generation	ACL Wait	ACL Token Generation	TLS Certificates	Artifact Bundling
t2.medium	100	0,0,0,0,0	15,15,15,15,15	12,11,12,12,12	8,7,7,7,7	0,0,0,0,0
	250	0,0,1,0,0	15,15,15,15,15	30,30,30,30,29	17,17,16,16,17	0,0,0,0,0
	500	0,0,0,1,0	15,15,15,15,15	53,52,52,52,53	35,33,33,31,34	0,0,0,1,0
	1000	0,0,0,0,0	15,15,15,15,15	94,96,98,100,101	66,66,64,65,67	0,0,1,0,0
t2.large	100	0,0,0,0,0	15,15,15,15,15	12,12,12,12,11	7,7,7,6,8	0,0,0,0,0
	250	0,0,0,0,1	15,15,15,15,15	28,28,28,27,28	18,17,17,16,16	0,0,0,0,0
	500	0,0,0,0,0	15,15,15,15,15	48,48,48,48,48	34,33,34,32,34	0,0,0,0,0
	1000	0,0,0,0,0	15,15,15,15,15	92,92,91,93,92	65,65,65,66,64	0,1,0,0,1
t2.xlarge	100	0,0,0,0,0	15,15,15,15,15	6,7,7,7,7	4,4,3,3,4	0,0,0,0,0
	250	0,1,0,0,0	15,15,15,15,15	15,15,15,15,15	8,8,9,8,8	0,0,0,0,0
	500	0,0,0,0,0	15,15,15,15,15	26,26,25,25,26	16,16,17,18,16	0,0,0,0,0
	1000	0,0,0,0,0	15,15,15,15,15	56,55,55,55,55	32,33,34,32,34	0,0,0,0,0
t2.2xlarge	100	0,0,0,0,0	15,15,15,15,15	5,4,4,5,4	2,2,2,2,2	0,0,0,0,0
	250	0,0,0,0,0	15,15,15,15,15	7,8,8,7,8	5,5,4,5,4	0,0,0,0,0
	500	0,0,0,0,0	15,15,15,15,15	18,14,14,13,14	9,8,8,9,8	0,0,0,0,0
	1000	0,0,0,0,0	15,15,15,15,15	28,27,27,27,27	16,16,16,17,16	0,0,0,0,0

Table 3.4: **Full Dataset of ServiceFRESH Microbenchmark Experimentation** – to measure the cost that individual processes within ServiceFRESH imposed upon the system downtime, each computational process was measured independently during rotation events. Each hardware configuration utilized and cluster size under test are provided as rows in the table. Columns of the table denote the computational process under test, while the values depicted in the table represent an individual measurement of the process under test. Each cell contains five unique, independent values, representing the values that were observed during experimentation. All values within the table are recorded in seconds. The results of this data collection are plotted within Section 3.6: Figure 3.12.

EC2 hardware configurations and differing amounts of cluster sizes.

3.6.3 ServiceFRESH Microbenchmark of Downtime Costs

To provide a more complete picture of the sources of downtime caused by ServiceFRESH, we perform a microbenchmark analysis of the individual processes that constitute this "best-effort" implementation. As part of this study, five computational processes present within ServiceFRESH were determined as potential sources for the downtime suffered. Table 3.4 displays the full dataset that was observed throughout our experimentation of ServiceFRESH under varying sizes of Consul clusters. These results were measured as a part of a series of independent experimentation trials. To perform the microbenchmark analysis of the various computational elements of ServiceFRESH, during script execution, the Unix *date* system utility was called immediately preceding the processes and immediately following. Due to the highly parallelized computations occurring

as a part of the rotation add-on, the Unix *wait* system utility was also utilized as a way of ensuring that *all* computation from one portion finished and was recorded before the next computational process was initialized.

As can be seen in Table 3.4, a constant source of downtime across all trials, hardware configurations, and cluster sizes was the need to delay initial operation of the Consul cluster until the ACL system transitioned from a "legacy" state to the "modern" ACL operational state. This 15 second cost to cluster operations varies in its proportional effect on the overall system downtime, however, this cost was necessary in order to ensure correct and complete rotation of security artifacts from one state to another.

3.6.4 Overlay Service Mesh Performance Costs

Mentioned previously in Section 3.2, the remaining state-of-art service meshes aside from Consul and Anvil require an underlying Kubernetes platform to provide the promised service mesh functionality. Due to this, a comparison of Anvil performance costs to overlay service meshes is unreasonable without also accounting for the cost of the underlying Kubernetes platform with the hosted service mesh.

In order to make the comparison effective, we leverage the experimental testbed described in Section 3.5 relating to our representative, overlay service mesh, Istio. Specifically, after creating the network traffic application that mimics Anvil and Consul quorum network traffic, we deploy the experimental workload and measure process overhead with respect to CPU and RAM utilization. Again, we leverage the *pidstat* Linux utility to collect these metrics and combine the individual process measurements together to get a "snapshot" of the total usage at a given moment in time. As noted in Section 3.5, the testbed system contained 5 VMs with the relevant software installed for each of the experiment types and then within each of the VMs, only *one* container or pod was deployed onto the VM. , representing an extremely streamlined and lightweight implementation of Kubernetes and an overlay service mesh.

Leveraging this deployment, we intended to mimic both the computing power and network

structure of the Anvil and Consul quorum members which hold higher privilege and responsibility in a service mesh deployment. In this way, we aim to create as close and as fair a comparison as possible to the Anvil environment for evaluating the platform-level cost of an overlay service mesh deployment. In each instance of experimentation, one pod (container) was selected as the "leader" of the makeshift quorum and the mimicry workload was executed. The processes associated with running this application and the underlying platform processes were measured for CPU and RAM utilization. It is important to note that we do *not* include the overhead of the network traffic workload in our performance evaluation of Kubernetes and the Istio overlay service mesh. The performance costs associated with our evaluation of a representative overlay service mesh are only the processes associated with Kubernetes and the Istio service mesh itself, in a near-idle state. After collection, the data points gathered were collated according to the experiment conducted and the type of node deployed. Figure 3.10 displays CPU utilization box plots of our experimental trials.

Our experimental trials for platform-dependent service meshes were: Kubernetes without TLS active in the network traffic workload, Kubernetes with TLS active in the workload, Istio without TLS active in the network traffic workload, and Istio with TLS active in the workload. The median CPU utilization percentages were 0.5%, 0.5%, 0.75%, and 1.0%, respectively. Next, we considered RAM utilization as another important factor to evaluate the cost of overlay service meshes.

Figure 3.11 shows the results of the collection and analysis of RAM utilization in our experiments. Following the experimental trials noted previously, the median RAM utilization values for each trial were 2.08%, 2.12%, 4.65%, and 5.28%. When comparing the cost of each overlay service mesh experiment trial to the cost of Anvil with security artifact rotations, the CPU utilization and RAM utilization values are very comparable. However, while the system costs are comparable between Anvil and overlay service meshes, the testbed environment utilized is a near best-case scenario for overlay service meshes and underlying platform, yet still requires manual system administrator intervention throughout the lifetime of the deployment as noted in Table 3.1. Considering this, the design-level benefits and security features that Anvil provides over *all* state-

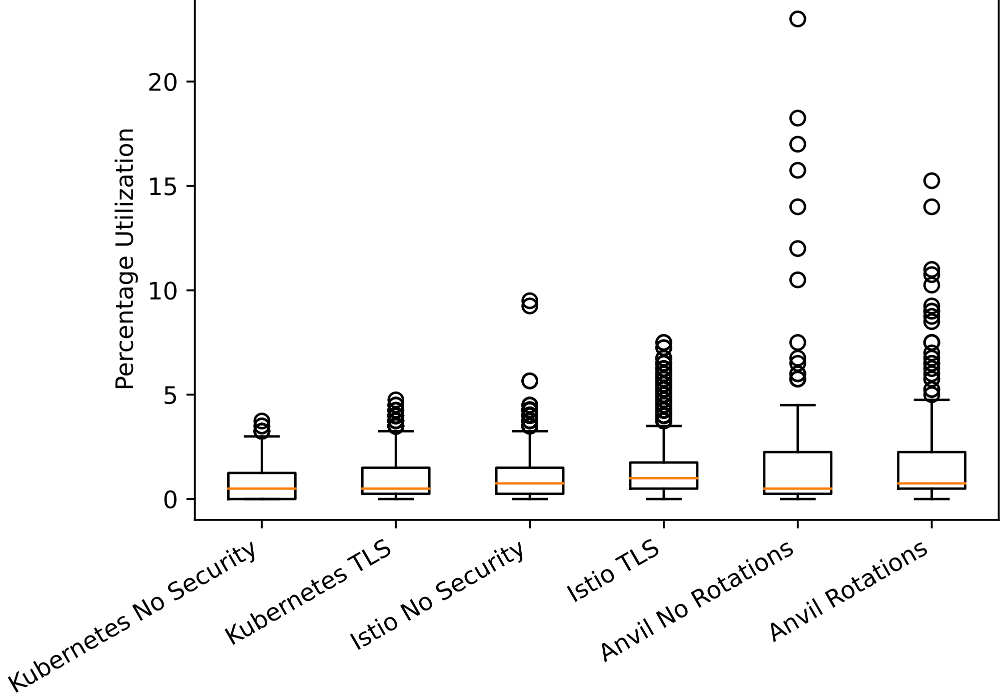

Figure 3.10: **CPU Utilization of Platforms During Network Mimicry** – Box plots illustrate the spread and locality of CPU utilization. Median values for each experiment are: Kubernetes without workload security features active (0.5%), Kubernetes with workload security features active (0.5%), Istio without workload security features active (0.75%), Istio with workload security features active (1.0%), Anvil without rotations active (0.5%), and Anvil with rotations active (0.75%).

of-art service meshes in conjunction with the comparable performance costs of Anvil to overlay service meshes is significant.

3.7 Discussion and Limitations

This work presents and demonstrates the design benefits of novel security features and automated security artifact rotations within the service mesh domain. Anvil, a proof-of-concept system, has been used to implement and demonstrate the feasibility of these design decisions relative to a "best-effort" implementation (ServiceFRESH) alongside a current, state-of-art service mesh. As an initial step in improving the security of service meshes, there are a range of future work opportunities and considerations associated with this work.

State-of-Art Integration: A direct, head-to-head, performance comparison of Anvil to the current state-of-art in service meshes is not indicative of the security design benefits that Anvil provides, or of their feasibility in modern service meshes. Rather, we frame our evaluation as a means to show relative costs and benefits associated with these decisions. Currently, production-ready service meshes have design decisions that prevent the direct integration or implementation of these security design decisions. Despite this, the security benefits and feasibility demonstrated by Anvil show that the engineering effort required to integrate these changes in modern service meshes can bring improvements to the domain.

Consensus Algorithms: Throughout development and testing, it was discovered that the overall structure of the Raft [94] consensus algorithm was causing high-levels of network traffic within Anvil. Specifically, when TLS was enabled within Anvil, the CPU and RAM utilization of the system rose dramatically. Upon investigation, the cause was discovered to be the implementation of Raft consensus initiating high numbers of TLS connections for secure messaging between quorum members. Raft was adopted in Anvil to enable a Consul-like quorum structure, providing scalability and flexibility.

However, after this performance overhead was identified, the Consul codebase was further examined and the usage of Raft within Consul was found to use long-lived TLS connections for an extended period of "heartbeat" messages within Consul. In this way, Consul avoids much of the overhead of renegotiating TLS connections at a high-rate and instead amortizes this cost over the lifetime of the connection between two quorum nodes. However, the implementation of Raft in this way conflicts with the flexibility, speed, and dynamic nature of service meshes overall. From a security perspective, rather than utilizing long-lived TLS connections, service meshes should renegotiate TLS connections for every message in case a certificate must be revoked or a node removed from the cluster, the certificate is no longer exposed.

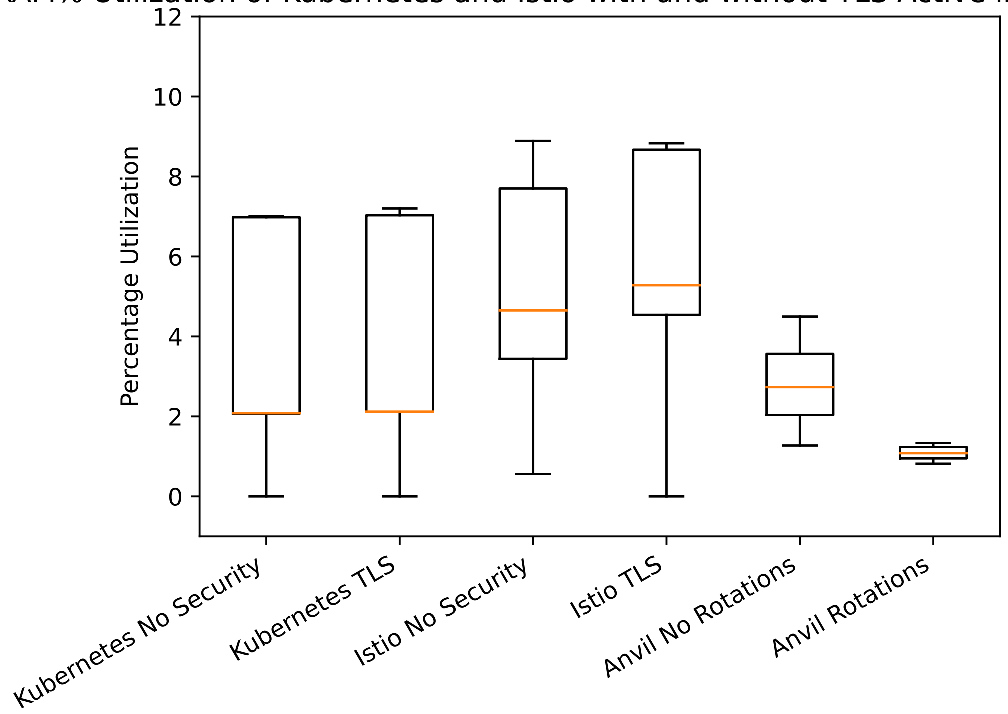

Figure 3.11: **RAM Utilization of Platforms During Network Mimicry** – Box plots illustrate the spread and locality of RAM utilization. Median values for each experiment are: Kubernetes without workload security features active (2.08%), Kubernetes with workload security features active (2.12%), Istio without workload security features active (4.65%), Istio with workload security features active (5.28%), Anvil without rotations active (2.73%), and Anvil with rotations active (1.08%).

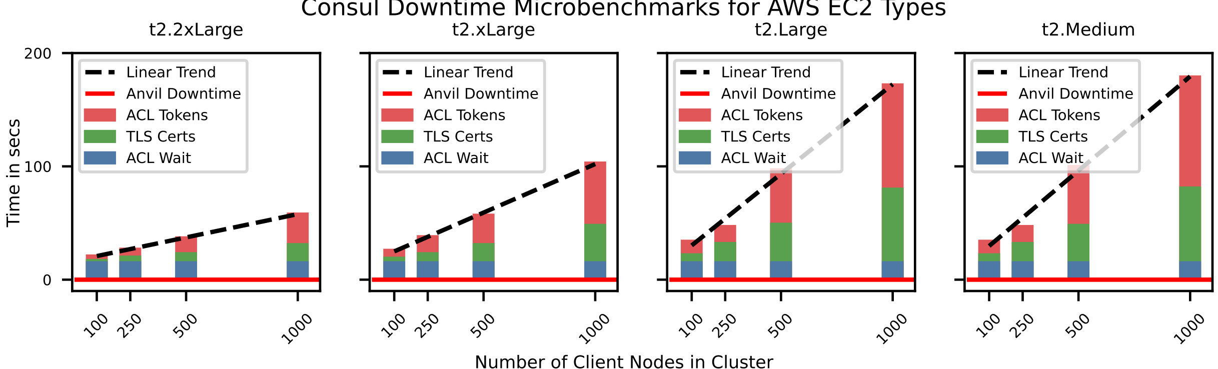

Figure 3.12: **Microbenchmark Analysis of ServiceFRESH Downtime Sources** – Each figure represents a different hardware configuration within Amazon Web Services' EC2 cloud t2-series of virtual machines. The total downtime associated with performing a rotation within Consul is separated into the components in which the leader of the quorum spends time. In contrast, Anvil requires *no downtime* to facilitate security artifact rotations and distributes the generation load among *all* quorum members rather than solely the leader. For the full dataset, refer to Appendix 3.6.3.

3.8 Related Work

Service meshes as an element of the DevOps toolset are a relatively recent development. Due to this, service meshes specifically have not been the target of a significant amount of research. However, microservices hosted within service meshes and the enabling container and platform technologies for microservices have been well-studied in literature.

Microservice Security: The microservice security structure has been studied closely and been shown to be vulnerable to exploits in a number of previous works. Rastogi, *et al*. [108] evaluate an automation system for decomposing a monolithic software deployment into a collection of collaborating microservices, the purpose of which, to better adhere to the principle of least privilege [115]. As a part of this effort, Rastogi, *et al*. developed a special-purpose system that uses a static binding to communicate between microservices, rather than an examination of the available security mechanisms within service meshes tools that dynamically connect services. A lack of security protections in the Docker container environment is noted by Yarygina, *et al*. [137] and they propose a security monitoring system for containers as a potential solution to this issue. Such a system could

be leveraged as part of the long-term security of a microservice deployment that utilizes the Anvil service mesh. Trust within deployed microservices is often inherent, as noted by Sun, *et al*. [124]. The authors study how the trust relationship between deployed microservices may result in the compromise of an entire system. Further, they propose a system for deploying network security monitors in microservice environments to detect and block threats to clusters. Anvil extends this work by providing zero-trust, communication security as a feature of the service mesh. In this way, exploits and attacks against containers are prevented from corrupting other portions of the service mesh cluster. Similarly, work by Li, *et al*. [76] studied how inherent trust among microservices can be exploited by an insider threat. Assuming the threat model of a compromised microservice that attempts to make unauthorized requests to other microservices, the authors design and implement a solution for scanning microservice source code and extracting relevant metadata relating to network requests. With this extraction, they generate a minimal set of the necessary Access Control List (ACL) entries necessary for the service mesh to operate correctly. This design could be leveraged within Anvil to automatically generate a minimal set of network relationships and the necessary access control tokens for the network connections between microservices deployed in an Anvil service mesh.

Analysis of Consensus Protocols: Within this study, we utilize the Raft consensus protocol for log consistency within the Anvil quorum in a very similar fashion to Consul. Work by Sakic, *et al*. [113], examines the availability and response time of nodes participating in Raft. With Raft used as the point of coordination for rotations within Anvil, availability and responsiveness of quorum members is extremely important.

Aside from the Raft consensus protocol, interest in blockchain technologies has fueled the development and proposal of security solutions that leverage blockchain for running consensus protocols in microservice clusters. Hyperledger Fabric [43, 122] is one such development that aims to offer consensus and membership support at scale. These studies, examine the security threats to consensus protocols, such as sybil attacks on collaborative network services. However, blockchain technology can defeat traditional sybil attacks via proof-of-work or related protocol-level mech-

anisms. In contrast, Anvil uses consensus protocols to coordinate service mesh leadership and facilitate synchronized rotations.

3.9 Summary

As microservice architectures have grown in popularity and the number of components to manage have exploded, service meshes have assumed the role of managing and coordinating actions within these deployments. However, the current state-of-art in service mesh technology fails to meet the security and long-term maintenance needs of the microservice domain. Through an analysis of the current state-of-art in service meshes and a relevant threat model to the domain, we identify key shortcomings and gaps in existing research within the area of microservices. Our proposed solution, Anvil, and our best-effort framework ServiceFRESH explores augmentations to service mesh design by providing a more domain-conscious approach to security and maintenance in service meshes. We demonstrate the feasibility of Anvil's design and features through an experimental evaluation, comparing Anvil's benefits with a best effort rotation system, ServiceFRESH. We recommend that Anvil's lively, automated rotation of security artifacts and synchronized artifact loading scheme be included in the development and design of current state-of-art service meshes to provide a more comprehensive security approach to users of these tools.

Chapter 4

Mitigating Microservice Compromise in Service Meshes

4.1 Introduction

Growing demands of software systems and enterprise software have led to developments and innovations in how software is deployed. What were once large, single-server applications now span thousands of containers and virtual machines hosted within massive datacenters globally [63, 110]. The transition of software from the single-server, *monolithic* structure to the modern, *microservice* approach has separated and isolated the logic components of large software systems. However, this has lead to a dramatic increase in unique components and subsequent dependencies between components that must be managed. This phenomenon has been referred to as *microservice explosion* [64, 125].

The microservice explosion issue has rendered previous management and deployment techniques, such as manual deployments and administrator interaction infeasible to use in enterprise software [1]. However, the benefits of scalability, faster deployments, ease of development, and separation of concerns push real-world environments to adopt microservice architectures [63]. Despite benefits of containerization and virtualization, full redeployments of microservice environments are impractical, often requiring full reconstruction and configuration of management platforms. Rather, the hosted containers and microservices are intended to be temporal, often capable of achieving continuous deployment strategies, but the underlying orchestration and management platforms are intended to be long-lived [63, 110]. This results in a vital need for strong security capabilities within the management elements of microservice architectures. With a growing need to maintain and deploy software at increasing speeds, innovations in tooling and management for

the microservices domain are necessary [28, 80, 134].

The modern approach to addressing these new challenges in software deployment are *service meshes*, a subset of tools within the wider collection of methods and practices referred to as *DevOps*. Service meshes, such as Consul [47] and Istio [60], have filled the role of controlling, securing, and connecting collections of single-purpose microservices. Microservices, the building blocks of large software systems, collaborate with one another to achieve high-level business goals and fulfill user needs [63]. Microservices are often deployed within containers, but may also be virtual machines or even dedicated, physical hosts. This model of system design is broadly referred to as the microservice architecture and has seen dramatic growth in popularity in recent years with a range of enterprise organizations transitioning their infrastructure [16, 31–34, 67, 129]. Additionally, trends of widespread adoption and increasing maturity have been noted in surveys conducted by the Cloud Native Computing Foundation (CNCF) [18, 19].

With a broad list of responsibilities and capabilities, service meshes bear the weight of providing infrastructure management as well as network coordination and security amongst deployed resources. For instance, due to the separation of different aspects of a microservices-based software deployment, what were once *intra*-service connections have become *inter*-service connections [110]. As such, connections between services must now traverse network boundaries, inevitably encountering uncontrolled, shared networks. Due to this, service meshes must also be responsible for coordinating and securing these connections to uphold the necessary security guarantees of the system as a whole [45, 120].

Despite being charged with responsibilities of network security, current service mesh designs fail to account for the possibility of security issues within microservices and containers. Among these are: software supply chain vulnerabilities, closed-source third-party code, and underlying container vulnerabilities [77, 92]. Additionally, real-world incidents and mishaps [23, 92, 112] show a need for the ability to detect misconfigurations or misbehavior within microservices and effectively alert on, or mitigate, these events. However, these vital capabilities are lacking in all modern service meshes. Due to service proxy designs present in *all* state-of-art service meshes,

there is no mechanism to detect, and effectively notify administrators, when access control policies are violated [9, 46, 55]. These drawbacks conflict with responsibilities delegated to service meshes and are worsened by the issue of microservice explosion. As microserivce systems grow to tens of thousands of services, as in [44, 83, 133], the strain on administrators is increased and environment redeployment becomes infeasible.

Our proposed framework, *ServiceWatch*, directly addresses these shortcomings with novel security and design choices. Leveraging automated rotation of all encryption keys, certificates, and access control credentials used in service meshes (herein referred to as *security artifacts*), ServiceWatch provides timely mitigations and responses to these threats while preserving the underlying management structure. In this way, ServiceWatch provides mechanisms for partial compromise and failure recovery, allowing the service mesh to continue operations without requiring full system redeployment, as could be the case in the current state-of-art. Additionally, security freshness guarantees claimed by alternative tools are automatically maintained in ServiceWatch through the lifetime of a deployment with no administrator intervention, easing operational burden.

ServiceWatch's inclusion of these security features in its core design addresses security issues that are not accounted for in the current state-of-art. Further, the inclusion of these security features was impractical to accommodate within these existing tools, requiring significant re-design and engineering efforts to accomplish. ServiceWatch brings beneficial security features to the microservice domain by addressing known attacks and issues, such as vulnerabilities present in Docker [27] container images [119], inherent trust between microservices [124], and unauthorized service requests from compromised microservices [76]. In recent years, a model of such a threat is the Log4j vulnerability [128] causing widespread effects and damage [36, 132]. We present ServiceWatch's beneficial design decisions and feasibility as a roadmap for future design incorporation, allowing tool designers and engineers to adapt their architectures for these novel capabilities.

Through an experimental implementation of ServiceWatch's design features, we analyze the network overhead of our proposed access control verification process, detailed further in Sec-

tion 4.4. Using experimental data from a real deployment of the Google "Online Boutique" microservice example system [103], we show the feasibility of ServiceWatch's design choices to handle large environments and how these capabilities scale with the size of the microservice deployment. With reasonable overhead and novel security design, ServiceWatch addresses the key issues of security management and maintaining strong security posture in microservice deployments unseen in the current state-of-art.

As part of this work, we provide the following contributions:

- We analyze state-of-art service meshes under a relevant threat model (based on real-world incidents) and identify security shortcomings and limitations.
- We present ServiceWatch, a service mesh platform with mitigation capabilities for misbehavior and misconfiguration and automated security artifact rotations to address against microservice threats and challenges.
- We present an experimental and theoretical cost analysis of ServiceWatch's security features showing its feasibility for use in modern service-mesh design.

4.2 Background and Motivation

Advancements in cloud computing and virtualization, have refined the architectures and design of modern software systems. Current trends in software engineering and cloud computing have encouraged adoption of microservice architectures managed by orchestration tools, such as service meshes.

4.2.1 Microservice Architectures

Traditionally, software has been designed as a singular, whole product. This practice, referred to as the *monolithic software architecture* [84] provides straightforward designs, but are highly limited in terms of scalability, manageability and speed of deployments and revisions. Due to this,

modern software engineering practices have trended towards the separation of software elements into *microservices* [130]. These simple, atomic elements of a larger software system are deployed in tandem with many other components to collaborate on high-level system tasks. These tasks may include user authentication, product purchasing, or inventory management [83]. Figure 2.3 illustrates an example microservice deployment, to demonstrate properties of microservice architectures. In this work, we utilize a subset of this deployment as a running example and experimental testbed to show ServiceWatch's security capabilities and features.

Bringing an increase in the speed to develop online applications, microservice architectures [63] are an emerging software engineering paradigm bringing key features to enterprise software systems. Additionally, virtualization has been a key enabling technology for the advent of microservice architectures in that many deployments leverage virtual machines or containers to host microservice code [64]. With extremely fast deployment times and increased efficiency by leveraging shared resources, containerization has played a major role in the adoption and growth of the microservice paradigm [15, 100, 110]. However, as enterprise software expands and the quantity of microservices deployed grows, the management, coordination, and security of these services is increasingly challenging. Service meshes are the latest solution, in a series of developments, to address the issue of microservice explosion in large-scale software. [44, 63, 110]

4.2.2 Service Meshes Overview

Service meshes embody the latest iteration of the DevOps toolset. Namely, due to the aforementioned microservice explosion challenge, managing and coordinating microservice "swarms" becomes increasingly difficult as the size of deployments increases [65, 83]. Figure 2.3 demonstrates this network architecture, showing deployed microservices operating a "service proxy" alongside their microservice code, which implements the business logic of a given component. The service proxy, a key element of the service mesh, handles the network connections and security responsibilities of the microservice deployment. By outsourcing these responsibilities from microservice code, developers may focus on implementing business logic rather than networking and security

functionality [9, 46, 55].

The current production-ready, state-of-art service mesh tools are Consul, Linkerd, Istio, OSM, and Kuma. [10, 47, 60, 73, 98]. Interest in service meshes has been growing in recent years, following a trend of adoption relative to other popular portions of the DevOps toolset, such as Kubernetes [69], a container orchestration platform. In addition to increased interest in contributors, stars and forks on GitHub [38], the annual developer survey conducted by the Cloud Native Computing Foundation (CNCF) [17] has shown that Kubernetes and service meshes have increased in both adoption and popularity across 2020 and 2021 [18, 19] (the CNCF 2022 survey has not yet been released as of the writing of this article). As adoption of these tools increases, so too does their maturity and the size of microservice deployments in production environments.

While the term service mesh is agreed upon in literature and the feature-set that is provided by service meshes is fairly consistent among the current state-of-art, the implementation and design choices of service mesh technologies vary. For example, Consul may be deployed as a standalone, platform-independent service mesh able to be utilized as a single binary on a range of operating systems and technologies, or as an "overlay" service mesh atop Kubernetes [50]. We use the term "overlay" to indicate a service mesh that is reliant upon an underlying container orchestration platform, such as Kubernetes, to function properly. For example, the Istio and Linkerd service meshes are permanently reliant upon an underlying Kubernetes infrastructure to provide the necessary structure and capabilities to operate correctly. Without an underlying Kubernetes implementation, all overlay service meshes (Istio, Linkerd, OSM, Kuma, etc.) are unable to provide any of the purported functionality to a microservice deployment. Due to this reliance, the associated operational costs and expertise requirements of Kubernetes are inherent when adopting any overlay service mesh.

Within ServiceWatch and Consul, rather than depending upon an orchestration platform, like Kubernetes, orchestration and management of the cluster is conducted by a specially configured group of higher-privilege nodes referred to as a "quorum" that handles the consensus and management functionality for a deployment. Kubernetes, when configured for enterprise environments,

also operates in a "multi-master" mode (with similar responsibilities to a ServiceWatch/Consul quorum). When Kubernetes is configured in this way, all "masters" are replicated with identical security artifacts and replicate cluster state changes amongst the group. While the design and structure of Kubernetes is quite different relative to platform-independent service meshes, such as ServiceWatch and Consul, both of these service meshes can be adapted and utilized within Kubernetes deployments [50], showing a greater flexibility and freedom over the more strict overlay service meshes.

4.2.3 Security Incident Handling in Existing Service Meshes

While appealing to software engineers and system architects, service meshes introduce new complexity, attack surface, and challenges to these environments. A primary feature lacking in all modern service meshes is the ability to detect misconfigurations or misbehavior within deployed microservices and effectively report or mitigate these incidents. Due to service proxy design decisions present amongst *all* state-of-art service meshes, there is no mechanism to detect, and effectively notify administrators, when access control policies are violated [9, 46, 55].

All access control decisions for microservices are made at their local service proxy which can lead to issues, such as resource constraints due to the service proxy's lightweight nature, a constrained view of the overall system structure, and no mechanism to alert administrators or change the offending microservice. Similarly, if a service proxy is misconfigured to have mutual-TLS connections disabled while the remainder of the deployment has this configuration active, there is no means to alert an administrator or remediate the specific misconfiguration issue. The likelihood that a mistake or misconfiguration has occurred in the declaration of a microservice deployment is increasingly likely as the size of these deployments increases. For example, Netflix, Spotify, and Uber have all reported deployment sizes of more than 1000 microservices in 2021 [20, 68]. Additionally, Twitter boasted more than 10,000 microservices at a LinuxCon presentation in 2016 [7]. With system sizes this large, the opportunity for misconfigurations and mishaps to occur is significantly more likely, as was the case with Twitter recently [23, 112], and the attack

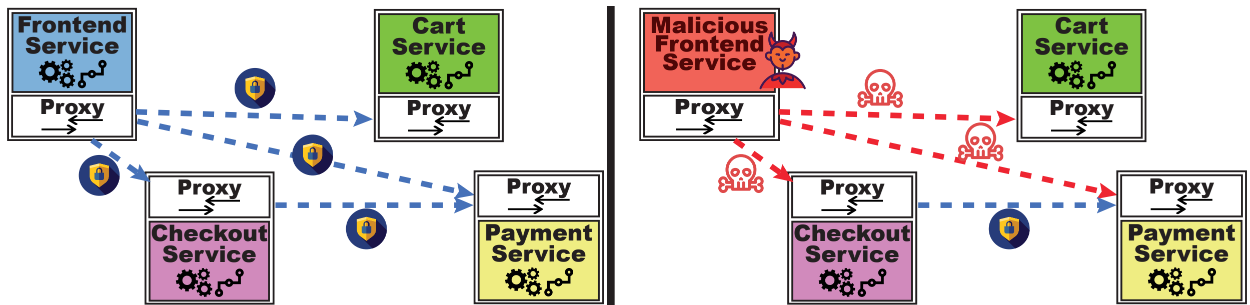

Figure 4.1: **Threat Model Overview** – With known issues in microservices and containerized applications shown in [76, 119, 124], the threat of compromised microservices must be considered in service mesh operations. Therefore, the interactions between microservices have the potential for adversarial influence.

surface of such a system also grows and becomes a more appealing target for adversaries. This reality demonstrates that service mesh security must match the microservice domain and provide mechanisms for detection, isolation, and recovery from misbehaving microservices.

4.3 Threat Model and Security Capabilities

Due to the diverse, distributed, and resource-sharing properties of microservice architectures, there are a range of potential threats to service meshes and the microservices they manage and maintain. As part of this study, we adopt a threat model that assumes microservice code to be untrustworthy and that microservices are deployed within a malleable, inconsistent network perimeter. These assumptions have been confirmed in previous research [76, 77, 119, 124] and exist in real-world deployments [54], aligning with our domain of study. Under these assumptions, we highlight the ability for our proof-of-concept, ServiceWatch, to demonstrate clear security benefits over current state-of-art service meshes.

4.3.1 Threat Model

The highly distributed nature of microservice deployments and the ability to utilize cloud infrastructure to distribute resources globally has created increased opportunities for adversaries to

threaten service meshes. We consider the two broad categories of external threats (adversaries with no prior access to service mesh resources) and microservice threats (adversaries originating from within service mesh resources, *e.g.* - a compromised microservice) as part of this work. Additionally, we deem direct compromise of ServiceWatch/Consul quorum nodes and overlay service mesh master nodes to be out-of-scope with respect to this work. We acknowledge that compromise of these elements is possible, but we believe that system administrators have a higher incentive and higher degree of control over these elements due to their increased level of privilege and the fact that they should *not* host microservices within service mesh deployments.

External Network Threats: The properties of cloud computing platforms and the globally distributed nature of modern enterprise software systems forces system administrators to maintain a malleable, inconsistent network perimeter with respect to their deployments. Due to the fact that microservice architectures have intricate and complex dependency relationships among the deployed microservices, direct network access to components that may be located in different datacenters or even different regions globally is often necessary. In this manner, there is an inherent lack of a distinct network perimeter for these deployments, causing some microservices to be directly exposed to shared, uncontrolled networks [54, 58]. As such, it is extremely important to consider the potential for skilled adversaries to monitor network traffic in attempts to eavesdrop or manipulate traffic moving across these networks. These standard, network-level attacks attempt to compromise the confidentiality, integrity, and availability of deployed microservices. Aside from the array of standard, network-level attacks available to external adversaries, we also consider the more severe potential of microservice threats that are the result of a compromised node or container.

Microservice Threats: With the potential of compromised microservice containers [76, 77, 119, 124], third-party software application compromise, and software supply-chain vulnerabilities [92], opportunities for an adversary to gain access to microservice deployments via deployed applications are increasingly concerning. A prime example of such an issue is the Log4j vulnerability [128], allowing remote-code-execution that has the potential to provide attackers with access to

the service mesh network after compromise. Within this study, we consider the potential for single-node or single-microservice compromise to be possible for an adversary. From this position, we assume an adversary has complete control over the component and service mesh proxy instance. Using this vantage point, they may initiate new network connections not explicitly defined within the microservice application code and may attempt to move or expand horizontally within the microservice deployment. However, as discussed previously, we exclude the ServiceWatch/Consul quorum instances and Kubernetes masters from potential insider threat compromise because they are known to have higher levels of privilege and responsibility within a service mesh deployment and are not intended to host microservice application code. In this way, we believe that system administrators have a larger incentive for increased scrutiny with respect to these components to prevent complete compromise of the service mesh deployment.

4.3.2 Security Capabilities

To combat these described threats, we use traditional security mechanisms already present within state-of-art service mesh design, but also introduce novel features and design changes unique amongst currently available service meshes. Aiming to uphold strong network security guarantees, freshness of security artifacts, and system-wide access control, ServiceWatch provides security design that better meets the needs of the microservice domain.

Misbehavior and Misconfiguration Mitigation: As described in Section 4.2, service meshes provide "management" functionality within the *control-plane* of a microservice deployment, while the microservices themselves are a "managed" element, within the *data-plane*. The control-plane is responsible for infrastructure management tasks, such as securing requests between microservices and enforcing access control policies. ServiceWatch's novel security functionality draws many similarities to self-adaptive software systems [5, 114]. However, we adopt these strategies to detect and mitigate nefarious or incorrect behaviors in microservices, rather than improving system throughput or performance.

Unique to modern service meshes, ServiceWatch provides additional security guarantees re-

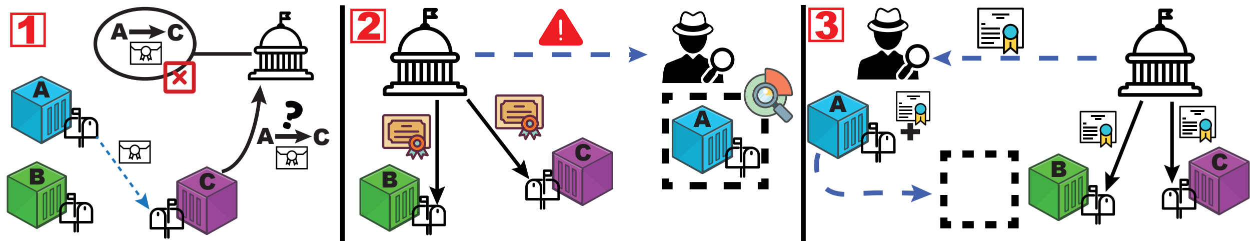

Figure 4.2: **Access Control Verification System** – The ServiceWatch quorum participates in the approval or denial of requests made between microservices. In this way, (t) should a service be misbehaving or be misconfigured, the quorum can deny the request and then mark the service for isolation and investigation. $(t+1)$ Fresh certificates will not be distributed to this service and a system administrator will be notified. To reintroduce the service to the active cluster, $(t+2)$ the administrator, or automated reintegration system, will simply attach a fresh set of artifacts to the service and integrate it to the active environment.

lated to the enforcement of access control. By involving the ServiceWatch quorum nodes in the approval and denial of access control tasks from microservice requests, the quorum may make informed decisions about the distribution of fresh security artifacts in future rotation periods. This process, illustrated in Figure 4.2, allows ServiceWatch to provide automated detection, isolation, and remediation of misbehavior within a microservice deployment. Additionally, by empowering all quorum members to verify access control requests, ServiceWatch prevents the possibility of a single-point-of-failure in this process.

ServiceWatch implements this informed revocation mechanism by making use of the frequent, coordinated security artifact rotations that occur regularly within the deployment. Every rotation within a ServiceWatch deployment involves a set of processes to generate, distribute, and activate a collection of security artifacts among the nodes in the cluster. However, based upon microservice behavior and node availability, the set of nodes for which security artifacts are generated is variable and is subject to adaptations within the cluster. These adaptations, controlled and managed collectively by the ServiceWatch quorum are able to be modified automatically by quorum members that detect issues, or manually by a system administrator wishing to implement deployment changes.

Work by Pauley, *et al.* [101] has shown that the dynamic nature and frequent scaling events in cloud resources can be exploited due to latent configurations issues present within deployments.

Figure 4.2 depicts a scenario in which a latent configuration in microservice A has caused misbehavior in the microservice deployment. At time period t, microservice A has been misconfigured to improperly request resources from microservice C instead of microservice B to which it was originally intended to receive information from. The ServiceWatch quorum during regular operations will detect these improper requests and mark microservice A for investigation. When the rotation occurring at time $t+1$ occurs, microservice A will be seamlessly removed from the cluster because it has not received a fresh set of security artifacts and the system administrator will be notified to investigate the reason for A's removal. Following an investigation, the administrator may solve the misconfiguration present within A and mark it for reintroduction within a deployment manifest hosted by the ServiceWatch quorum. This results in the quorum at time $t+2$ generating a valid set of artifacts for microservice A and allowing its reintroduction with valid credentials. Similarly, compromised microservices, are isolated, allowing administrators to investigate threats as they occur. These novel capabilities allow ServiceWatch to appropriately monitor and secure highly dynamic microservice deployments without impeding the benefits service meshes provide to these environments.

Zero-Trust Networking: The premise of zero-trust networking is to verify all connections, identities, and content exchanges between microservices within a service mesh, rather than relying upon inherent trust within the deployed components [111]. This approach matches the needs of the microservice domain closely and is implemented as an explicit design goal within ServiceWatch.

Similar to the current state-of-art, ServiceWatch adopts these accepted security mechanisms for zero-trust networking capabilities. Namely, ServiceWatch makes use of symmetric key encryption for membership messages, naming, and service metadata exchanged between nodes; x509 TLS certificates to create and secure mutual-TLS connections between services; and access control policies and tokens to ensure proper authorization for requests made between services in the deployment. While the exact implementation of these security mechanisms varies slightly among service meshes, all state-of-art tools either provide a form of each of these mechanisms or depend upon an underlying platform. In the case of overlay service meshes, Kubernetes provides this

Service Mesh	Automatic Rotations	Attack Window (Data Plane)	Attack Window (Control Plane)	Time to Alert
ServiceWatch	✔	*1 min**	*1 min**	*Immediate*
Consul	◒	∞	∞	∞
Istio	◒	24 hours [30]	≥1 year [30]	∞
Linkerd	◒	24 hours [6]	1 year [6]	∞
OSM	◒	24 hours [47]	1 year [47]	∞
Kuma	◒	24 days [40]	≥1 year [40]	∞

Table 4.1: **Security Analysis of State-of-Art Service Meshes** – Support for domain-relevant security features in current state-of-art service meshes. ServiceWatch supports very short attack windows and immediate notification of violations, while alternative meshes have long windows of attacker opportunity and fail to report occurrences of violations. **1 minute is the current configured time for rotations within ServiceWatch, however, it is customizable and may be reconfigured as needed for environment or domain needs.*

functionality [52, 62, 71, 78].

In contrast to other service meshes, ServiceWatch implements mandatory zero-trust networking, enabling a "secure-by-default" approach to the service mesh. An additional concern to the long-term security of a service mesh is the exposure of security artifacts for long periods of time. ServiceWatch's design prevents security from dwindling over the lifetime of a deployment by refreshing security artifacts automatically, maintaining freshness and avoiding expiration and long-term exposure. Alternative service meshes may be capable of performing security artifact rotations, however, these processes involve manual administrator action and may be infeasible as deployments scale. ServiceWatch's ability to notify system administrators of access control violations is novel to service meshes. Due to their presence within the access control verification loop, ServiceWatch quorum members have unique insights to microservice behavior and activity. When a violation occurs, ServiceWatch notifies administrators immediately, while alternative tools have no alerting capabilities because access control is verified at the recipient service proxy.

Attack Window: The highly dynamic nature of microservice deployments and the scaling-up and scaling-down capabilities provided by cloud platforms have strained and challenged the manage-

ment solutions available to administrators [63]. As such, questions regarding the ability for service meshes to be long-lived platforms arise [45]. As Table 4.1 shows, Consul, fails to provide meaningful support for automatic rotation and refreshment of security artifacts used within deployments without requiring manual administrator intervention [52]. Additionally, overlay service meshes that depend upon Kubernetes, while gaining some beneficial security features such as data-plane certificate rotation from the underlying Kubernetes infrastructure, fail to accommodate control-plane rotations as well, falling short of complete security artifact rotation [11, 59]. In this way, manual, system administrator intervention is required to rotate and refresh the certificates of this system as time progresses, often resulting in necessary system downtime. With certificate lifetimes of more than a year in some overlay service meshes, the potential for certificate exposure increases as well.

Table 4.1 illustrates the benefits of the security and design choices made within ServiceWatch relative to alternative service mesh tools. ServiceWatch accommodates complete security artifact rotation as part of its design and provides mechanisms for completing this task in an automated, synchronized manner across all service mesh components. Distributed generation and distribution of security artifacts occur within the quorum of higher-privileged ServiceWatch nodes, and are then distributed to other components through synchronized file downloads. Lastly, all service mesh components are notified to perform the necessary configuration changes to rotate to the new set of artifacts and this changeover occurs using a lively certificate rotation mechanism as part of the ServiceWatch service proxy. As the service mesh transitions from one set of security artifacts to another, we observe no noticeable downtime or drop in microservice availability as part of this process. In contrast, alternative service mesh technologies may not support automatic security artifact rotations, or may only support a subset of the artifacts to be rotated automatically. Additionally, alternative service meshes expose security artifacts for significant periods of time before retiring their use in a deployment, extending an adversary's ability to use compromised artifacts.

4.4 ServiceWatch Design and Evaluation

ServiceWatch expands upon the current state-of-art in service meshes by employing a scalable design, while also providing a novel approach for enabling unique, domain-conscious capabilities to the microservice space. Through its ability to detect, isolate, and alert upon potentially malicious or misconfigured services, ServiceWatch provides the next logical step in service mesh design by enabling service meshes to properly manage and enforce security across the microservices they are intended to manage and maintain.

4.4.1 Malicious Service Mitigation

Leveraging a security artifact rotation process, ServiceWatch is capable of seamlessly isolating and mitigating microservice compromise within a microservice cluster. The issue of microservice compromise, container compromise, and latent issues within microservice code have been studied and have shown that inherent trust of microservice behavior is dangerous to large-scale systems. ServiceWatch addresses this issue by providing a mechanism to identify when authorization violations occur within microservices and remediate these events in a timely manner. Our process of access control verification (described in detail below) allows ServiceWatch to perform a high-level security evaluation of microservices that make requests within a service mesh and to seamlessly remove them from the operational environment to be examined and remediated. Figure 4.2 illustrates this process in panels $(t+1)$ and $(t+2)$. By marking suspicious services for removal from the operational cluster and alerting system administrators, the benign operation of the microservice cluster may resume without issue and the overall security posture of the cluster may be maintained. Once remediated, replaced, or reconfigured, the offending microservice may be easily reintroduced to the operational environment by updating a service mesh inventory file and distributing fresh security artifacts to the entity.

4.4.2 Attack Window

Through the automatic and frequent security artifact rotations present within ServiceWatch, guarantees of freshness and up-to-date cluster state are ensured. Additionally, ServiceWatch makes use of an update-able system manifest (either by the ServiceWatch quorum or system administrator) to determine which entities in a microservice deployment are provided fresh security artifacts, and subsequently which entities remain a part of the microservice cluster. Within ServiceWatch, security artifact rotations of all encryption keys, certificate-key pairs, and access control tokens, are synchronized across the cluster at regular intervals and ensure that artifacts deployed within the environment are never *stale*, or in other words, exposed for extended periods of time. This characteristic of limited exposure provides a constrained window of time for an attacker to attempt to discover and make use of any of the security-sensitive artifacts within the cluster. In conjunction with the *verification-in-the-loop* feature of ServiceWatch, these capabilities combine to limit the actions and effects that an attacker may have against the microservice deployment before they are detected and removed from the operational environment. As defined in [6], we see an *attack window* as a continuous time interval an attacker may leverage without being interrupted by system changes.

In the context of ServiceWatch, we utilize the following definition of "attack window" in this research:

> **Attack Window** – the period of time in which any given set of security artifacts are active within a ServiceWatch deployment.

Below we formalize this concept within our context.

$$W = t + T_r \tag{4.1}$$

$$T_r = T_{gen} + T_{dist} + T_{ch} \tag{4.2}$$

In Equation 4.1, W represents the attack window, or time in which an adversary has the potential to compromise or discover actively used security artifacts within a ServiceWatch cluster. The variable t represents the time between rotations that is configured by the system administrator of a ServiceWatch cluster. T_r represents the "time to rotate" that is required for the various processes in ServiceWatch to transition the cluster from one set of security artifacts to another. T_r is further expanded in Equation 4.2 showing its component elements.

The variables T_{gen}, T_{dist}, and T_{ch} represent the time required to generate a new set of artifacts, time required to distribute the new set of artifacts to members, and the time required to synchronize a changing of node configurations within the cluster, respectively. However, T_{gen} and T_{dist} may be performed as background processes that do not affect normal ServiceWatch cluster operations. As described in Section 4.3, the ServiceWatch quorum is responsible for the generation and distribution of security artifacts, meaning that microservice tasks and performance are not effected by this workload. The only component of a rotation that *requires* a synchronized response from cluster members and may potentially incur downtime within the cluster is T_{ch}, however, in our experimentation, we find that changeover time within ServiceWatch is extremely fast due to the use of lively certificate rotations and results in near-zero downtime rotations of artifacts. Specifically, the ServiceWatch process is never stopped during rotation of certificates and no noticeable or measurable effect to system downtime is observed. Currently, within the default ServiceWatch configuration, security artifact rotation frequency (t) is configured to occur every 60 seconds, resulting in over 1440 rotations in a 24-hour period. This configuration may be altered according to the application needs and the risk that a given system may accommodate, limited only by the time required to generate, distribute, and synchronize changing the set of artifacts.

4.4.3 Access Control Verification

In order to provide strong access control guarantees across microservice deployments, ServiceWatch introduces a novel *verification-in-the-loop* mechanism to the standard checks performed by service proxies when a microservice request is made through the service mesh data-plane. Fig-

ure 4.2 illustrates the control logic involved during these events. Examining panel (t) of the figure, microservice A makes a request to microservice C. Upon receipt, the service proxy intercepts the traffic bound for the target microservice and verifies that the request is authenticated, encrypted, and maintains integrity before beginning the checks for authorization. To verify that the requesting service is authorized to receive resources from microservice C, the request metadata and attached access control token(s) is forwarded to any member of the ServiceWatch quorum.

When a quorum member receives an authorization request, the same authenticity and integrity checks are performed between the verification requester and the quorum member before the list of policies and valid access control tokens are searched. If a valid token and policy are located, the quorum member responds with a binary `TRUE` value to microservice C and the resources requested by A are ultimately exchanged between microservices. Figure 4.2 also illustrates the process of handling verification denial. In the event of a microservice requesting a resource they do not have authorization to, a binary `FALSE` is sent to the verification requester, the connection between the two microservices is severed, and the requesting microservice is marked for isolation, excluded from future security artifact rotations, and an administrator is alerted to investigate the occurrence. By including the ServiceWatch quorum entities, in the access control verification loop, not only is ServiceWatch able to provide redundancy in enforcement of access control, but a holistic view of the overall health of the cluster is achieved. From this position of oversight, quorum members make informed decisions as to which microservices are deemed "healthy" and which should receive updated, fresh security artifacts during the next rotation period.

4.4.4 Experimental Evaluation – Access Control Verification

When a resource request is made between any two microservices in a ServiceWatch deployment, the service proxies first examine and verify the authenticity and integrity of every message received, then extract the attached authorization headers and forward the request's metadata along with the authorization header to the quorum for verification. In our experimental testbed environment, based upon the Google Cloud Platform "Online Boutique" example microservice deploy-

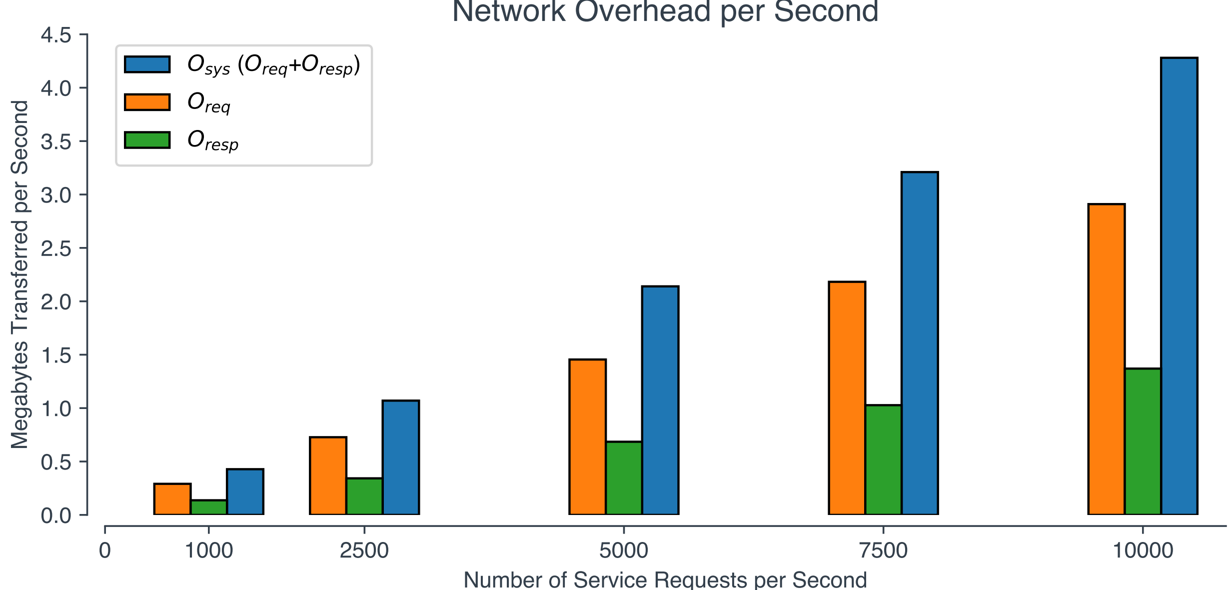

Figure 4.3: **Theoretical Network Overhead** – Using the experimental data collected by our small-scale testbed, we plot the overall system overhead (O_{sys}) expected and the component parts (O_{req} and O_{resp}). Even as the system scales to tens of thousands of requests per second, the system-wide overhead remains manageable. With 10,000 microservice verification requests per second in the cluster, we observe a system network load of ~4.28 MB/sec.

ment [103], we find that the network cost of this process requires a single TLS connection from the recipient service to the quorum carrying a payload size of 290.91 bytes on average. The subsequent verification decision that the quorum responds with was found to be 137.2 bytes on average. These results coincide with the system-level design of ServiceWatch, requiring only small amounts of request metadata to be included in the verification request and a simple binary response necessary to complete the quorum verification loop.

4.4.5 Theoretical Evaluation – Access Control Verification

Expanding upon our real-world observations of ServiceWatch's network burden, we formalize the system-level overhead using the following equation:

$$O_{sys} = (O_{req} + O_{resp}) * n \tag{4.3}$$

In Equation 4.3, the overall system-level overhead (O_{sys}) is computed by a summation of the size of verification request payloads (O_{req}) and the size of verification response payloads (O_{resp}) multiplied by the number of service requests (n) occurring across the entirety of the cluster at any given moment. Figure 4.3 illustrates this computation as the size of the cluster increases (and the subsequent number of service requests between microservices also increases).

As shown by the figure, the overall system cost for providing our novel *verification-in-the-loop* for microservice requests scales linearly with the number of requests generated within the deployment. In the event of 10,000 microservice verification requests per second, the *overall* system network load is ∼4.28 MB/sec. However, the above calculation represents the total cost of network load to the entire microservice deployment, but does not depict the network load imposed upon each of the ServiceWatch quorum nodes individually. ServiceWatch allows service proxies to choose a verifier among any of the quorum members because the verification token log is shared amongst all members. Due to this, verification overhead is distributed across the quorum. As part of our experimentation, we utilized 5 quorum nodes, splitting the cost equally across members. ServiceWatch is able to accommodate any number of quorum nodes, but tradeoffs of system cost, network overhead of quorum maintenance, and system complexity should all be considered when designing a microservice deployment.

4.5 Related Work

Microservice and Container Security: The security structure of microservices and containerized applications has been studied closely and been shown to be vulnerable to exploits in a number of previous works. Rastogi, *et al.* [108] evaluate an automation system for decomposing a monolithic software deployment into a collection of collaborating microservices, the purpose of which is to better adhere to the principle of least privilege [115]. As a part of this effort, Rastogi, *et al.* developed a special-purpose system that uses a static binding to communicate between microservices, rather than an examination of the available security mechanisms within service meshes that dy-

namically connect services. A lack of security protections in the Docker container environment is noted by Yarygina, *et al*. [137] and they propose a security monitoring system for containers as a potential solution to this issue. Such a system could be leveraged as part of the long-term security of a microservice deployment that utilizes the ServiceWatch service mesh. Trust within deployed microservices is often inherent, as noted by Sun, *et al*. [124]. The authors study how the trust relationship between deployed microservices may result in the compromise of an entire system. Further, they propose a system for deploying network security monitors in microservice environments to detect and block threats to clusters. ServiceWatch extends this work by providing zero-trust, communication security as a feature of the service mesh. In this way, exploits and attacks against containers are prevented from corrupting other portions of the service mesh cluster.

Access Control in Service Meshes: In a similar manner, work by Li, *et al*. [76] studied how inherent trust among microservices can be exploited by an insider threat. Assuming the threat model of a compromised microservice that attempts to make unauthorized requests to other microservices, the authors design and implement a solution for scanning microservice source code and extracting relevant metadata relating to network requests. With this extraction, they generate a minimal set of the necessary Access Control List (ACL) entries necessary for the service mesh to operate correctly. The proposed system design addresses the primary concern of microservice explosion by creating an automated methodology for determining the appropriate set of access control policies for a microservice deployment. This design could be utilized within ServiceWatch to automatically generate a minimal set of network relationships and the necessary access control tokens for the network connections between microservices deployed in a ServiceWatch service mesh. Afterwards, ServiceWatch's novel *verification-in-the-loop* capability would enforce those policies.

4.6 Summary

ServiceWatch expands upon the current state-of-art, scalable service mesh technologies by enabling service meshes to consider authorization requests between microservices as an indicator of

overall system health. Through a novel detection, isolation, and mitigation scheme, ServiceWatch introduces the next logical step in service mesh security mechanisms to the domain. Through an access control monitoring system that involves the privileged quorum nodes and scales appropriately with microservice deployments, ServiceWatch provides effective security, even as the microservice explosion phenomenon challenges these environments. ServiceWatch embodies a design for future service meshes that provides them the means to manage and maintain the health of microservice clusters autonomously, easing administrator burden.

Chapter 5

Multi-Hop Access Control in Service Meshes

5.1 Introduction

The explosion in popularity of online software services and applications has created a dramatic increase in the size, complexity, and interconnectedness of software deployments. An enabling factor to this has been cloud computing and the development of novel computing abstractions such as virtualization and containerization. With these advances, the structure and strategy of deploying software has shifted from large, single-server applications to distributed, network-connected components referred to as *microservices* [63]. This transition from *monolithic* software design to large pools of collaborating microservices has enabled software engineers to develop systems that are more reliable, robust, and scalable, relative to monolithic software applications [63, 120]. Further, the separation and isolation of software components has allowed developers to test, debug, and scale their applications in a more focused fashion. However, adoption and growing popularity of the microservices paradigm has lead to a significant increase in the quantity of unique software elements that must be deployed and an explosion in the number of dependency relationships that exist between these components. Referred to as the *microservice explosion* [64, 125] challenge, this phenomenon of large, interconnected webs of unique services has strained the management and security methods of modern software systems.

With modern tooling and practices, broadly referred to as *DevOps*, emerging to address the microservice explosion challenge, the construction and management of microservice architectures is changing rapidly. *Service meshes*, the latest in a series of iterations in DevOps have become appealing ways to manage and deploy microservice architectures with a number of state-of-art im-

plementations available, such as Istio [60], Linkerd [10], and Consul [47]. Additionally, platform orchestrators such as Kubernetes [69] address the challenge of rapidly deploying and managing the computation infrastructure upon which microservices operate. Despite these advancements, open challenges and security concerns still remain in the microservices domain. Namely, concerns over inherent trust between containers that host microservices [124], software-supply-chain issues [92], and critical flaws in microservice software [66, 126] all provide potential for adversaries to gain footholds within microservice deployments and escalate their privilege and capabilities within target systems.

Due to these open vulnerabilities and challenges within microservices and container technology, we identify two novel threats that are previously unaddressed by related work and state-of-art technologies. Namely, we refer to these threats as the "confused deputy" attack and the "short circuit" attack. These threats are enabled due to the complex dependency relationships that exist between microservices. With such a complex web of connections and paths between microservices in real-world deployments [23, 112], these threats go uncaptured by existing methods in service meshes and access control. The confused deputy attack involves an adversary manipulating a valid service-to-service *hop*, but causing a benign service to create an invalid *path* to a resource the adversary should not have access to. The short circuit attack similarly involves an adversary controlling an intermediate hop in a microservice path, but truncating the valid path to generate an illegitimate hop that makes requests to a microservice without a valid origin request. The details of these threats are discussed further in Section 5.3. To alleviate these adversarial actions we design and implement a system able to handle such a complex threat model. Through robust static analysis methods and a holistic, system-wide view of access control, we are able to alleviate these issues in deployments managed by our proof-of-concept framework, *CloudCover*.

As part of this work, we introduce CloudCover, a novel access control verification framework that addresses the issue of microservice compromise within microservice deployments. We make use of modern, state-of-art tooling, but design and implement a novel, *verification-in-the-loop* approach to address the potential for adversarial presence and influence within a deployed ser-

vice mesh that hosts microservices. Verification-in-the-loop is enabled by deploying a standalone verification oracle alongside microservices and their service mesh service proxies. In this way, CloudCover implements a strong, stateful method for access control that maintains knowledge and history of the service-to-service requests as they propagate through a microservice deployment and traverse the existing dependencies. Through a static analysis of microservice source code, network requests made through source code APIs are traced through the entirety of the microservice deployment and are used to populate a dependency graph of service-to-service requests that occur within the cluster. By extracting the possible paths through the dependency graph, CloudCover provides insights and visibility into the complex and interconnected dependency relationships between the large number of unique microservices present within modern software. Using graph traversal, we use CloudCover to derive a series of network access policies that are then used to populate the ruleset needed for verification. Within our verification-in-the-loop system, CloudCover is involved in all access control decisions regarding service-to-service requests that occur within the microservice system. By providing request introspection and verification at every hop in a service request path, CloudCover is able to track and maintain long-term access control for the lifetime of a service request as it propagates through the environment. Providing authorization decisions to the service proxies located next to deployed microservices, CloudCover provides the necessary oversight and capabilities for service meshes to provide holistic, system-wide introspection into microservice behavior and actions. This unique and novel capability within CloudCover provides a vital security mechanism to service meshes that prevents adversaries from manipulating dependency relationships between services to achieve their goals. CloudCover achieves this protection with minimal overhead and is a drop-in system designed to be implemented in existing software systems that are currently deployed.

Combining novel designs and capabilities with existing tooling, CloudCover presents a novel framework for improved access control in service meshes that leverages existing administrator expertise and knowledge. Additionally, CloudCover effectively provides administrators a means to enforce zero-trust networking policies and apply the "principle of least privilege" to the mi-

croservices that they deploy, providing a complete and strong system-wide security policy for all microservice traffic.

As part of this work, we make the following contributions:

- We analyze and identify two novel, unaddressed threats within microservice environments caused by complex microservice dependency relationships.
- We present an open-source, scalable code-scanning methodology to expose microservice dependency relationships and we enumerate these dependencies for access control policy generation.
- We design and implement CloudCover, an industry-ready solution for multi-hop access control verification in microservices that ingests the dependency relationships between microservices and utilizes a novel *verification-in-the-loop* approach for policy enforcement.
- We evaluate CloudCover for its feasibility to be used in modern large-scale software systems with tens of thousands of microservice requests generated per second.

5.2 Background and Related Work

With cloud computing emerging as a dominant model of deploying distributed software for a global scale, DevOps tools and methodologies have been created to address the new challenges of managing and controlling these environments. With microservice architectures gaining popularity, service meshes have been embraced to provide networking and security logic that manages microservice behaviors and connections. CloudCover expands upon this current state-of-art with improvements to access control mechanisms in service meshes for microservice deployments.

5.2.1 Service Meshes and Microservice Architectures

The primary goal of service meshes is to provide network abstraction and security for microservices. In an effort to allow developers a simplified process of developing and deploying their soft-

ware, the responsibilities of service discovery, connection, and management has been delegated to service meshes. Leveraging *service proxies*, service meshes implement a *sidecar* application alongside microservice code that intercepts network traffic and requests to and from microservices. This adjacent positioning of service proxies allows them to provide traffic introspection and layer in security mechanisms such as access control and mutual-TLS (mTLS) for microservice applications. Figure 2.3 illustrates this network architecture and the implementation of service proxies that provide mTLS connections and access control between elements of a microservice deployment. Through these capabilities, confidentiality, integrity, authentication, and authorization are provided to microservice applications, but do not require any source code changes to the microservices themselves. By securing all network connections between microservices and verifying all requests, service meshes are a critical first step in microservice architectures towards adopting "zero-trust networking".

In the current state-of-art service meshes, access control authorization is conducted through a series of policies and traffic rules present in each of the deployed service proxies. These network policies, often requiring significant manual effort by system administrators, are consulted each time a network request is made to a target service mesh. Based upon the outcome of running these rules against the received traffic, the request is either denied entry to the microservice, or is accepted and the request payload is ultimately forwarded to the target microservice. However, under this model, access control policies are checked in isolation at the receiving service proxies and may leave the microservice deployment open to adversarial actions, such as the confused deputy and short circuit attacks. These threats, previously unaddressed in prior work, are discussed further in Section 5.3.

5.2.2 Static Analysis of Source Code

Previous work by Li, *et al.* [76] has demonstrated a capability of leveraging microservice source code as a ground truth for mapping the dependencies between microservices. Using static analysis methods to locate and extract the relevant connection information from programming language APIs, AutoArmor has shown a capability for generating a graph of microservice dependencies.

Extending this work, we leverage the dependencies that are extracted from the dependency graph demonstrated in AutoArmor as the policy foundation for CloudCover. CloudCover then populates a policy ruleset which is used as the foundation for the verification-in-the-loop authorization by the CloudCover verification oracle. CloudCover further addresses key shortcomings in the work by Li, *et al*. [76] and considers the potential for complex microservice dependencies that are not accommodated by existing service mesh technologies.

5.2.3 Related Work

Previous work in the microservice and service mesh security domain have sought to address a range of issues and challenges within these domains.

Container and Microservice Security: In recent years, growing concern and a range of vulnerabilities and exploits emerging within microservice applications and the container ecosystem have driven a range of research and development to alleviate these issues. With vulnerabilities, such as Log4j [85, 128] making headlines and leading to significant damage [36, 132] work to understand the existing vulnerabilities and security issues in containerized applications has emerged [66, 126]. To address these vulnerabilities and challenges, various works have explored applying principle of least privilege to microservice systems that were constructed from decomposed monolithic applications [108]. Others have examined the potential for utilizing security monitoring in container systems [137]. Additionally, work exploring the dangers of inherent trust within deployed microservices has been considered [124].

Access Control Methods in DevOps: Work by Nam, *et al*. [89] explores security of container networks and the vulnerabilities exposed by some container networking systems. They provide a proof-of-concept framework for mitigating these threats and enforcing proper container networking policies. CloudCover expands upon this work by providing proper access control for microservice deployments that operate at higher layers in the networking stack, specifically with service mesh networking rather than container networking, and we defend against existing vulnerabilities caused by complex microservice dependency relationships. However, these methods may be com-

bined to provide a defense-in-depth approach. Additionally, previous work has considered proper access control in a serverless application context [2, 25, 116]. However, this work is limited in scope due to its implementation for serverless applications specifically. Instead, CloudCover applies holistic, system-wide access control for service meshes which may be deployed alongside serverless applications due to their ability to be adopted into service mesh clusters [93]. Work by Li, *et al*. [76] has investigated the issue of inherent trust within microservices and how this practice may be exploited by an insider threat. In a similar manner, we adopt a threat model that assumes the potential of insider threats or microservice compromise with an attacker that attempts to take action within a microservice deployment. However, we expand upon this work by considering a threat model where the complex dependency relationships and connection maps within microservice deployments enable an attacker to achieve goals if not mitigated. Specifically, in [76] only single-hop service-to-service connections are considered and secured when in actuality, multi-hop paths exist in real-world deployments. Due to this, isolated access control methods are insufficient and a holistic, system-wide access control methodology is necessary. CloudCover expands upon this work by capturing these relationships and providing a proof-of-concept framework to address this issue.

5.3 Threat Model

A range of concerns and known issues plague the microservices domain, however, the benefits that this deployment and software engineering paradigm bring to large-scale systems continues to be attractive to system designers. Of particular concern are issues related to the hosted services themselves and the security of the containerized applications that perform the business logic of software systems. For example, software supply-chain issues, inherent trust among containers, latent misconfigurations within containers, and a host of container vulnerabilities have been previously found in research and reported to developers. Due to these factors, it is clear that deployed microservices should be untrusted during operation and should have only the minimum necessary permissions and access within a deployed environment.

5.3.1 State-of-Art Shortcomings

As part of this work, we assume that the previously discovered issues and vulnerabilities within the microservice domain have the potential to manifest within real-world deployments. As such, we stipulate that an adversary has the potential to compromise or maintain access to an arbitrary microservice deployed within a target environment. From this standpoint, the adversary has the capability to create arbitrary network connections, attempt to contact other microservices within the deployment, and even to maliciously make requests to other microservices. This threat model was previously presented and leveraged in work by Li *et al.* [76]. This work expands upon the threat model presented in [76] by introducing two novel threats to the microservice domain that are not addressed by the work presented by Li *et al.*. Namely, these threats are the *confused deputy attack* and the *request chain short circuit*. We do not make assumptions regarding adversarial actions with respect to tampering or modification of application data. We deem these out-of-scope and instead focus upon the ability for an adversary to create network requests to other microservices within the deployment. We elaborate on these threats and the characteristics of microservice deployments that enable their existence below.

5.3.2 Novel Attacks

Due to the extremely complex and interconnected nature of microservice deployments, it is highly challenging to accurately describe and map all of the potential connections that microservice may make between one another. As such, we identify a previously uncharacterized structure that exists within microservice deployments that we refer to as *multi-hop* connections. These types of connections can be thought of as "dependency chains". Beginning with an origin service, a target endpoint is selected and some amount of work or data is requested from the target. However, in order to fulfill this request, the original target service, deemed a "helper" must make a new request to another service before ultimately returning the final response to the origin service. This relationship of services requesting resources *via* other services is shown below in Figures 5.1 and 5.2. Through these behaviors and processes, more complex dependencies structures, or *paths*, are cre-

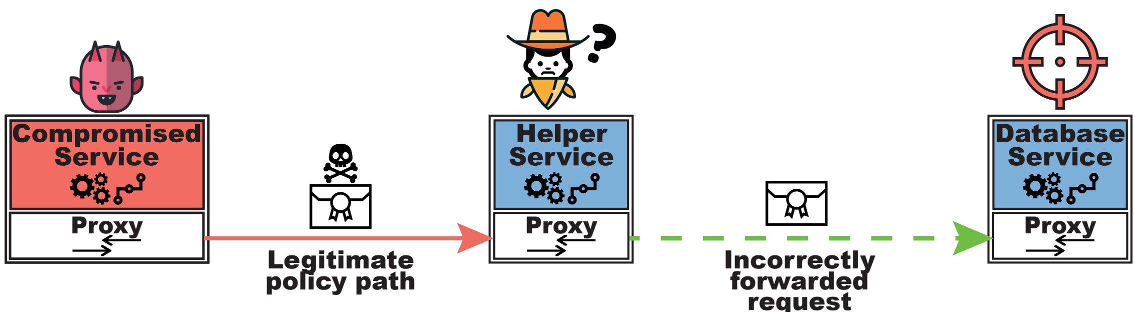

Figure 5.1: **Confused Deputy Attack**

ated within microservice architectures that are unable to be captured and addressed by *any* current state-of-art methods in service meshes or access control.

Confused Deputy Attack: Within the context of this work, we refer to the "Confused Deputy Attack" as one that occurs when an benign service is leveraged for malicious purposes via a legitimate intermediary hop. Due to current implementation of access control in service mesh tools, these multi-hop relationships are unable to be reasoned about in totality. Rather, current service mesh tools examine service-to-service requests in isolation, enforcing defined policies only at the recipient service proxy. This shortcoming allows an opening for adversaries to traverse legitimate paths and manipulate the intermediary, or "helper" services to do their bidding for them. In this way, the intermediary service is unknowingly participating in a malicious action and is a "confused deputy" on the behalf of the adversary. The scenario depicted in Figure 5.1 shows how a compromised service traverses a legitimate path between them and the deputy in order to make illegitimate requests against a terminal service, such as a database where user credentials or sensitive data may be stored. However, despite the strength of this attack under current service mesh design, the improvements and novel implementation of CloudCover allows us to detect and mitigate such threats in service meshes. The design of CloudCover is presented further in Section 5.4.

Short Circuit Attack: With intricate relationships between microservices in a deployment, we believe that an additional threat present within the domain is one that enables attackers to "short circuit" multi-hop service relationships. In other words, with the existence of a legitimate multi-hop path in a microservice deployment, the entirety of the path should be secured and access should

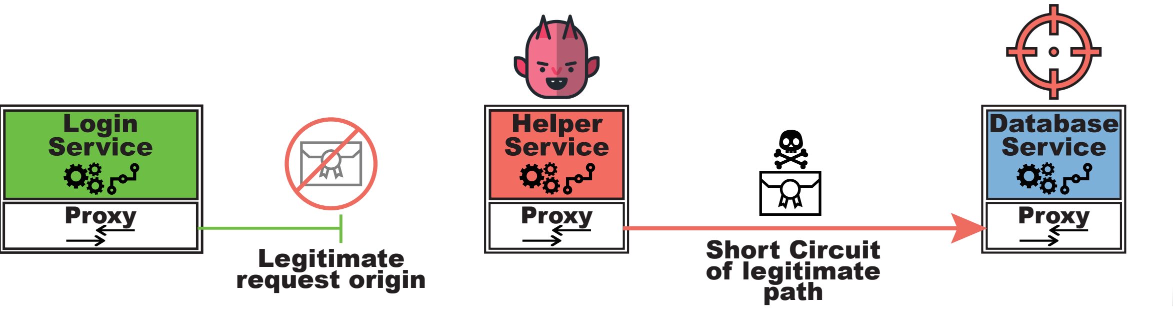

Figure 5.2: **Short Circuit Attack**

only be provided in the case where the origin service generates the beginning of the service chain. As Figure 5.2 shows, if an adversary has presence and control of a microservice that exists along a multi-hop service path, they may choose to generate illegitimate requests towards target services that exist further along the multi-hop path. Again, due to the fact that current service mesh design only account for isolated, single-hop service-to-service connections between elements of the deployment, this complex dependency relationship goes addressed. In this way, the adversary may bypass the requirement for the legitimate service to be the originator of the service request chain and may instead simply replay or generate new, illegitimate requests towards the target service, as is shown in Figure 5.2. By accounting for the entirety of the service request path and enforcing proper pre-requisite approval for service requests in a microservice deployment, CloudCover is able to prevent such threats in deployments, ensuring that "zero-trust" [111] networking is maintained and malicious presence within a microservice deployment is limited.

5.4 CloudCover

Utilizing the current state-of-art technologies in DevOps and service meshes, we design and implement CloudCover to address the shortcomings in access control enforcement for microservice deployments. We elaborate on our novel *verification-in-the-loop* design and describe the implementation testbed utilized to evaluate CloudCover for real-world deployments.

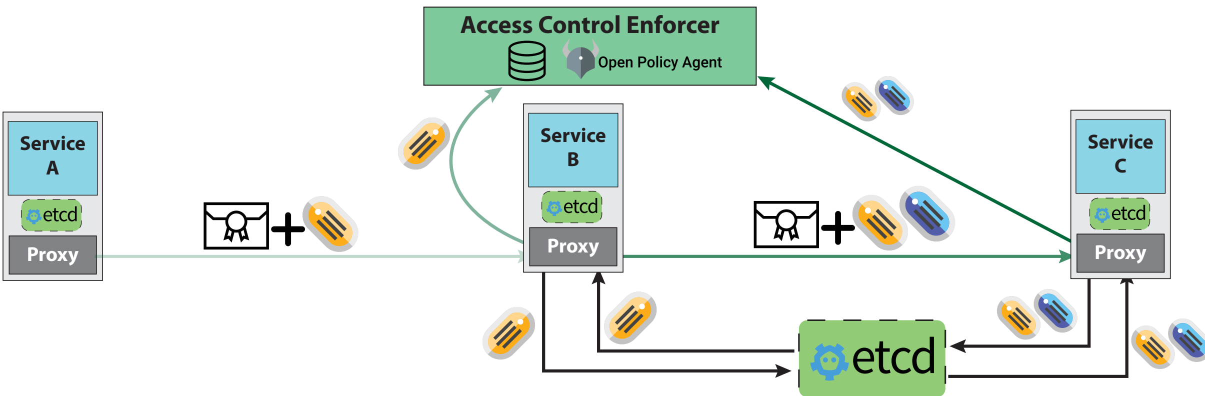

Figure 5.3: **Verification-in-the-Loop** – Service proxies in a microservice deployment consult with an access control verification oracle for decisions regarding approval or denial of all service-to-service requests. We implement this verification oracle using Open Policy Agent [95]. Additionally, to preserve the full context of microservice request paths, we leverage the etcd [30] key-value storage system to temporarily hold previous-path metadata. In our experiments, we deploy etcd as a standalone service or as a microservice sidecar container to explore performance benefits.

5.4.1 Architecture and Design

Due to the observed issues within the current state-of-art in microservice access control, holistic visibility and control over microservice behavior and request routing is critical in providing a solution to the noted threats. As such, we design CloudCover to operate as a supplemental access control system alongside the existing tooling used for deploying and operating microservice deployments. Namely, we focus upon deploying an access control verification oracle that is able to be contacted via the service proxies that are deployed at the network boundary of all microservices utilized within a service mesh. Instead of utilizing the built-in access control methods that are provided by the service mesh, we re-route all request verification to the verification oracle for approval. By re-routing this traffic to a standalone entity within the deployment, observation of microservice behavior can be captured and holistic, system-wide visibility into the security concerns of the cluster are possible. An additional concern however is that the normal service-to-service traffic conducted by the default service proxies within a service mesh do not carry all of the necessary request metadata to perform the access control verification tasks. To accomplish this, we deploy storage elements for temporary storage of previous request metadata so that a record or

"chain" of previous request information can be passed along with the service payload. Once this request metadata is compiled by the recipient service proxy, the verification request is made to the access control oracle, a verification decision is rendered, and the result is returned to the service proxy. If the verification decision is to accept the traffic, the request is stripped of any verification-in-the-loop materials, and is ultimately passed to the target microservice.

While this design necessitates new network connections be generated by the service mesh elements for effectively processing access control verification in CloudCover deployments, separating the access control logic from the service proxy logic is essential to provide the holistic, system-wide protection that CloudCover offers. Otherwise, administrators are left with isolated, single-hop access control verification that leaves microservice deployments open to potential compromise and attack. We provide an overview of CloudCover's implementation details and the setup for our experimental testbed below. Afterwards, in Section 5.5, we provide theoretical and experimental data to support the feasibility and scalability of CloudCover's deployment in real-world environments.

5.4.2 Implementation

We implement our previously described approach using industry-standard technologies, namely Kubernetes for container orchestration and management of workloads. Atop Kubernetes [69], we leverage the Istio [60] service mesh for service proxy capabilities and service management and discovery. Next, we make use of Open Policy Agent (OPA) [95] for providing the mechanism of enforcement for microservice requests made within the deployment and we utilize etcd [30] for storage of tokens and source chains from previous connections. The combination of these technologies and the overall structure of deployment is provided in Figure 5.4.

Next, we deploy experimental workloads of our proposed design to a private managed and operated cloud environment. This environment is comprised of 52 desktop hosts with identical hardware configurations of Intel i7-9700K (3.60GHz) CPUs and 16 GB of RAM. These desktop hosts are connected via a 1 Gbps switching network. Additionally, an industry-grade Dell PowerEdge [26] server was used for Kubernetes control-plane purposes to coordinate the deployment

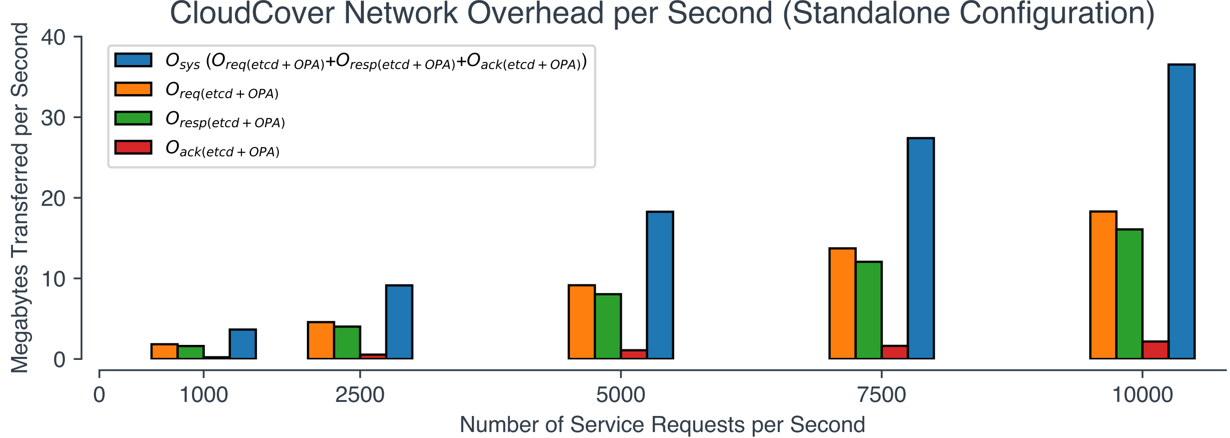

Figure 5.4: **Network Overhead of CloudCover (standalone configuration)** – Under a "standalone" deployment configuration where both OPA [95] and etcd [30] are deployed separately from the microservice deployment, we calculate the network overhead imposed. Under 10,000 service-to-service requests across microservices in the cluster occurring per second, we observe a network overhead of ∼36.54 MB/sec.

and management of workloads. To evaluate the feasibility and scalability of CloudCover for real-world workloads, we design a series of experiments to capture the network costs and performance behavior of CloudCover in different circumstances.

5.5 Evaluation

The evaluation of CloudCover is comprised of two components: an analysis and theoretical calculation of the network overhead imposed by CloudCover components and experimental microservice deployments under load. This approach is intended to elucidate both the security-performance tradeoff and the network overhead costs that exists when making use of CloudCover for microservice access control enforcement in service meshes.

5.5.1 Network Overhead Calculation

To evaluate the feasibility and practicality for CloudCover's deployment in real-world environments and applications, we first consider the network load imposed upon the microservice deploy-

ment due to network connections generated to CloudCover components. During initial testing and design, we implement CloudCover with OPA and etcd operating as separate services within the service mesh that are reachable via the service proxies operating alongside the deployed microservices. We refer to this configuration as the "standalone" configuration of CloudCover. To measure network cost and number of generated connections in the system, we leverage the `ksniff` [29] Kubernetes plugin that deploys an instance of `Wireshark` [35] at the Kubernetes pod boundary to collect and report network traffic that passes through the pod.

CloudCover Standalone Configuration: Based on our experiments and observations, in the "standalone" configuration, CloudCover requires 5 new network connections by the service mesh service proxy to facilitate CloudCover's verification-in-the-loop system. We mathematically represent the cost of these network connections below in Equation 5.1 and provide the observed cost values in bytes exchanged in Equation 5.2.

$$O_{sys} = (O_{req(e+OPA)} + O_{resp(e+OPA)} + O_{ack(e+OPA)}) * n \tag{5.1}$$

$$O_{sys} = (2673 + 2992 + 360) * n \tag{5.2}$$

Namely, 2 network connections are required on the inbound route to store the received CloudCover tokens, and the received CloudCover microservice hop history. Additionally, 1 network connection is required to be made to OPA for the access control request that determines whether the traffic is allowed to traverse to the target microservice. Lastly, 2 network connections are required on the outbound route to retrieve the stored tokens and hop history before passing the network request to the next target microservice in the request chain. Based on our results from `ksniff` experiments, the total cost of this process is 5 network connections with a total size of 6.025 KB exchanged as part of the TLS handshakes and data exchange between elements. Using this experimental data, we perform scalability calculations to extrapolate the overall network load imposed upon a microservice deployment that makes use of CloudCover. The results of the extrapolation of

CloudCover's cost in large-scale systems is provided in Figure 5.4. As we observe, in microservice deployments that generate 10,000 service-to-service requests per second, CloudCover imposes a total network load upon the deployment of roughly 36.54 MB per second, which we believe to be an acceptable cost for large, performant networks. The other component of cost that we consider with respect to network load when using CloudCover is the additional response latency imposed upon microservices by having to generate new network connections to the verification-in-the-loop system. These details are discussed below in Section 5.5.2 where we perform load testing upon CloudCover.

CloudCover Sidecar Configuration: In an effort to offset and improve the performance and limit the cost of CloudCover on microservice deployments, we design and implement an alternative configuration for CloudCover that makes use of `etcd` within a sidecar container that operates alongside of deployed microservice containers in a similar way to the service mesh service proxies. By adapting CloudCover in this way, we are able to avoid the need for 4 network connections to leave the pod boundary and make connections to the temporary storage element.

$$O_{sys} = (999 + 162 + 72) * n \tag{5.3}$$

Equation 5.3 represents the optimized cost of CloudCover when operating in the sidecar configuration and significantly decreases the network packets and size of packets that traverse pod boundaries and cause network load on the system. In this scenario, only 1.233 KB are exchanged as part of the verification-in-the-loop process and only 1 network connection is required to communicate with OPA for access control requests. Under these circumstances, for microservice deployments that generate 10,000 service-to-service requests per second, CloudCover imposes a total network load upon the deployment of about 12.25 MB per second. Details regarding network latency costs imposed under this configuration and the performance of CloudCover relative to a baseline, no security active configuration are provided below in Section 5.5.2.

Trial Type	Total Reqs	Avg. Latency (ms)	90% Latency (ms)	Reqs/sec
Default	37937	3909	4100	63.23
Istio Authz.	37265	3979	4300	62.11
CloudCover (standalone)	36962	4012	4200	61.60
CloudCover (sidecar)	38587	3844	4100	64.31

Table 5.1: **BookInfo Experiment Results** – Using Istio's BookInfo [56] example deployment as the target microservice system, we conduct load testing trials over a 10 minute time frame using Python's Locust library [81]. These experiments used a Locust configuration of 250 concurrent users, spawned at a rate of 25 new users per second, and full experiment duration of 10 minutes. The target page queried was the BookInfo ProductPage service and the endpoint of "/productpage".

5.5.2 Load Testing Experiments

To capture the practical overhead imposed upon a microservice deployment when using CloudCover, we conduct a series of performance trials using the Locust load testing framework. We test example microservice deployments provided by state-of-art cloud providers as example systems and additionally design a lightweight, multi-hop testbed using dummy services as a best-case microservice performance baseline. These experiments were conducted within our testbed environment as discussed previously in Section 5.4.

BookInfo Example: The BookInfo example deployment [56], provided by Istio, is comprised of four microservices. We deployed the example system in our testbed environment under various security configurations and present the results of these experimental trials with respect to total number of requests answered, average latency, 90% latency, and requests per second. The results of which are presented in Table 5.1 with all trials lasting 10 minutes. As seen by the results, the BookInfo example deployment behaves nearly identically in all trials conducted with every form of security configuration having little to no effect on the results. We believe that this is due to the microservices themselves consuming the vast majority of processing time associated with the requests. In addition to this behavior, we also observe that the request connections made to the BookInfo deployment are held open for a significant amount of time after the request is made. Even through attempts to manually close the connection via the load testing application, the BookInfo behaved at the rates observed in Table 5.1. The outcome of this series of trials indicates

Trial Type	Total Reqs	Avg. Latency (ms)	90% Latency (ms)	Reqs/sec	△ Reqs/sec
NoSec 1-hop	2389526	60	85	3982.54	—
NoSec 2-hop	1763016	82	140	2938.36	—
NoSec 3-hop	12233605	118	250	2056.01	—
NoSec 4-hop	923177	159	340	1538.63	—
NoSec 5-hop	732724	201	430	1221.21	—
Istio Authz 1-hop	2347843	61	88	3913.07	-69.47
Istio Authz 2-hop	1668969	87	150	2781.62	-156.74
Istio Authz 3-hop	1114017	131	260	1856.70	-199.31
Istio Authz 4-hop	791131	185	370	1318.55	-220.08
Istio Authz 5-hop	635204	232	450	1058.67	-162.54
CC Standalone 1-hop	717258	203	280	1195.43	-2387.11
CC Standalone 2-hop	406495	363	570	677.49	-2260.87
CC Standalone 3-hop	361766	410	580	602.94	-1453.07
CC Standalone 4-hop	267769	554	730	446.28	-1092.35
CC Standalone 5-hop	237863	624	800	396.44	-824.77
CC Sidecar 1-hop	1074254	136	240	1790.42	-2192.12
CC Sidecar 2-hop	644579	229	350	1074.30	-1864.06
CC Sidecar 3-hop	538086	275	450	896.81	-1159.20
CC Sidecar 4-hop	433796	341	550	722.99	-815.64
CC Sidecar 5-hop	314473	471	730	524.12	-697.09

Table 5.2: **Multi-Hop Fake Service Experiment Results** – Using a collection of `fake-service` [91] containers deployed as microservices and connected to one another via upstream dependencies, creating varying levels of service "hops", we conduct load testing trials over a 10 minute time frame using Python's Locust library [81]. Additionally, we vary the configuration and deployment strategy of CloudCover by operating etcd as a sidecar container for the `fake-service` applications. These experiments used a Locust configuration of 250 concurrent users, spawned at a rate of 25 new users per second, and full experiment duration of 10 minutes, which targeted the "/" endpoint of the first `fake-service` in the chain.

that when implemented with microservice systems that consume most of the request processing time, CloudCover has very little impact on the performance outcomes of the deployed system.

Fake Service Multi-Hop: To capture and compare the performance results of CloudCover in a microservice application that spends minimal time processing microservice requests, we conduct experimental trials with microservices emulated by the `fake-service` container deployment [91]. Under this configuration, the microservices themselves are extremely lightweight web server deployments that perform little to no manipulation on the incoming request information. Additionally, we construct trial scenarios with incrementally increasing numbers of service-to-service hops to represent "deep" microservice deployments where there are many dependent services that re-

quire the results of upstream service calls before ultimately returning their response to the origin of the request. The results of these experiments are presented in Table 5.2. As shown in the table, basic Istio authorization policies perform very near the baseline, no security deployments, however, Istio authorization policies only consider isolated, single-hop access control leaving deployments open to exploit from the confused deputy and request chain short circuit attacks discussed in Section 5.3. The Istio authorization policies are enforced adjacent to the microservices deployed and do not involve new network connections that leave the pod network boundary, resulting in little overhead over baseline performance. While in comparison, CloudCover configurations introduce new network connections in order to consult access control decisions with the OPA verification oracle. However, CloudCover mitigates the novel threats we consider as part of this work, providing a stronger, more holistic access control mechanism within service mesh deployments.

To offset the cost of verification-in-the-loop required network connections, we explore various configurations and deployment structures for CloudCover. Namely, we believe that access control in service meshes is a vital component to security microservice systems and that administrators will have incentive to ensure performant applications at the cost of additional computing nodes/hosts. Due to this, we deployed CloudCover in a replicated fashion and investigate the performance under various degrees of replication of OPA and etcd components. These results are presented in Table 5.3. A notable comparison within these scenarios is the baseline performance relative to basic Istio authorization policies relative to CloudCover when deployed with etcd as sidecars and OPA with 5 replications. Under this model, CloudCover is able to achieve performance very close to Istio and even close to baseline performance. Due to the fact that the microservices deployed are so lightweight and minimal, we believe that these results are promising for CloudCover to be adopted in real-world deployments to secure practical microservice systems.

5.6 Discussions and Limitations

When considering whether to centralize or decentralize access control in distributed environments, it is critical to consider a range of effects and tradeoffs. Specifically, performance and security

Trial Type	Total Reqs	Avg. Latency (ms)	90% Latency (ms)	Reqs/sec	△ Reqs/sec
NoSec 5-hop	732724	201	430	1221.21	—
Istio Authz 5-hop	635204	232	450	1058.67	-162.54
CC 1-OPA 2-etcd	338163	438	600	563.61	-657.60
CC 1-OPA 3-etcd	368907	402	570	614.85	-606.36
CC 1-OPA 4-etcd	379362	391	540	632.27	-588.94
CC 1-OPA 5-etcd	388362	382	510	647.27	-573.94
CC 2-OPA 1-etcd	222267	668	860	370.45	-850.76
CC 3-OPA 1-etcd	239349	620	800	398.91	-822.29
CC 4-OPA 1-etcd	246717	602	770	411.20	-810.01
CC 5-OPA 1-etcd	236343	628	810	393.91	-827.30
CC 2-OPA 2-etcd	352668	420	550	587.78	-633.43
CC 3-OPA 3-etcd	400608	370	490	667.68	-553.53
CC 4-OPA 4-etcd	413539	358	460	689.23	-531.98
CC 5-OPA 5-etcd	413915	358	460	689.86	-531.35
CC 2-OPA sidecar-etcd	466758	317	500	777.93	-443.28
CC 3-OPA sidecar-etcd	499264	290	470	832.11	-389.10
CC 4-OPA sidecar-etcd	529294	280	440	882.16	-339.05
CC 5-OPA sidecar-etcd	551640	268	420	919.40	-301.81

Table 5.3: **Multi-Hop Fake Service Replication Experiment Results** – Using a collection of `fake-service` [91] containers deployed as microservices and connected to one another via upstream dependencies for a total of 5-hops, we conduct load testing trials over a 10 minute time frame using Python's Locust library [81]. Additionally, we vary the configuration and deployment strategy of CloudCover by adding replicated instances of OPA [95] and etcd [30], but also exploring the effect of operating etcd as a sidecar container for the `fake-service` applications. These experiments used a Locust configuration of 250 concurrent users, spawned at a rate of 25 new users per second, and full experiment duration of 10 minutes, which targeted the "/" endpoint of the first `fake-service` in the chain.

are vital in environments that are designed to operate with high performance and strong reliability guarantees. Below, we discuss these considerations with respect to CloudCover and potential threats highlighted as part of this work.

CloudCover Deployment Strategies: Through our experimental observations of microservice systems under load, we observe performance loss due to the use of CloudCover. However, through informed deployment options, such as placing the etcd component in a sidecar container for the deployed microservices and replicating multiple instances of OPA for load balancing, a large portion of this cost is offset. Specifically, we observe the best performance of CloudCover under the sidecar etcd and 5 replicated instances of OPA scenario, achieving highly performant response rate and good levels of requests per second. Due to this, we recommend that administrators looking to implement CloudCover as part of their deployments consider following this configuration.

Security vs. Performance: As with any security solution, it is important to consider the security-performance tradeoff that exists by introducing new processes and tasks that may effect performance of a system. While cloud computing and microservice architectures are highly appealing due to their performance and reliability, security is a vital consideration that should be accounted for in system architecture and design. As we have discussed previously, CloudCover is the *only* access control solution for microservice architectures that addresses the novel threats of a confused deputy and request chain short circuit attack. Enforcing proper access control in large-scale systems becomes increasingly difficult to reason about and understand as a system administrator as the size of the deployment grows. Additionally, with large amounts of service-to-service requests occurring per second within these deployments, the traditional monitoring and observation tools present for microservices may not be able to report errors or provide administrators insights into microservice behavior in a timely manner to prevent such attacks from occurring within their environments. Therefore, deploying an automated, comprehensive security system such as CloudCover will significantly strengthen the security posture of these deployments without requiring administrator intervention or effort.

5.7 Summary

As microservice applications grow in their rate of adoption and popularity in modern software, the threats of microservice compromise and latent vulnerabilities bring new challenges to the security of the service mesh domain. CloudCover seeks to address these challenges by providing system-wide, holistic access control in service meshes. Through multi-hop dependency tracking that leverages static analysis source code scanning methods, CloudCover is able to track and monitor service-to-service requests that occur in microservice deployments. Additionally, through a novel verification-in-the-loop approach, CloudCover provides accurate verification for service requests at each step in the request propagation process. CloudCover embodies complete access control authorization security for the complex and interconnected nature of microservices that is not currently addressed by available tools and previous work.

Chapter 6

Conclusions and Future Work

The growing popularity and interest in microservice applications has stressed the current management and security systems available. While service meshes present appealing benefits and address many of the security needs present within the microservice environment, their current designs fall short in their ability to meet the dynamic properties of this domain. Through this work, we have demonstrated improvements and initial steps towards more closely aligning the service mesh security capabilities to the microservice domain. Specifically, we provide: (1) an initial investigation into supporting security artifact rotation automatically in service meshes; (2) a novel microservice misbehavior and misconfiguration detection, altering, and mitigation scheme for improving long-term health of microservice deployments; and (3) a novel framework for verification-in-the-loop access control in current state-of-art service meshes that addresses the needs to defend against previously unaddressed threats in microservice applications that exploit complex dependency relationships between services.

By examining the current state-of-art in service mesh technologies and considering the existing challenges and shortcomings in modern tools, we designed, implemented, and evaluated our prototype systems of ServiceFRESH and Anvil. Attempting to address the primary issues of security artifact freshness and the long-term security health of service mesh deployments, ServiceFRESH and Anvil provide an important analysis into the security-performance tradeoffs of enabling security artifact rotations within this domain. Some of the insights and capabilities presented within these works are now found within modern service meshes, such as lively artifact reloading capabilities, as seen in the Consul service mesh. In addition to providing security artifact rotations, Anvil explored the ability for a service mesh to perform informed artifact revocation. This secu-

rity artifact revocation was employed under the circumstances that a microservice be suspected of misbehavior or misconfiguration. This key insight and capability was critical to the development and implementation of ServiceWatch.

ServiceWatch leverages the underlying service mesh design and capabilities provided by the Anvil prototype, but explores the unique capability and effects of informed security artifact revocation. Service-to-service requests made as part of a ServiceWatch cluster are inspected by the ServiceWatch quorum to ensure that access control policies are enforced and maintained as part of the deployment. From their unique position of access control checking and security artifact generation, ServiceWatch quorum nodes are able to make informed decisions regarding which portions of a microservice deployment should receive refreshed artifacts. When a service-to-service request is denied for violating an access control policy, ServiceWatch marks the offending element for removal, automatically and seamlessly removing the element from the operational environment requiring no system administrator effort. In this way, ServiceWatch provides the service mesh domain with the unique and novel capability of microservice isolation and mitigation, expanding the current state-of-art to maintain security across the lifetime of the microservice deployment.

Building upon the concepts of ServiceWatch, we designed and implemented CloudCover. Within ServiceWatch's design, we integrate the privileged quorum nodes as a primary component in the process of access control between microservice instances. CloudCover expands upon this process by considering the underlying relationships between microservices. With incredibly complex and interconnected dependencies between microservices in real-world deployments, the potential for adversaries to manipulate these relationships is tremendous. Additionally, the current state-of-art in service mesh access control does not account for complex relationships between microservices that extend beyond single-hop relationships. Due to this, we identified two novel threats in the microservices domain that exploit the dependency structure of these systems and consider multi-hop relationships. To defend against these threats, we implemented a novel verification-in-the-loop approach that provides visibility to microservice request behavior in a system-wide, holistic manner. Through this capability, CloudCover is able to detect violations

in multi-hop request paths and deny the service-to-service requests from occurring, preventing the exploitation of our identified threats. CloudCover is a drop-in solution for modern service meshes and microservice applications, capable of supporting tens of thousands of microservice requests per second.

To expand upon this work for future development and research, we believe that the microservice architecture and service mesh domain is very promising for advancement and significant progress in securing software systems. Specifically, as part of this work, we present improvements to service mesh access control mechanisms through our framework, CloudCover. We believe that while CloudCover's current performance is feasible and acceptable in practical applications, there are improvements possible in the design and performance of the system. Namely, due to the nature of microservice applications and the desired property of "zero-trust" [111] within the cloud computing domain, there are design improvements possible in the structure of the system. We believe that possibility of compromised microservices present within a deployment encourages general distrust and suspicion of messages exchanged between microservices. Due to this, applying the verification-in-the-loop framework to a distributed blockchain [122, 138] hosted alongside deployed microservices as sidecar containers may bring performance improvements while still retaining the multi-hop verification capabilities currently available in CloudCover. In addition to framework design and improvements to the CloudCover implementation, we believe there is significant work available in the source code scanning and static analysis portion of this research. Namely, compiled and third-party applications that are used within a microservice deployment may present promising attack vectors to adversaries. Currently, CloudCover is able to scan available source code for microservice network requests, however, real-world systems often include third-party software or compiled applications where source code is unavailable. Under the threat of compromise to these systems, it is important for system administrators to still be able to exercise and identify which dependencies exist to these components and the proper access control policies to enforce upon their environments.

We provide these contributions, designs, and future work recommendations to the community

of cloud computing and microservice architectures to improve on the existing security mechanisms and designs for long-lived, highly scalable systems. Additionally, through our experimental evaluations of the discussed contributions, we present these results to demonstrate the feasibility of these designs for inclusion in existing technologies and solutions for real-world systems.

References

[1] Aksakalli, I. K., Celik, T., Can, A. B., & Tekinerdogan, B. (2021). Systematic approach for generation of feasible deployment alternatives for microservices. IEEE Access, 9, 29505–29529.

[2] Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., & Winstein, K. (2018). Secure serverless computing using dynamic information flow control. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1–26.

[3] Amazon Web Services (2023a). Amazon aurora (accessed 04/2023). https://aws.amazon.com/rds/aurora/.

[4] Amazon Web Services (2023b). Amazon elasticache (accessed 04/2023). https://aws.amazon.com/elasticache/.

[5] Andersson, J., De Lemos, R., Malek, S., & Weyns, D. (2009). Reflecting on self-adaptive software systems. In 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (pp. 38–47).: IEEE.

[6] Bardas, A. G., Sundaramurthy, S. C., Ou, X., & DeLoach, S. A. (2017). Mtd cbits: Moving target defense for cloud-based it systems. In Computer Security – ESORICS 2017 (pp. 167–186).: Springer International Publishing.

[7] Benedict, M. & Charanya, V. (2016). How we built a metering & chargeback system to incentivize higher resource utilization (accessed 07/2022). https://www.linux.com/training-tutorials/how-we-built-metering-chargeback-system-incentivize-higher-resource-utilization-michael/.

[8] Bouyant, Inc. (2023a). Architecture (accessed 07/2022). https://linkerd.io/2.12/reference/architecture/.

[9] Bouyant, Inc. (2023b). Proxy Configuration. https://linkerd.io/2.12/reference/proxy-configuration/ (accessed 07/2022).

[10] Buoyant, Inc. (2023a). Linkerd (accessed 01/2020). https://linkerd.io.

[11] Buoyant, Inc. (2023b). Manually rotating control plane tls credentials | linkerd (accessed 01/2020). https://linkerd.io/2.12/tasks/manually-rotating-control-plane-tls-credentials/.

[12] Burns, B. & Oppenheimer, D. (2016). Design patterns for container-based distributed systems. In 8th {USENIX} workshop on hot topics in cloud computing (HotCloud 16).

[13] bytebase (2022). Star-history.com, the missing github star history graph of github repos (accessed 07/2022). https://github.com/bytebase/star-history.

[14] Chandramouli, R. & Butcher, Z. (2020). Building secure microservices-based applications using service-mesh architecture. NIST Special Publication, 800, 204A.

[15] Chen, L. (2018). Microservices: Architecting for continuous delivery and devops. In 2018 IEEE International Conference on Software Architecture (ICSA) (pp. 39–397).

[16] Christopherson, J. (2017). Spaceflight uses HashiCorp Consul for Service Discovery and Runtime Configuration in their Hub-and-Spoke Network Architecture (accessed 02/2020). https://www.hashicorp.com/blog/spaceflight-uses-hashicorp-consul-for-service-discovery-and-real-time-updates-to-their-hub-and-spoke-network-architecture/.

[17] Cloud Native Computing Foundation (2023). CNCF Cloud Native Interactive Landscape (accessed 01/2020). https://landscape.cncf.io.

[18] Cloud Native Computing Foundation (CNCF) (2020). CNCF Survey 2020 (accessed 01/2022). https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[19] Cloud Native Computing Foundation (CNCF) (2022). CNCF Survey 2021 (accessed 10/2022). hhttps://www.cncf.io/wp-content/uploads/2022/02/CNCF-Annual-Survey-2021.pdf.

[20] CloudZero (2021). Netflix architecture: How much does netflix's aws cost? (accessed 07/2022). https://www.cloudzero.com/blog/netflix-aws.

[21] containerd Authors (2023). containerd – an industry-standard container runtime with an emphasis on simplicity, robustness and portability (accessed 04/2023). https://containerd.io/.

[22] Conway, S. (2018). Kubernetes is first cncf project to graduate (accessed 07/2022). https://www.cncf.io/blog/2018/03/06/kubernetes-first-cncf-project-graduate/.

[23] Corginia, C. (2022). Elon musk and twitter's system design (accessed 01/2023). https://corgicorporation.medium.com/elon-musk-and-twitters-system-design-8bc2a97680e6.

[24] Dadgar, A. (2023). What is Infrastructure as Code and Why is it Important? (accessed 04/2023). https://www.hashicorp.com/resources/what-is-infrastructure-as-code.

[25] Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., & Bates, A. (2020). Valve: Securing function workflows on serverless computing platforms. In Proceedings of The Web Conference 2020 (pp. 939–950).

[26] Dell (2022). Servers: Poweredge servers | dell usa (accessed 01/2022). https://www.dell.com/en-us/work/shop/dell-poweredge-servers/sc/servers?~ck=bt.

[27] Docker, Inc. (2023). Docker Home (accessed 04/2023). https://docker.io.

[28] Duvall, P. (2018). Measuring DevOps Success with Four Key Metrics (accessed 01/2020). https://stelligent.com/2018/12/21/measuring-devops-success-with-four-key-metrics/.

[29] eldadru (2023). ksniff (accessed 04/2023). https://github.com/eldadru/ksniff.

[30] etcd (2023). etcd (accessed 04/2023). https://etcd.io/.

[31] Fishner, K. (2014). How Lithium Technologies Uses Consul in a Hybrid-Cloud Infrastructure (accessed 02/2020). https://www.hashicorp.com/blog/how-lithium-technologies-uses-consul-in-a-hybrid-cloud-infrastructure/.

[32] Fishner, K. (2015a). Consul in a Microservices Environment at Neofonie GmbH (accessed 02/2020). https://www.hashicorp.com/blog/consul-in-a-microservices-environment-at-neofonie-gmbh/.

[33] Fishner, K. (2015b). How BitBrains/ASP4all uses Consul for Continuous Deployment across Development, Testing, Acceptance, and Production (accessed 02/2020). https://www.hashicorp.com/blog/how-bitbrains-asp4all-uses-consul/.

[34] Fishner, K. (2015c). Using Consul at Bol.com, the Largest Online Retailer in the Netherlands and Belgium (accessed 02/2020). https://www.hashicorp.com/blog/using-consul-at-bol-com-the-largest-online-retailer-in-the-netherlands-and-belgium/.

[35] Foundation, W. (2023). Wireshark (accessed 02/2020). https://www.wireshark.org/.

[36] FTC CTO, FTC DPIP staff, and the FTC AI Strategy team (2021). Ftc warns companies to remediate log4j security vulnerability (accessed 01/2023). https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2022/01/ftc-warns-companies-remediate-log4j-security-vulnerability.

[37] Fulton III, S. (2018). Service mesh: What it is and why it matters so much now (accessed 01/2020). https://www.zdnet.com/article/what-is-a-service-mesh-and-why-would-it-matter-so-much-now/.

[38] GitHub, Inc. (2023). Github (accessed 07/2022). https://github.com.

[39] Godard, S. (2022). pidstat(1) — linux manual page (accessed 01/2022). https://man7.org/linux/man-pages/man1/pidstat.1.html.

[40] Google (2022a). Github – pprof (accessed 01/2022). https://github.com/google/pprof.

[41] Google (2022b). The go programming language (accessed 01/2022). https://go.dev/.

[42] Gremlin, Inc. (2022). The cost of downtime for the top us ecommerce sites (accessed 07/2022). https://www.gremlin.com/ecommerce-cost-of-downtime/.

[43] Gupta, D., Saia, J., & Young, M. (2019). Peace Through Superior Puzzling: An Asymmetric Sybil Defense. In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 1083–1094).: IEEE.

[44] Haddad, E. (2015). Service-oriented architecture: Scaling the uber engineering codebase as we grow (accessed 01/2021). https://eng.uber.com/service-oriented-architecture/.

[45] Hahn, D. A., Davidson, D., & Bardas, A. G. (2020). Mismesh: Security issues and challenges in service meshes. In International Conference on Security and Privacy in Communication Systems (pp. 140–151).: Springer.

[46] HashiCorp (2023a). Connect Proxies. https://developer.hashicorp.com/consul/docs/connect/proxies (accessed 07/2022).

[47] HashiCorp (2023b). Consul by HashiCorp (accessed 01/2020). https://www.consul.io/index.html.

[48] HashiCorp (2023c). Download Consul (accessed 04/2023). https://developer.hashicorp.com/consul/downloads.

[49] HashiCorp (2023d). Github hashicorp/consul (accessed 02/2020). https://github.com/hashicorp/consul.

[50] Hashicorp (2023). Kubernetes (accessed 01/2020). https://www.consul.io/docs/k8s.

[51] HashiCorp (2023a). Manage ACL Policies | Consul (accessed 02/2020). https://developer.hashicorp.com/consul/tutorials/security/access-control-manage-policies.

[52] HashiCorp (2023b). Security Model (accessed 04/2023). https://developer.hashicorp.com/consul/docs/security.

[53] HashiCorp (2023c). Terraform by HashiCorp (accessed 04/2023). https://www.terraform.io/.

[54] Hashizume, K., Rosado, D. G., Fernández-Medina, E., & Fernandez, E. B. (2013). An analysis of security issues for cloud computing. Journal of internet services and applications, 4(1), 1–13.

[55] Istio (2023a). Architecture. https://istio.io/latest/docs/ops/deployment/architecture/ (accessed 07/2022).

[56] Istio (2023b). Bookinfo application (accessed 04/2023). https://istio.io/latest/docs/examples/bookinfo/.

[57] Istio (2023c). Github istio/istio (accessed 02/2020). https://github.com/istio/istio.

[58] Istio (2023d). Ingress gateways (accessed 07/2022). https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/.

[59] Istio (2023e). Istio / security faq (accessed 01/2020). https://istio.io/latest/about/faq/security/.

[60] Istio (2023f). Istio (accessed 01/2020). https://istio.io.

[61] Istio (2023g). Secure Gateways (accessed 04/2023). https://istio.io/latest/docs/tasks/traffic-management/ingress/secure-ingress/.

[62] Istio (2023h). Security (accessed 02/2020). https://istio.io/latest/docs/concepts/security/.

[63] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. IEEE Software, 35(3), 24–35.

[64] Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using docker technology. In SoutheastCon 2016 (pp. 1–5).: IEEE.

[65] Jewell, T. (2018). The exploding endpoint problem: Why everything must become an api (accessed 01/2022). https://thenewstack.io/the-exploding-endpoint-problem-why-everything-must-become-an-api/.

[66] Jian, Z. & Chen, L. (2017). A defense method against docker escape attack. In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy (pp. 142–146).

[67] Joy, G. (2017). Distil Networks securely stores and manages all their secrets with Vault and Consul (accessed 02/2020). https://www.hashicorp.com/blog/distil-networks-securely-stores-and-manages-all-their-secrets-with-vault-and-consul/.

[68] Krishna, H. (2021). 5 microservices examples: Amazon, netflix, uber, spotify, and etsy (accessed 07/2022). https://www.sayonetech.com/blog/5-microservices-examples-amazon-netflix-uber-spotify-and-etsy/.

[69] Kubernetes (2023a). Kubernetes - Production-Grade Container Orchestration (accessed 01/2020). https://kubernetes.io/.

[70] Kubernetes (2023b). Kubernetes components (accessed 07/2022). https://kubernetes.io/docs/concepts/overview/components/.

[71] Kubernetes (2023c). Kubernetes Docs (accessed 02/2020). https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[72] Kubernetes (2023d). Kubernetes Pods (accessed 02/2020). https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/#kubernetes-pods.

[73] Kuma (2023a). Kuma (accessed 01/2022). https://kuma.io/.

[74] Kuma (2023b). Mutual tls | kuma (accessed 01/2022). https://kuma.io/docs/1.2.x/policies/mutual-tls/#usage-of-provided-ca.

[75] Lardinois, F. (2022). Google donates the istio service mesh to the cloud native computing foundation (accessed 07/2022). https://techcrunch.com/2022/04/25/google-donates-the-istio-service-mesh-to-the-cloud-native-computing-foundation/.

[76] Li, X., Chen, Y., Lin, Z., Wang, X., & Chen, J. H. (2021). Automatic policy generation for inter-service access control of microservices. In 30th USENIX Security Symposium, USENIX Security 21).

[77] Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., & Zhou, Q. (2018). A Measurement Study on Linux Container Security: Attacks and Countermeasures. In Proceedings of the 34th Annual Computer Security Applications Conference (pp. 418–429).

[78] Linkerd (2023a). Automatic mTLS (accessed 02/2020). https://linkerd.io/2.12/features/automatic-mtls/index.html.

[79] Linkerd (2023b). Github linkerd/linkerd2 (accessed 01/2022). https://github.com/linkerd/linkerd2.

[80] Liquibase (2023). 9 DevOps Metrics Teams should Track (accessed 04/2023. https://www.liquibase.com/resources/ebooks/devops-metrics.

[81] Locust.io (2023). Locust - a modern load testing framework (accessed 04/2023). https://locust.io/.

[82] Ltd, R. (2018). The mikrotik routeros-based botnet.

[83] Mauro, T. (2015). Adopting microservices at netflix: Lessons for architectural design (accessed 01/2022). https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[84] Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In 2017 IEEE International Conference on Web Services (ICWS) (pp. 524–531).: IEEE.

[85] Mirza, M. (2022). Protect kubernetes workloads from apache log4j vulnerabilities (accessed 03/2023). https://aws.amazon.com/blogs/containers/protect-kubernetes-workloads-from-apache-log4j-vulnerabilities/.

[86] Mishra, A. & McCarron, P. (2021). Service mesh at global scale (accessed 01/2022). https://www.hashicorp.com/cgsb.

[87] Morabito, R., Kjällman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: a performance comparison. In 2015 IEEE International Conference on cloud engineering (pp. 386–393).: IEEE.

[88] Morgan, W. (2021). Announcing linkerd's graduation (accessed 07/2022). https://linkerd.io/2021/07/28/announcing-cncf-graduation/.

[89] Nam, J., Lee, S., Seo, H., Porras, P., Yegneswaran, V., & Shin, S. (2020). Bastion: A security enforcement network stack for container networks. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (pp. 81–95).

[90] NGINX (2015). Service discovery in a microservices architecture (accessed

01/2020). https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

[91] nicholasjackson (2023). fake-service – simple service for testing upstream service communications (accessed 04/2023). https://github.com/nicholasjackson/fake-service.

[92] Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 23–43).: Springer.

[93] O'Keefe, M. (2019). Welcome to the service mesh era: Introducing a new istio blog post series (accessed 04/2023). https://cloud.google.com/blog/products/networking/welcome-to-the-service-mesh-era-introducing-a-new-istio-blog-post-series.

[94] Ongaro, D. & Ousterhout, J. (2015). The raft consensus algorithm. Lecture Notes CS, 190, 2022.

[95] Open Policy Agent (2023). Open policy agent (accessed 04/2023). https://www.openpolicyagent.org/.

[96] Open Service Mesh (2023a). Certificate management | open service mesh (accessed 01/2020). https://release-v0-11.docs.openservicemesh.io/docs/guides/certificates/.

[97] Open Service Mesh (2023b). Open service mesh documentation (accessed 01/2022). https://release-v1-2.docs.openservicemesh.io/.

[98] Open Service Mesh (2023c). Open service mesh (osm) (accessed 01/2021). https://openservicemesh.io/.

[99] Pahl, C. (2015). Containerization and the paas cloud. IEEE Cloud Computing, 2(3), 24–31.

[100] Pahl, C. & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study:. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (pp. 137–146).: SCITEPRESS - Science and and Technology Publications.

[101] Pauley, E., Sheatsley, R., Hoak, B., Burke, Q., Beugin, Y., & McDaniel, P. (2022). Measuring and mitigating the risk of ip reuse on public clouds. arXiv preprint arXiv:2204.05122.

[102] Perforce Software, Inc. (2023). Puppet infrastructure & it automation at scale (accessed 04/2023). https://www.puppet.com/.

[103] Platform, G. C. (2023). Online Boutique (accessed 12/2022). https://github.com/GoogleCloudPlatform/microservices-demo.

[104] Progress Software Corporation (2023). Chef software devops automation solutions (accessed 04/2023). https://www.chef.io/.

[105] Prometheus (2023). Prometheus - monitoring system & time series database (accessed 07/2022). https://prometheus.io/.

[106] Quaker, D. (2023). Amazon stats: Growth, sales, and more (accessed 04/2023). https://sell.amazon.com/blog/amazon-stats.

[107] Rahman, A., Mahdavi-Hezaveh, R., & Williams, L. (2019). A systematic mapping study of infrastructure as code research. Information and Software Technology, 108, 65–77.

[108] Rastogi, V., Davidson, D., De Carli, L., Jha, S., & McDaniel, P. (2017). Cimplifier: Automatically Debloating Containers. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 476–486).

[109] Red Hat, Inc. (2023). Ansible is Simple IT Automation (accessed 04/2023). https://www.ansible.com.

[110] Richardson, C. (2015). Introduction to Microservices (accessed 02/2020). https://www.nginx.com/blog/introduction-to-microservices/.

[111] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture. Technical report, National Institute of Standards and Technology.

[112] Ruth, J.-P. S. (2022). What happens if microservices vanish – for better or for worse (accessed 01/2023). https://www.informationweek.com/strategic-cio/what-happens-if-microservices-vanish-for-better-or-for-worse.

[113] Sakic, E. & Kellerer, W. (2018). Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane. IEEE Transactions on Network and Service Management, 15(1), 304–318.

[114] Salehie, M. & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. ACM transactions on autonomous and adaptive systems (TAAS), 4(2), 1–42.

[115] Saltzer, J. H. (1974). Protection and the control of information sharing in multics. Communications of the ACM.

[116] Sankaran, A., Datta, P., & Bates, A. (2020). Workflow integration alleviates identity and access management in serverless computing. In Annual Computer Security Applications Conference (pp. 496–509).

[117] Services, A. W. (2023a). Amazon ec2 t2 instances (accessed 10/2020). https://aws.amazon.com/ec2/instance-types/t2/.

[118] Services, A. W. (2023b). DevOps (accessed 01/2020). https://aws.amazon.com/marketplace/solutions/devops.

[119] Shu, R., Gu, X., & Enck, W. (2017). A Study of Security Vulnerabilities on Docker Hub. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (pp. 269–280). New York, NY, USA: Association for Computing Machinery.

[120] Smith, F. & Garrett, O. (2018). What is a Service Mesh (accessed 12/2020). https://www.nginx.com/blog/what-is-a-service-mesh/.

[121] Smith, J. E. & Nair, R. (2005). The architecture of virtual machines. Computer, 38(5), 32–38.

[122] Sukhwani, H., Martínez, J. M., Chang, X., Trivedi, K. S., & Rindos, A. (2017). Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS) (pp. 253–255).: IEEE.

[123] Sun, W., Zhang, K., Chen, S.-K., Zhang, X., & Liang, H. (2007). Software as a service: An integration perspective. In Service-Oriented Computing–ICSOC 2007: Fifth International Conference, Vienna, Austria, September 17-20, 2007. Proceedings 5 (pp. 558–569).: Springer.

[124] Sun, Y., Nanda, S., & Jaeger, T. (2015). Security-as-a-Service for Microservices-Based Cloud Applications. In 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 50–57).: IEEE.

[125] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018: SciTePress.

[126] Tak, B.-C., Isci, C., Duri, S. S., Bila, N., Nadgowda, S., & Doran, J. (2017). Understanding security implications of using containers in the cloud. In USENIX annual technical conference (pp. 313–319).

[127] The Linux Foundation (2022). Cloud native computing foundation – graduated and incubated projects (accessed 07/2022). `https://www.cncf.io/projects/`.

[128] The MITRE Corporation (2021). Cve-2021-44228 (accessed 01/2023). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228`.

[129] Thomson, R. (2018). LogicMonitor Uses Terraform, Packer & Consul for Disaster Recovery Environments (accessed 02/2020). `https://www.hashicorp.com/blog/logic-monitor-uses-terraform-packer-and-consul-for/`.

[130] Thones, J. (2015). Microservices. IEEE Software, 32(1), 116–116.

[131] Tunggal, A. T. (2021). The cost of downtime at the world's biggest online retailer (accessed 07/2022). https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer.

[132] Uberti, D., Rundle, J., & Stupp, C. (2021). The log4j vulnerability: Millions of attempts made per hour to exploit software flaw (accessed 01/2023). https://www.wsj.com/articles/what-is-the-log4j-vulnerability-11639446180.

[133] Watson, C., Emmons, S., & Gregg, B. (2015). A microscope on microservices (accessed 01/2021). https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4.

[134] Watson, M. (2017). 15 Metrics for DevOps Success (accessed 01/2020). https://stackify.com/15-metrics-for-devops-success/.

[135] Wolf, N. (2016). Ddos attack that disrupted internet was largest of its kind in history, experts say.

[136] Xiao, Z., Song, W., & Chen, Q. (2012). Dynamic resource allocation using virtual machines for cloud computing environment. IEEE transactions on parallel and distributed systems, 24(6), 1107–1117.

[137] Yarygina, T. & Bagge, A. H. (2018). Overcoming Security Challenges in Microservice Architectures. In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE) (pp. 11–20).: IEEE.

[138] Zheng, Z., Xie, S., Dai, H.-N., Chen, X., & Wang, H. (2018). Blockchain challenges and opportunities: A survey. International journal of web and grid services, 14(4), 352–375.