Anvil: A Novel, Zero-Downtime Service Mesh for Automatic Security Artifact Rotations in Microservice Deployments

Abstract

The advent of cloud computing and novel software architectures has shifted the focus of enterprise deployments to flexible microservice structures. With these unique deployments, new challenges in managing them have arisen. Service meshes fulfill these new gaps in functionality, however, shortcomings in their security design create serious issues over the lifetime of the managed systems. Attempts to address these issues have resulted in high costs for system performance and management, and inherent system downtime.

This paper presents *Anvil*, a flexible, dynamic, and secure service mesh that does not compromise on the long-term health and maintainability of the system it secures. Through zero-downtime security artifact rotation, Anvil combines the main benefits from state-of-art service mesh designs to offer a novel, long-lived service mesh. With more than 114,600 unique CPU/RAM data points of operational Anvil, Consul, Kubernetes, and Istio deployments in different evaluation environments, our results show that automated security artifact rotations, zero-downtime, and platform-agnostic deployment functionality can be achieved in a reliable and consistent way.

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* [4, 46, 48]. Modern software engineering has trended towards a microservice paradigm and away from the traditional, monolithic structure of software applications [18, 29, 48]. Through separation of concerns and segregation of software components, the size of deployments has grown dramatically and the issue of *microservice explosion* has emerged.

Service meshes [13], part of the *DevOps* [1] approach, have filled the role of managing this explosion in the number of microservices and we see increasing adoption rates of service meshes managing microservice architectures in real-world systems [5, 9–12, 31, 56]. Service meshes now provide infrastructure management as well as network coordination. 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 requiring the creation of connections across network boundaries. Due to this, service meshes now bear the responsibility of coordinating and securing these connections to uphold the necessary security guarantees.

However, despite being charged with responsibilities of network security, current service mesh designs fail to meet vital needs. From systems that prevent rotation of security artifacts such as cryptographic certificates [22], to platform-dependent service meshes that require high-cost, preexisting platforms in order to function correctly [24, 28, 35, 43], the current state of service mesh tools lack the means to effectively ensure the security of the systems they have been designed for [19]. For example, in case security artifacts (e.g., cryptographic keys) at the service mesh-level need to be replaced due to a potential compromise, the entire microservice deployment managed by a current state-of-art service mesh would be severely affected. Significant downtime and performance costs would be incurred, or a full redeployment may be warranted.

While competing designs and varying architectures are present within the domain (e.g., Consul [20], Istio [28], Linkerd [24], OSM [43], Kuma [35], etc.), *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 the platform itself [19]. This creates potential opportunities for adversaries to gain and maintain privilege in the growing number of distributed microservice deployments.

Our work, *Anvil*, directly addresses these concerns with modern service mesh design and provides key benefits that more closely meet the needs and challenges of microservice architectures without sacrificing security for performance. Anvil is a novel service mesh prototype that alleviates design concerns in state-of-art service meshes while also balancing the security-performance tradeoff present within the DevOps space. Anvil is, to the best of our knowledge, the first service mesh that provides automated, synchronous, zero-downtime

rotation of *all* security artifacts utilized within a service mesh in a flexible, low-cost fashion. 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. By building upon modern service mesh design without sacrificing the feature set, but with a particular focus towards long-lived systems, Anvil more closely meets the flexible and dynamic needs of the DevOps space, while also providing strong security guarantees to system administrators.

Through a security analysis and a wide range of experiments in different environments, we evaluate Anvil and demonstrate the benefits of its unique design. Leveraging testbed environments on local, controlled infrastructure, as well as Amazon Web Service's (AWS) [52] Elastic Compute Cloud (EC2) [51] resources, we conducted experimental trials to collect CPU and RAM utilization as well as system downtime metrics. From these metrics, we show that Anvil performs better than Consul with respect to system downtime incurred and Anvil performs overall better than all platform-dependent service meshes (Istio, Linkerd, OSM, Kuma, etc.) that require an underlying Kubernetes [32] platform, with respect to flexibility and in terms of CPU/RAM utilization.

Due to the diversity in design and implementation of current service meshes (Consul, Istio, Linkerd, OSM, Kuma), a direct lateral performance comparison of Anvil to these tools is difficult and often not representative. Because of this, we analyze Anvil in conjunction to the state-of-art tools according to the traits that Anvil shares with each. Using 29,400 unique CPU and RAM data points collected in a controlled testbed environment, we show that security artifact rotations within Anvil do not incur significant processing costs (up to 1% utilization when comparing median values across experiment trials). Additionally, we show that Anvil's zero-downtime rotation capability, if implemented alongside current Consul design creates inherent system downtime, which is undesirable in microservice deployments. Lastly, we analyze the platform costs of Kubernetes and Istio. Despite bringing Kubernetes to its most lightweight version, 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 with regards to platform-agnostic deployments, as well as the long-term health of a service mesh by automatically rotating certificates, rather than requiring manual system administrator intervention [25, 27, 36, 41].

By alleviating and mitigating these shortcomings within the service mesh domain, Anvil embodies a service mesh design that more closely fills the needs of microservice environments. Through the key features of zero-downtime operation, automated security artifact rotations, and platform-agnostic deployment options, 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 existing service mesh tools and identify security shortcomings and limitations in design that we then accommodate within our proposed solution, Anvil.
- We present Anvil, a proof-of-concept, secure-by-design service mesh platform that provides zero-downtime, automated rotation of security artifacts.
- We perform an experimental evaluation comparing Anvil with the current state-of-art in service meshes, showing the benefit of zero-downtime rotations and low performance cost of the Anvil platform relative to alternative tools.

2 Background and Security Shortcomings

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.

Microservice Architectures: Traditionally, software has been created, packaged and deployed as a singular, whole product. This practice has been referred to as the *monolithic architecture*. However, while this practice is simple to understand and straightforward to build and deploy, the scalability, manageability and speed at which such a system can be deployed is highly limited. Due to this, software has been separated into components or modules that are called *microservices*. 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.

The shift towards microservices has enabled an increase in the speed, development and availability of online services, pushing software engineering to adopt this novel, architectural design referred to as microservice architectures. 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. With extremely fast deployment times and increased efficiency by leveraging shared resources, containerization has been a key factor in the adoption and growth of the microservice paradigm within modern software engineering. However, as enterprise software expands in size and the quantity of microservices being deployed grows, the management and coordination of these services is increasingly difficult. Service meshes have emerged as the latest solution, in a series of developments, to address the issue of microservice explosion in large-scale software.

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 [30, 40].

The current production-ready, state-of-art service mesh tools are HashiCorp's Consul, Linkerd, Istio, and Kuma. [20, 24,28,35] (OSM is, at the moment, in a pre-release stage. [43]) 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 platform-dependent 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 platform-dependent service meshes (Istio, Linkerd, OSM, Kuma, etc.) are unable to provide any of the purported functionality to a deployment, meaning that expertise with Kubernetes is an inherent cost in utilizing these tools. Within Anvil and Consul, rather than depending upon an orchestration platform, like Kubernetes, a membership "quorum" is established that provides similar functionality.

Service Meshes Feature Set: Despite differences in deployment options and underlying requirements, there is a common set of features that service meshes aim to provide to system administrators in managing the microservice deployments. Among the features offered across service meshes, *service discovery and management* [53], *dynamic load balancing* [53], *secure network communications* [53], and *observability* [53] are key functions that all modern service meshes claim.

Service discovery and management are handled differently within the two competing models (platform-dependent and platform-independent), however, this feature can be summarized as the ability for the deployed microservices to locate and create connections to other microservices upon which they depend. Additionally, service meshes provide various capabilities for configuring and modifying metadata about deployed microservices, such as the service name, priority in routing, and other features such as blue/green or A/B testing.

Leveraging the service discovery and management functionality, dynamic load balancing is possible due to the service mesh making intelligent decisions upon the routing scheme and current load of each of the microservices that are deployed. For example, microservice *A* may be deployed and a duplicate created to provide horizontal scaling. A service mesh, under heavy load can choose to route requests to either of the microservices in order to maintain a high degree of availability and keep request latency low. Together with the traditional *intra*-service connections turned to *inter*-service connections and requiring connections across network boundaries, service meshes are responsible to uphold the necessary security guarantees. In this regard, the burden of network security falls to service meshes and is implemented similarly within the current state-of-art by utilizing service proxies that reside at the network boundary of a node, which may take the form of a virtual machine, container, Kubernetes pod, etc.

All modern service meshes leverage TLS connections when making service-to-service requests and often allow for mutual-TLS (mTLS) connections as well, in which both communicating parties authenticate and verify one other before accepting data transmission. This service proxy element, while necessary for network communication security, provides additional insight into the deployed microservices and acts as a mechanism for observability within the service mesh by providing access to metrics such as requests per second, error rate, latency, etc. Despite these key features and functionality, service meshes are still an emerging and evolving area of development with key limitations and shortcomings that must be addressed. Of particular concern is the conflict between current service mesh design and high-level goals of DevOps and the cloud computing domain.

2.1 Security Shortcomings in Service Meshes

In the current state-of-art for service mesh technologies, there is a mismatch between security design and the goals of microservice architectures managed by service meshes. With the goals of flexibility, speed, and maintainability at the forefront of microservice architectures and the general trends of DevOps, the current static security mechanisms embedded within service meshes directly conflict with these goals.

Static Security Conflicts: Within the Consul service mesh, shared, lifetime encryption keys and single-point security artifact generation conflict with the goals of flexible and dynamic security. Shared, lifetime encryption keys are used to secure UDP connections made between nodes in a Consul deployment and these keys are *required* to be shared and *must* remain static throughout the deployment's lifetime. Further, in order to establish access control within the Consul service mesh, all access control policies must be ingested and processed on a single node, most often the Consul "leader" node, within the cluster and then distributed to all other members before service requests may be authorized. This aspect conflicts with the goal of speed in that a single point of processing is required and may not be distributed amongst the other quorum members of the cluster due to limitations in security design.

Platform Dependence and Lock-In: Extending these shortcomings, platform-dependent service meshes, such as Istio, Linkerd, OSM, Kuma, etc. have inherent conflicts with the goal of flexibility. By locking system administrators into a single platform, namely Kubernetes, the ability for a service mesh to adapt to the environment in which it is needed is extremely limited. System administrators are thus required to understand and have expertise in managing and configuring Kubernetes simply to establish a working cluster for operations. Installing and configuring Kubernetes to begin with can

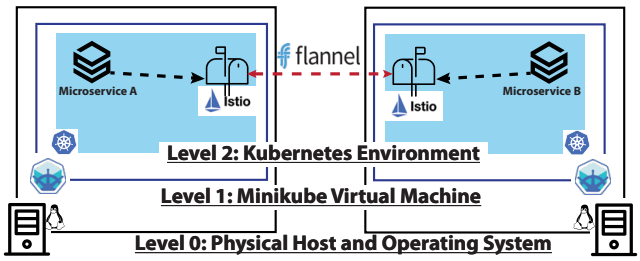

Figure 1: Complexity of Kubernetes-based Service Mesh Deployment – Deploying a Kubernetes environment consists of many different software components and layers of dependent applications. An example Kubernetes deployment is shown, highlighting each of the dependent layers within a small-scale Kubernetes environment.

be extremely challenging and complex as noted by a Cloud Native Computing Foundation (CNCF) survey in 2020 [6, 7]. Kubernetes is a CNCF sponsored project and according to the more than 1300 respondents to the survey, 41% found complexity a leading issue in adoption of their organizations to using and deploying containers. Figure 1 illustrates this complexity and layered aspect of Kubernetes deployments. These sentiments are echoed in a separate study conducted by Circonus that surveyed 200 Kubernetes operators and found that finding administrators with expertise in Kubernetes was a top challenge in implementing and managing their Kubernetes deployments [44]. Additionally, security concerns and lack of training were significant issues reported as well [23], showing a rift between knowledge and expertise held by real-world administrators and the *required* knowledge and expertise to properly leverage Kubernetes effectively and safely.

Overall, the complexity introduced by Kubernetes is a noted issue in a range of surveys and online articles, showing that a key weakness and shortcoming in Kubernetes as a state-of-art platform is its usability and degree of complexity [6,7,23,44,57]. Anvil, in contrast, creates the control plane and management networking of the service mesh at runtime and in a secure manner with certificates that are automatically rotated and refreshed. Rather than imposing significant expertise in multiple tools and technologies, Anvil provides a flexible, simplified deployment option for administrators that desire a secure, long-lived service mesh.

Overcoming Design Limitations with Anvil: Due to these shortcomings in current service mesh design, we propose and implement Anvil as a proof-of-concept service mesh that combines the benefits of the two leading designs in the current state-of-art service mesh domain. Additionally, Anvil incorporates notable features not present in modern service meshes. We believe that Anvil embodies a secure-by-design service mesh and can serve as a model for future improvements in service mesh design and research. Anvil balances the performance cost between the lightweight platform-independent service mesh, Consul, and the feature-rich, platform-dependent service meshes Istio, Linkerd, OSM, Kuma, etc.. Anvil provides a range of features such as artifact rotation and automatic mutual-TLS connections found in platform-dependent

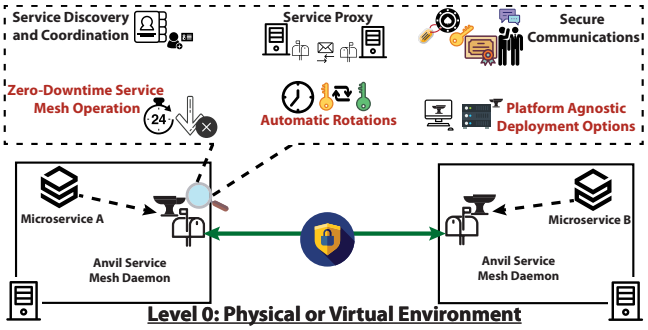

Figure 2: Anvil Deployment Structure and Features – Anvil implements the core functionality of state-of-art service meshes. These functions are service discovery and coordination, traffic proxying for hosted microservices, and communication security. Anvil also extends this base functionality and includes novel features such as zero-downtime operation, automatic security artifact rotations, in a platform-agnostic structure. Comparatively, platform-dependent service meshes *must* be deployed with a Kubernetes platform.

service meshes without the need for an underlying, full-fledged orchestration platform like Kubernetes. In this way, Anvil fills the much needed middle-ground between platform-independent (Consul) and platform-dependent service meshes (Istio, Linkerd, OSM, Kuma, etc.) by combining the benefits of each design, but avoiding much of the shortcomings and costs that plague current offerings. Figure 2 provides an overview of the functions and features present within Anvil.

3 Anvil Design

By reviewing the current state-of-art in service meshes, Anvil was designed to provide the core functions of service meshes with an emphasis on security-by-design. As part of this effort, Anvil implements the core design principles of automated artifact rotations, zero-downtime, and platform-independence.

3.1 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.

3.2 Platform Independence and Modularity

Similar to the Consul service mesh, Anvil is platform-independent, meaning that it does not require an underlying Kubernetes infrastructure or alternative container orchestration platform in order to utilize its functionality. Through this independence, Anvil may be deployed in a range of environments from cloud computing environments to physical devices such as Raspberry Pis or desktop computers. Additionally, by being platform independent, Anvil avoids much of the overhead that is required from an underlying container orchestration platform such as Kubernetes. The cost of underlying platforms is discussed later within Section 5. Further, orchestration platforms require additional expertise in order to operate and configure correctly. Anvil avoids this cost as well by providing its flexible means of deployment.

For ease of development and extendability of the system overall, Anvil was created in a modular way. Intended as a platform for future research and development, many of the components of Anvil can easily integrate with other designs and systems. For example, Raft is currently utilized as the consensus algorithm to coordinate and control artifact rotations within the cluster, as well as maintain the list of active access control tokens and policies among the services deployed within an Anvil cluster. However, Anvil is not dependent upon Raft as part of its design, alternative consensus algorithms such as Paxos [37], or instead of a consensus algorithm, a shared log file may be used to keep the roster of active access control policies and tokens within the cluster. In this way, Anvil may be used as a simplified service mesh for the purpose of examining the performance and security of different service mesh design decisions.

3.3 Automated Artifact Rotations

A key contribution that Anvil makes to the community is the capability and demonstrated feasibility of automated security artifact rotations within the service mesh. Current state-of-art service meshes either provide no means of circulating fresh security artifacts (Consul), or require complex orchestration platforms (Kubernetes) to facilitate this refreshment, as is the case of platform-dependent service meshes like Istio and Linkerd. However, Anvil bridges these designs and incorporates automated artifact rotations as part of the overall service mesh design. Through this design, Anvil provides the security of regularly refreshed security artifacts within the cluster, allowing for long-lived containers or virtual machines without the potential for exposure of security artifacts over long periods of time. Further, by performing the rotations within the service mesh itself, rotations are easily coordinated and conducted in a synchronized way resulting in very little overhead and no interruptions imposed upon service mesh operation.

3.4 Zero-Downtime Rotations

A primary shortcoming in the Consul service mesh as noted in Section 2 is the inherent downtime cost associated with refreshing the security artifacts within a Consul deployment. As such, it was a key design decision facilitate security artifact rotations in Anvil without downtime costs. Currently, Anvil avoids downtime costs by performing live updates of configuration files associated with the service proxy. The Anvil service proxy is implemented as a lightweight web server that provides routing and proxy functionality to the hosted microservices. In this way, the key configuration details that the Anvil service proxy must manage for a security artifact rotation is the TLS certificate-key pair associated with the running web server. When a new set of security artifacts is sent to an Anvil proxy, a new web server instance is created and the fresh artifacts are associated with this process. Next, the old server is stopped and any ongoing connections are

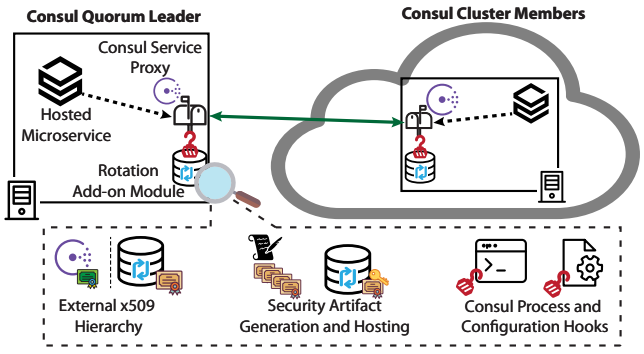

Figure 3: Consul Rotation Add-On – "Best-effort" Consul rotation add-on abstraction. Deployed within AWS and using a separate x509 certificate hierarchy from the Consul TLS structure, the rotation add-on provides generation and distribution of security artifacts without Consul code modifications.

finalized. Lastly, the new instance of the web server becomes the primary point of contact for the service proxy and any stale connections are discarded along with old security artifacts. By coordinating these steps among nodes, Anvil is able to actively distribute fresh security artifacts to all cluster members and coordinate a synchronized rotation of service proxy instances to a fresh set of artifacts. All new network connections are then made using the fresh security artifacts.

4 Implementation

Throughout development and prototyping of Anvil, container technology was heavily utilized. However, Anvil, similar to Consul, is built with the capability to be deployed in a range of environments and platforms. The Anvil application is written using Golang [16] and is compiled into an executable binary. To execute Anvil properly YAML-format configuration files are utilized to supply the necessary parameters for service mesh cluster setup and operations.

4.1 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 [8]. 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) [51] infrastructure to deploy and measure the downtime associated with the Consul service mesh for security artifact rotations.

Anvil 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 2, quorum members 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 [45], 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.

Consul Setup: To demonstrate the effects of a "best-effort" attempt 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 [51] providing 8 vCPUs with 32 GB of RAM and 2 vCPUs and 4 GB of RAM, respectively. Our best-effort system for security artifact rotations 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 provides an abstract view of the rotation add-on. 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 and Kubernetes Setup: To create an accurate comparison of the performance of Anvil against Kubernetes and Kubernetes-dependent service mesh architectures, we design and implement an example scenario within our testbed environment that attempts to mimic the traffic behavior of both Anvil and Consul quorums as accurately as possible. Therefore, we create an example application that generates network traffic patterns that match what we have observed within Anvil and Consul quorum members and deploy it via Docker containers within a Kubernetes environment. 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 any service mesh deployed on top of an existing Kubernetes platform. Specifically, we aim to install and configure an extremely lightweight Kubernetes 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 upon each of the

Service Mesh Tool	Security Artifact Rotations	Automated Rotations	Zero-Downtime Rotations	Flexible Deployments	Attack Window (Data Plane)	Attack Window (Control Plane)
Anvil	✓	✓	✓	✓	5 minutes*	5 minutes*
Consul	⊗	⊗	⊗	✓	∞	∞
Istio	✓	◐	✓	⊗	24 hours [27]	≥1 year [27]
Linkerd	✓	◐	✓	⊗	24 hours [25]	1 year [25]
OSM	✓	◐	✓	⊗	24 hours [41]	1 year [41]
Kuma	✓	◐	✓	⊗	24 days [36]	≥1 year [36]

Table 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. **5 minutes is the current default time for rotations within Anvil, however, it is customizable and may be reconfigured as needed for environment or domain needs.*

Kubernetes nodes that make up our 5 VM cluster. Extending the experiments, we we make use of the Istio service mesh as a representative of the platform-dependent service mesh design. Istio is currently the most widely used service mesh based upon GitHub metrics [21, 26, 39, 42].

5 Evaluation

Anvil is intended to fill a gap in the current state-of-art in service meshes by combining the flexibility and performance present within the Consul service mesh with the security features present within platform-dependent service meshes such as Istio and Linkerd. Due to this, 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 four aspects of evaluation to place Anvil within the correct context of comparison for each point of consideration:

- First, we present a metric referred to as the "attack window" of a given system and compare Anvil to all state-of-art service meshes with respect to the attack window. Also, we consider how Anvil's rotations affect the overall system entropy over time relative to a service mesh without rotation capabilities, such as Consul.
- Next, we examine the Anvil proof-of-concept and assess the performance cost associated with Anvil's rotation capabilities relative to baseline Anvil without rotations.
- We examine the associated cost of implementing a "best-effort" security artifact rotation system in Consul due to its lack of rotation capabilities. We evaluate Consul with the rotation system according to the system downtime incurred.
- Lastly, we compare the associated CPU and RAM utilization costs of platform-dependent service meshes. We use the Kubernetes-dependent Istio service mesh as a representative example of this design to assess performance and operational costs relative to Anvil.

5.1 Attack Window

Due to the transformative nature of security artifact rotations in Anvil, it is important to consider the security of a service mesh deployment 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 [2], 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.

$$W = t + T_r \tag{1}$$

$$T_r = T_{gen} + T_{dist} + T_{ch} \tag{2}$$

In Equation 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 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

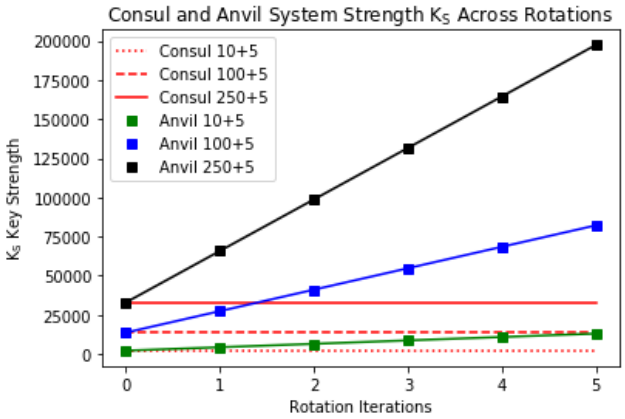

Figure 4: Cumulative System Key Strength – Using NIST's Recommendations for Key Management [3], the cumulative system security strength K_S is plotted between Anvil and Consul deployments of various client cluster sizes with a consistent, 5 member quorum. Anvil's cumulative security strength grows linearly over time due to security artifact refreshment while Consul's remains constant due to lack of rotation.

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 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 five minutes, resulting in over 250 rotations in a 24-hour period. This configuration may be altered according to the application needs and the risk tolerance of a system.

Table 1 presents the attack windows and provided security features of all state-of-art service meshes relative to Anvil. According to our definition of an attack window, the current length of exposure in alternative service meshes is alarming. For example, platform-dependent service meshes all rotate TLS certificates and keys regularly within the deployment for traffic that occurs between microservices every 24 hours. This traffic occurs within the "data plane" of the service mesh and represents microservice communications. However, all platform-dependent service meshes do *not* provide automatic rotations of TLS certificates used for cluster management. These certificates are used for cluster management and coordination of the service mesh elements themselves and are referred to as "control plane" communications. These certificates must be manually rotated by system administrators and additionally have lifetimes of upwards of 1 year. More alarming than extended certificate exposure in platform-dependent service meshes is the lack of rotation support entirely in the Consul service mesh. Further, the attack window of a Consul service mesh may be infinite, stopping only when a deployed TLS certificate expires and is no longer accepted by recipient service proxies. In contrast to both designs, Anvil intentionally limits the exposure of any given set of security artifacts within a service mesh deployment and automatically refreshes these artifacts to maintain a dynamic security posture. By leveraging regular artifact rotations and limiting an adversary to a small window of time in which their attacks may succeed, the degree of difficulty for a successful attack is raised and the usefulness of the stolen/leaked credentials is limited.

5.2 Key Strength and System Entropy

The security capabilities of Anvil include the flexibility to change cipher-suites and cryptographic schemes as needed during cluster setup. Due to this, Anvil can accommodate a range of risk levels that a given organization or administrator are willing to assume for their given workloads. For example, a low priority system that does not deal with sensitive information may have a very different risk tolerance than a system dealing with safety-critical data or with medical data. With these varying characteristics in mind, we present a formal definition of the key entropy present within Anvil with respect to cluster size and cipher suites utilized. With Anvil's current implementation, a shared AES256 bit UDP key is used to encrypt gossip communication within the cluster, each node is provided an ECC256 certificate and key pair for TLS encryption, and then every access control policy within the cluster is assigned a 64-character randomized string to attach to requests as a token for authorization. Overall, the key strength of an Anvil deployment can be measured as follows:

$$K_S = S_{udp} + (S_{tls} * n) + (S_{acl} * k) \tag{3}$$

Within Equation 3, K_S represents the overall key strength of a deployed Anvil cluster. S_{udp} represents the key strength of the shared, symmetric UDP key, while S_{tls} represents the key strength of a given cryptographic scheme for TLS encryption (i.e. - RSA2048, RSA4096, ECC256, etc.). The variable n represents the number of unique nodes deployed within the cluster and is multiplied with S_{tls} because each node requires a unique key-certificate pair. Lastly, the variable k represents the number of unique access control policies deployed within the cluster and is multiplied with S_{acl} because each access control policy requires a unique token value.

Figure 4 provides a graphical representation of Equation 3 and how the relative key strength of Anvil grows over time due to artifact rotations relative to Consul. Using the security strength values provided by NIST in [3] and the default configurations within Consul and Anvil, we can compute the system strength of various size deployments. Examining Figure 4, the key strength of the initial Consul and Anvil deployments increases with the size of the cluster, due to a larger number of unique certificates and keys deployed within the environment. However, as time passes for a given deployment, Consul remains static in terms of key strength while Anvil grows linearly due to consistent refreshment of security artifacts within the system. As previously mentioned with respect to the attack

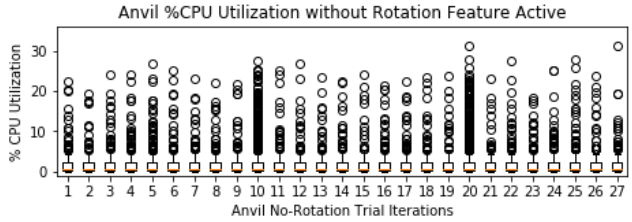

Figure 5: CPU Utilization in Anvil without Rotations – Box plots show CPU utilization across all experiment trials of Anvil when security artifact rotations were *not* active. Trials shown are those where full data collection was achieved. Median values are heavily focused near 0.5% utilization while outliers extend upwards of 30% utilization, but are highly infrequent.

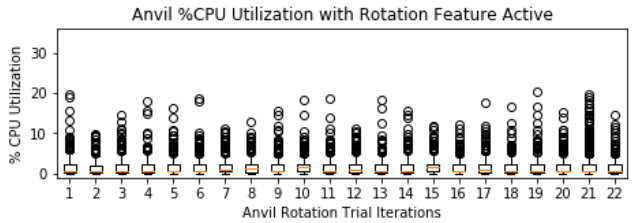

Figure 6: CPU Utilization in Anvil with Rotations – Box plots show CPU utilization across all experiment trials of Anvil when security artifact rotations were active. Trials shown are those where full data collection was achieved. Median values are heavily focused near 0.7% utilization while outliers extend upwards of 20% utilization, but are highly infrequent.

window metric, the consistent refreshment and transition from one set of security artifacts to another creates uncertainty for attackers and can significantly raise the difficulty and effort required to develop a meaningful attack against Anvil.

Utilizing the key strength definition provided in Equation 3, system administrators may wish to consider the security-performance tradeoff that is most applicable to their environment based upon the perceived risk of their domain. For example, mission-critical applications and deployments may opt for longer key lengths or more secure cryptographic schemes according to their domain requirements, while low-risk environments may desire more performant schemes and are willing to incur a greater risk. In addition to altering the cryptographic schemes utilized by Anvil, as described above, the frequency with which the cluster rotates security artifacts is equally important in mitigating risk of key exposure.

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 within an active Anvil service mesh cluster. As such, we leverage the experimental setup previously described in Section 4 to conduct our experiments and measurements. Additionally, we monitor active CPU and RAM utilization of nodes within the cluster by employing the *pidstat* [14] 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

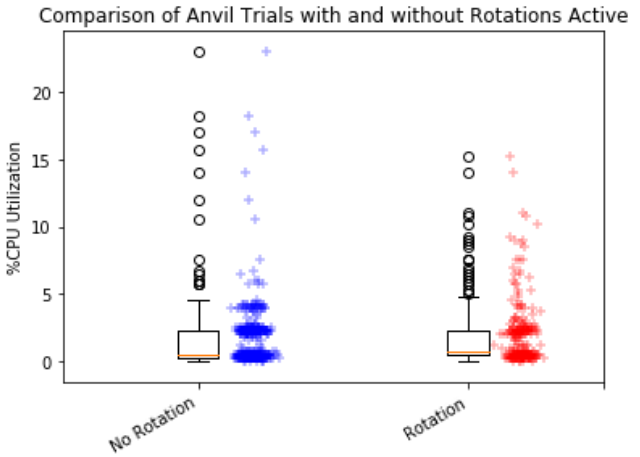

Figure 7: CPU Utilization in Anvil (Single Trial) – Box plots of CPU utilization in showing one experiment trial of Anvil with and without security artifact rotations active. Distribution of data points plotted alongside box plots to show density of data points and highlight outliers.

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 5 and 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 7 and plot the raw data points alongside each box plot to demonstrate the distribution of the data in a more concise fashion. Figure 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. Again, the frequency of rotations is configurable 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.

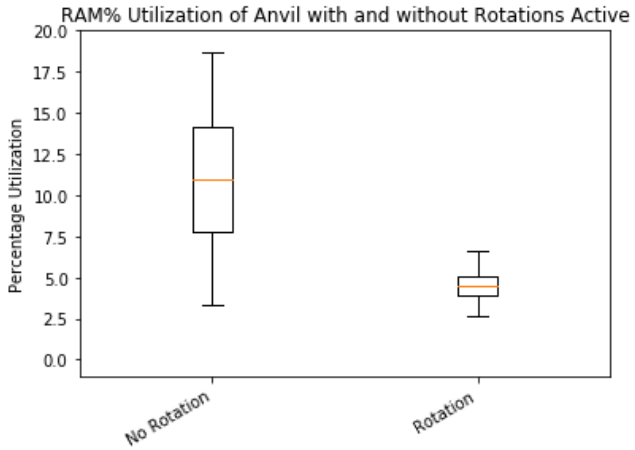

Figure 8: RAM Utilization in Anvil – Box plots of RAM utilization in Anvil with and without security artifact rotations active. After a careful examination of collected data that included use of Golang's profiling library *pprof* [15], we believe Golang's garbage collection responsible for this behavior and is triggered more often in the case of rotations.

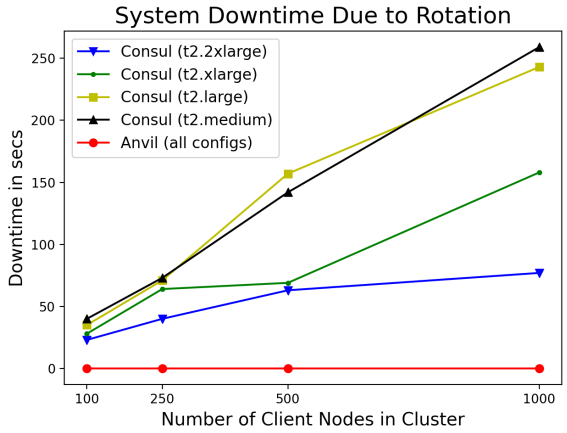

Figure 9: Downtime Costs of Rotations in Consul and Anvil – Each line represents a different hardware configuration within Amazon Web Services' EC2 cloud t2-series of virtual machines. The leader of Consul quorums is required to perform the generation and distribution tasks for cluster startup, this 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 Appendix A.

5.3 Consul 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 9 displays a best-effort attempt to automate the shutdown, artifact creation, artifact deployment and subsequent cluster recreation of such a scenario. As the figure 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 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 the best-effort rotation add-on 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.

5.4 Platform-Dependent Performance Costs

Mentioned previously in Section 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 platform-dependent 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 4 relating to our representative, platform-dependent service mesh Istio and Kubernetes. Specifically, after creating the network traffic application that mimics Anvil and Consul quorum network

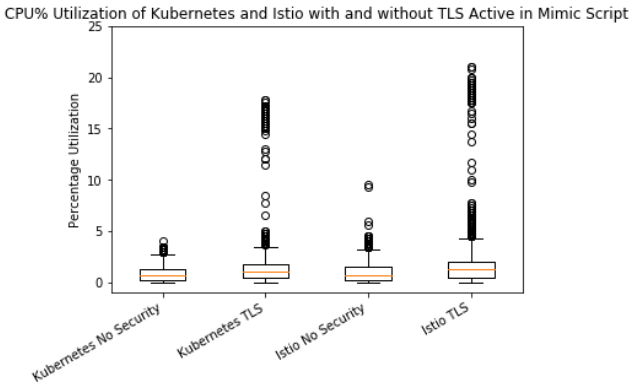

Figure 10: CPU Utilization of Platforms During Network Mimicry – Box plots illustrate the spread and locality of CPU utilization. Median values for each experiment are: Kubernetes without security features active (0.75%), Kubernetes with security features active (1.0%), Istio without security features active (0.75%), and Istio with security features active (1.25%).

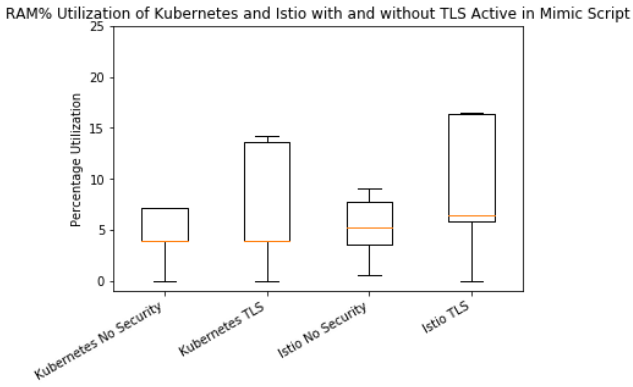

Figure 11: RAM Utilization of Platforms During Network Mimicry – Box plots illustrate the spread and locality of RAM utilization. Median values for each experiment are: Kubernetes without security features active (3.90%), Kubernetes with security features active (3.96%), Istio without security features active (5.20%), and Istio with security features active (6.46%).

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 4, 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. This setup was constructed to represent an extremely streamlined and lightweight implementation of Kubernetes.

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 Kubernetes and Istio. In each instance of experimentation, one container or pod was selected as the "leader" of the makeshift quorum and the network traffic mimicry application was executed. The processes associated with running this application and the underlying platform processes were measured for CPU and RAM utilization. After collection, the data points gathered were collated according to the experiment that was conducted and the type of node deployed. Figure 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 mimicry script, Kubernetes with TLS active, Istio without TLS active in the network mimicry script, and Istio with TLS active. The median CPU utilization percentages were 0.75%, 1.0%, 0.75%, and 1.25%, respectively. Next, we considered RAM utilization as another important factor to evaluate the cost of Kubernetes and Kubernetes-dependent service meshes.

Figure 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 3.90%, 3.96%, 5.20%, and 6.46%. When comparing the cost of each platform-dependent 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 platform-dependent service meshes, the testbed environment utilized is a near best-case scenario for Kubernetes and platform-dependent service meshes, yet still requires manual system administrator intervention throughout the lifetime of the deployment as noted in Table 1. Considering this, the the design-level benefits that Anvil provides over *all* state-of-art service meshes in conjunction with the comparable performance costs of Anvil to Kuberntes and platform-dependent service meshes is significant.

6 Discussion and Limitations

This work presents and demonstrates the design benefits of a service mesh with automatic, zero-downtime rotations. However, Anvil is still a prototype system intended to demonstrate the benefits of these initial design revisions. Due to this, there is still a range of future work opportunities and considerations.

Usability of Kubernetes: The Kubernetes ecosystem, while growing in popularity and adoption [7], still faces many issues with regards to usability and complexity. One source of the complexity within Kubernetes, and one of the leading challenges in the creation and configuration of Kubernetes, is the networking model employed. Kubernetes requires a third party Container Network Interface (CNI) in order to host a multi-node Kubernetes cluster [33]. From the CNCF 2020 survey, the most popular response with 38% of respondents to the question of how they are deploying Kubernetes, reported utilizing Minikube [34], which deploys a preconfigured Kubernetes instance in a virtual machine. Due to this, system administrators who deploy via this popular method must account for the understanding and expertise required to deploy

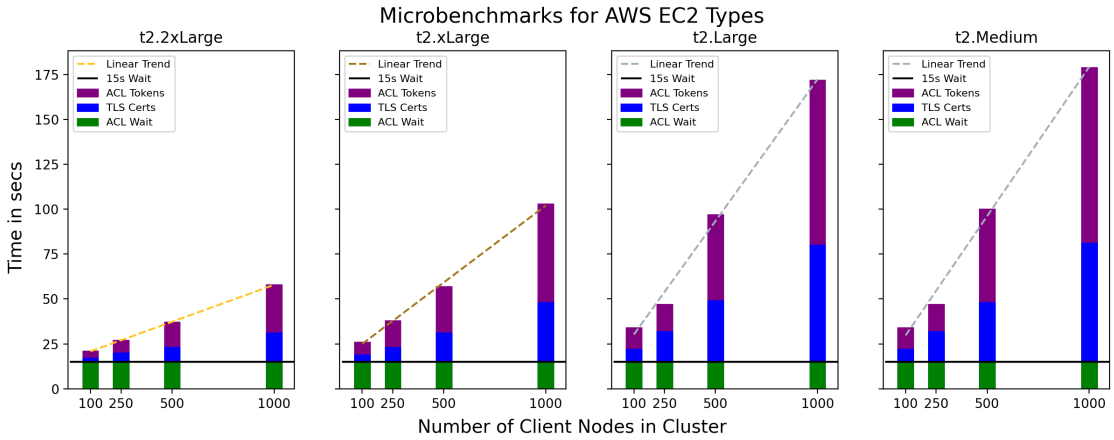

Figure 12: Microbenchmark Analysis of Consul Downtime Sources – Each figure represents a different hardware configuration within Amazon Web Services' EC2 cloud t2-series of virtual machines. The total downtime associated with performing a rotation within Consul is separated into the components in which 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 B.

and secure a third-party CNI, as well as account for the overhead to performance that installing Kubernetes within virtual machines may have on their deployed systems. As part of our evaluation in Section 5 we deploy Kubernetes directly within the operating system to avoid any platform or hypervisor overhead associated with Minikube and to create a near best-case deployment scenario. Aside from required expertise in Kubernetes to simply configure and install a Kubernetes platform, administrators have significant overhead and required expertise to properly configure and secure a service mesh system that gets deployed within Kubernetes. Due to these factors, many system administrators opt to utilize Kubernetes through cloud-provider managed systems [7]. However, for certain workloads, on-premise or local deployments may be required, motivating a need for simplified and more usable solutions.

Scalable Service Mesh Security: While our performance comparison experimentation can be regarded as small-scale, the proposed design does not impede scalability and the results are indicative of wider issues present within state-of-art service meshes. The shortcomings noted and highlighted as part of this study are representative to those present in operational environments. As illustrated by the downtime costs associated with the Consul service mesh, the security issues presented in this work scale with the size of the service mesh deployments, exacerbating the impact and costs of these shortcomings. Anvil takes these shortcomings as considerations to design choices and alleviates many of the issues and challenges present in the service mesh domain. As a platform for research and experimentation, we envision Anvil as a tool for testing novel design decisions in service meshes or applications that make use of a service mesh infrastructure.

Consensus Algorithms: Throughout development and testing, it was discovered that the overall structure of the Raft consensus algorithm was causing high-levels of network traffic within Anvil. Specifically, when security was enabled within Anvil, the CPU and RAM utilization of the system rose dramatically. After investigation, the cause was discovered to be the implementation of Raft consensus and the high numbers of TLS connections so that "heartbeat" messages can be passed between quorum members securely. Raft was adopted in Anvil to enable a Consul-specific quorum structure which enables 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.

7 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 containers and platforms 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.* [47] evaluate an automation system for decomposing a monolithic software deployment into a collection of collaborating mi-

croservices, the purpose of which, to better adhere to the principle of least privilege [50]. 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*. [58] 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*. [55]. 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*. [38] 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. By performing this extraction, the authors are able to 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*. [49], 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 [17, 54] 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 mechanisms. In contrast, Anvil uses consensus protocols to coordinate service mesh leadership and facilitate synchronized rotations.

8 Conclusion

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 needs of the microservice domain. Our proposed solution, Anvil, fills the existing gaps in service mesh design through a performance-efficient approach that enables zero-downtime and automated security artifact rotations in a platform-agnostic manner.

Availability

The authors pledge to make all source code and data created and analyzed as a part of this work available publicly to the community, should this work be accepted for publication.

References

[1] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: Introducing Infrastructure-as-Code. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 497–498, May 2017.

[2] Alexandru G. Bardas, Sathya Chandran Sundaramurthy, Xinming Ou, and Scott A. DeLoach. Mtd cbits: Moving target defense for cloud-based it systems. In Computer Security – ESORICS 2017, pages 167–186. Springer International Publishing, 2017.

[3] Elaine Barker, Elaine Barker, William Burr, William Polk, Miles Smid, et al. Recommendation for key management: Part 1: General. National Institute of Standards and Technology, Technology Administration, 2006.

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

[5] Jay Christopherson. 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/.

[6] Cloud Native Computing Foundation (CNCF). Cloud Native Survey 2020: Containers in production jump

300% from our first survey (accessed 01/2022). https://www.cncf.io/blog/2020/11/17/cloud-native-survey-2020-containers-in-production-jump-300-from-our-first-survey/.

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

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

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

[10] Kevin Fishner. 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/.

[11] Kevin Fishner. 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/.

[12] Kevin Fishner. 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/.

[13] Scott Fulton III. 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/.

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

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

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

[17] Diksha Gupta, Jared Saia, and Maxwell Young. Peace Through Superior Puzzling: An Asymmetric Sybil Defense. In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 1083–1094. IEEE, 2019.

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

[19] Dalton A Hahn, Drew Davidson, and Alexandru G Bardas. Mismesh: Security issues and challenges in service meshes. In International Conference on Security and Privacy in Communication Systems, pages 140–151. Springer, 2020.

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

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

[22] HashiCorp. Gossip Encryption | Consul (accessed 02/2020). https://learn.hashicorp.com/consul/security-networking/agent-encryption#enable-gossip-encryption-existing-cluster.

[23] Lawrence E Hecht. The Top Challenges Kubernetes Users Face with Deployment (accessed 01/2022). https://thenewstack.io/top-challenges-kubernetes-users-face-deployment/.

[24] Buoyant Inc. Linkerd (accessed 01/2020). https://linkerd.io.

[25] Buoyant Inc. Manually rotating control plane tls credentials | linkerd (accessed 01/2020). https://linkerd.io/2.10/tasks/manually-rotating-control-plane-tls-credentials/#.

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

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

[28] Istio. Istio (accessed 01/2020). https://istio.io.

[29] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The Journey So Far and Challenges Ahead. IEEE Software, 35(3):24–35, May 2018.

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

[31] Grant Joy. 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/.

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

[33] Kubernetes. Network Plugins | Kubernetes (accessed 01/2022). https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[34] Kubernetes. Welcome! | minikube (accessed 01/2021). https://minikube.sigs.k8s.io/docs/.

[35] Kuma. Kuma (accessed 01/2022). https://kuma.io/.

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

[37] Leslie Lamport. The part-time parliament. 2019.

[38] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for inter-service access control of microservices. In 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021.

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

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

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

[42] Open Service Mesh. Github openservicemesh/osm (accessed 01/2022). https://github.com/openservicemesh/osm/.

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

[44] Heather Miller. Study: The Complexities of Kubernetes Drive Monitoring Challenges and Indicate Need for More Turnkey Solutions (accessed 01/2022). https://www.circonus.com/2020/12/study-the-complexities-of-kubernetes-drive-monitoring-challenges-and-indicate-need-for-more-turnkey-solutions/.

[45] Anubhav Mishra and Peter McCarron. Service mesh at global scale (accessed 01/2022), 2021. https://www.hashicorp.com/cgsb.

[46] Claus Pahl and Pooyan Jamshidi. Microservices: A Systematic Mapping Study:. In Proceedings of the 6th International Conference on Cloud Computing and Services Science, pages 137–146. SCITEPRESS - Science and and Technology Publications, 2016.

[47] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 476–486, 2017.

[48] Chris Richardson. Introduction to Microservices (accessed 02/2020). https://www.nginx.com/blog/introduction-to-microservices/.

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

[50] Jerome H. Saltzer. Protection and the control of information sharing in multics. Communications of the ACM, 1974.

[51] Amazon Web Services. Amazon ec2 t2 instances (accessed 10/2020). https://aws.amazon.com/ec2/instance-types/t2/.

[52] Amazon Web Services. Amazon web services (accessed 10/2020). https://aws.amazon.com/.

[53] Floyd Smith and Owen Garrett. What is a Service Mesh (accessed 12/2020). https://www.nginx.com/blog/what-is-a-service-mesh/.

[54] Harish Sukhwani, José M Martínez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), pages 253–255. IEEE, 2017.

[55] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. Security-as-a-Service for Microservices-Based Cloud Applications. In 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), pages 50–57. IEEE, 2015.

[56] Randall Thomson. 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/.

[57] Christopher Tozzi. 8 Problems with the Kubernetes Architecture (accessed 01/2022). https://www.itprotoday.com/hybrid-cloud/8-problems-kubernetes-architecture.

[58] Tetiana Yarygina and Anya Helene Bagge. Overcoming Security Challenges in Microservice Architectures. In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), pages 11–20. IEEE, 2018.

Appendix

A Consul 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 between 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 develop and implement a "best-effort" rotation add-on to Consul. To measure the downtime cost of operating Consul in conjunction with the rotation add-on, 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", 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 2 illustrates the results of five independent experimentation trials under varying EC2 hardware configurations and differing amounts of cluster sizes.

B Consul Microbenchmark of Downtime Costs

To provide a more complete picture of the sources of downtime caused by the Consul rotation add-on, we perform a microbenchmark analysis of the individual processes that constitute the rotation add-on. As part of this study, five computational processes present within the rotation add-on were determined as potential sources for the downtime suffered. Table 3 displays the full dataset that was observed throughout our experimentation of the Consul rotation add-on under varying sizes of service mesh 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 the rotation add-on, 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, 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.

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 2: Full Dataset for Consul Downtime Analysis – to measure the downtime cost incurred by the Consul rotation add-on, 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 5: Figure 9.

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: Full Dataset of Consul Microbenchmark Experimentation – to measure the cost that individual processes within the Consul rotation add-on 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 5: Figure 12.