CloudCover: Enforcement of Multi-Hop Network Connections in Microservice Deployments

Dalton A. Brucker-Hahn[*]
Sandia National Laboratories
Livermore, CA; USA
dabruck@sandia.gov

Wang Feng[+]
University of North Texas
Denton, TX; USA
wangfeng@my.unt.edu

Shanchao Li[+]
University of North Texas
Denton, TX; USA
shanchaoli@my.unt.edu

Matthew Petillo
University of Kansas
Lawrence, KS; USA
mpetillo@ku.edu

Alexandru G. Bardas
EECS and I2S
University of Kansas
Lawrence, KS; USA
alexbardas@ku.edu

Drew Davidson
EECS and I2S
University of Kansas
Lawrence, KS; USA
drewdavidson@ku.edu

Yuede Ji
University of Texas at Arlington
Arlington, TX; USA
yuede.ji@uta.edu

***Abstract*—Microservices have emerged as a strong architecture for large-scale, distributed systems in the context of cloud computing and containerization. However, the size and complexity of microservice systems have strained current access control mechanisms. Intricate dependency structures, such as multi-hop dependency chains, go uncaptured by existing access control mechanisms and leave microservice deployments open to adversarial actions and influence.**

This work introduces *CloudCover*, an access control mechanism and enforcement framework for microservices. CloudCover provides holistic, deployment-wide analysis of microservice operations and behaviors. It implements a *verification-in-the-loop* access control approach, mitigating multi-hop microservice threats through control-flow integrity checks. We evaluate these domain-relevant multi-hop threats and CloudCover under existing, real-world scenarios such as Istio's open-source microservice example and under theoretic and synthetic network loads of 10,000 requests per second. Our results show that CloudCover is appropriate for use in real deployments, requiring no microservice code changes by administrators.

***Index Terms*—Microservices, Access Control, Authorization, Static Analysis, Service Mesh, Cloud Computing.**

1. Introduction

The advent of *microservice* architectures for large-scale distributed software systems has strained existing management techniques [1]. Additionally, what were once *intra*-system connections are now separated by network boundaries in microservice applications, becoming *inter*-system connections. For example, in traditional, *monolithic* applications, function calls are made between software components, whereas under microservice architectures, these calls now cross network boundaries, requiring domain-specific security solutions for defense-in-depth.

Transitioning to *microservice* software design has enabled more reliable, robust, and scalable systems. However, this adoption has led to a significant increase in system size, dependencies, and interservice connections referred to as *microservice explosion* [1], [2]. As part of this work, we highlight two domain-enabled threats that abuse the complex, interconnected nature of microservices and circumvent the available, state-of-the-art protections. These threats, while presented in past works, have gone unaddressed by state-of-the-art technologies (e.g., [3]), due to a focus on traditional point-to-point network connections, rather than "chains" of network connections that are common in microservice architectures. The densely connected nature of microservices has revived the domain-critical threats of confused deputy and path-suffix attacks that manifest even in simple systems, such as [4], [5]. Through abuse of intricate trust relationships between services and existing policy structures, attackers may violate intended behavior of deployed systems.

Our work addresses this need by providing control-flow integrity checks and security to microservice systems across the full lifespan of network connections. In contrast, traditional static analysis approaches are unable to provide comprehensive coverage, focusing instead upon single-hop, or "point-to-point" relationships between microservices. This research effort expands the state-of-the-art by considering relationships between services that extend beyond single-hop relationships by capturing *multi-hop* dependencies present in the simplest microservice deployments [4]. To thwart the aforementioned threats, we design and implement a system able to reason about such a complex threat model. Through robust static analysis methods and

[*] *Part of this work was completed while Dalton A. Brucker-Hahn was at the University of Kansas, under the supervision of Dr. Alexandru Bardas.*
[+] *This work was done while Wang Feng and Shanchao Li were working in the Graph Lab, under the supervision of Dr. Yuede Ji.*

a holistic, system-wide view of access control, our system *CloudCover* alleviates these issues. CloudCover requires no microservice code changes and introduces a *verification-in-the-loop* approach, addressing the potential for adversarial influence within a microservice environment.

CloudCover's verification-in-the-loop methodology is enabled through a verification oracle deployed alongside microservices. The core verification-in-the-loop technique within CloudCover offloads access control requests to a standalone service external to deployed microservices that examines requests from a holistic, system-wide perspective. Leveraging the replication capabilities and default access controls in service meshes, the verification oracle is protected from network requests external to deployed microservices and gains load-balancing features for scalability. Through a static-analysis-based method, microservice source code is scanned for all outgoing network requests, and a dependency graph of service-to-service requests is created. Using graph traversal, CloudCover derives a verification ruleset of network access policies, stored within the verification oracle along with a mapping of access tokens for authorization of requests. Verification-in-the-loop tracks and maintains multi-hop access control for the lifespan of service requests as they propagate through the environment.

We evaluate our static analysis source code scanning through real-world microservice example systems (*e.g.* [4]) and manual, ground truth comparison. CloudCover successfully uncovered all multi-hop dependencies identified via manual effort in examples. Further, we evaluate the theoretical and experimental costs that CloudCover's implementation imposes by leveraging a testbed environment. Theoretically, CloudCover introduces an additional network cost of 36.54 MB/s under 10,000 microservice requests per second. However, when deployed with an example microservice system implementation under load [4], CloudCover's protection introduced negligible overhead due to the vast majority of processing occurring within the microservice applications rather than CloudCover's components. Finally, we utilize load-testing scenarios with synthetic microservices to demonstrate a worst-case performance analysis of the feasibility for CloudCover. Under such tests, the CloudCover-augmented system was able to sustain a throughput of more than 900 requests per second with network connections traversing five sequential microservices. These results demonstrate CloudCover's ability to be used in real-world environments while maintaining performant response rates.

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

- We analyze two domain-critical, previously discovered threats to microservice environments that manifest in existing microservice deployments, such as Istio's Bookinfo.
- We present an open-source, scalable code-scanning methodology using static analysis to find multi-hop microservice dependencies.
- We present CloudCover, a solution for multi-hop access control verification in microservices and evaluate its feasibility in large-scale (real-world and synthetic) microservice systems.

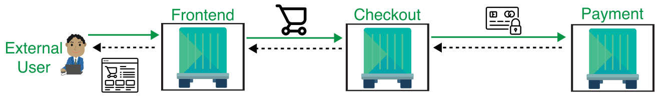


Figure 1: **Sample Microservice System** – Within the microservice eCommerce system, each microservice interacts with one another through service proxies that handle network communication and security logic.

2. Background

With cloud computing emerging as a dominant model of deploying distributed software at global scale, DevOps tools and methodologies have risen to handle new challenges. Due to enterprise-scale environments, often exceeding hundreds to thousands of deployed microservices [6]–[8], the need for better management and control methods for these systems are provided through DevOps innovations. With microservice architectures gaining popularity, service meshes have been embraced to provide networking and security logic that manages microservice activities. CloudCover expands upon these systems with improvements to access control mechanisms in service meshes.

Service Mesh Overview: The primary goal of service meshes is to provide network abstraction and security for microservices [9]. To allow developers a simplified process of developing and deploying their software, the responsibilities of service discovery, connection, and management have been delegated to service meshes. Leveraging *service proxies* [10], 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 overlay security mechanisms such as access control and mutual-TLS (mTLS) authentication for microservice applications.

Figure 1 illustrates this network architecture and the implementation of service proxies that provide mTLS connections and access control between elements of a microservice deployment. Within Figure 1, each unique microservice (frontend, checkout, and payment) operates as a containerized application. Within each of the microservice boundaries, a service mesh service proxy intercepts all network communications, incoming and outgoing, and routes them to the appropriate target microservice and applies security, such as message encryption. In our example, a small eCommerce site is constructed, able to service purchasing requests from external users. Through the capabilities of the service mesh, confidentiality, integrity, authentication, and authorization are provided to microservice applications, but do not require source code changes to the microservices [11]. By securing and verifying requests between services, service meshes contribute towards "zero-trust networking" [12] for microservice architectures.

Alternative Cloud Architectures: Serverless computing [13] has emerged as a competing architectural paradigm to microservices and service mesh deployments, but has key differences and challenges relative to microservices deployed via a service mesh. Namely, the serverless computing paradigm is a cutting-edge product that many software deployments are not mature enough to adopt, leading many to opt for Kubernetes- and service mesh-based deployments as their needs and experience matures within the cloud-computing domain. Additionally, of the available serverless computing products major cloud service providers offer unique implementations with few native runtime languages supported [14], [15]. Due to this, vendor lock-in is a present issue and restrictions of codebases may arise. In contrast, Kubernetes may be run as a cloud-provided service, or may be configured atop well-adopted operating systems and deployed in on-premise environments. While there are emerging, on-premise solutions for serverless computing, such as Knative [16] and OpenFaaS [17], these solution have yet to see major adoption in large-scale enterprises and Knative is currently listed as "incubating" by the Cloud Native Computing Foundation (CNCF) [18]. Additionally, serverless computing architectures are not suitable for many of the scenarios in which microservices are meaningful architectural choices. Namely, microservices are the prevailing, cloud-computing based, architecture for handling high-volume requests, low-latency response times, and cost-effectiveness for "permanent" processes. Due to microservices having services deployed and configured at all times, they are able to respond to requests at baseline time cost whereas under the serverless computing model, resources must be allocated and deployed before responding, creating time cost and may also face similar issues when evaluating application performance.

State-of-the-Art Shortcomings: In the current state-of-the-art service meshes, access control authorization is conducted through a series of policies and traffic rules present in service proxies [19]. These network policies, often requiring significant manual effort by system administrators, are consulted for each network request made to microservices within the service mesh. Based upon these policy evaluations, 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 because of this limited visibility. As part of this work, we analyze two domain-specific threats to microservice deployments: the ***confused deputy*** and ***path-suffix*** attacks. These threats, identified in prior work, are discussed further in Section 4. To enumerate and properly enforce the complex graph of dependencies within a microservice deployment, our design of CloudCover leverages static code analysis to reveal connections in the source code of microservices.

Code Analysis for Microservices: Code analysis aims to perform automatic inspection, understanding, and analysis of code which can provide identification of potential defects in code [20]. As developers create and improve microservices, they work directly with source code. Therefore, source code-based analysis can be applied to identify and profile microservice behavior [3]. In this work, we focus upon the behaviors of requests between microservices, referred to as *dependencies*. Additional details of commonly used code analysis techniques, such as control flow analysis, data flow analysis, and taint analysis are provided in the Appendix.

3. Related Work

Primary concerns within the domain of microservices include containerization and microservice security, access control methods in microservice systems, and analysis of microservice source code. Below, we highlight a number of relevant works to this domain, noting how our work expands upon these investigations.

Container and Microservice Security: In recent years, a range of vulnerabilities and exploits have emerged within microservice applications and the container ecosystem, such as Log4j [21], [22], leading to significant damage [23], [24]. From these events, work to understand the existing vulnerabilities and security issues in containerized applications have materialized [25], [26]. To address these concerns, various works have explored applying principle of least privilege [27], or utilizing security monitoring for containers [28]. Our development of CloudCover furthers this research by highlighting the dangers of inherent trust between microservices via an analysis of two previously discovered threats.

Access Control Methods in DevOps: Work by Nam, *et al*. [29] explores the security of container networks and the vulnerabilities exposed by some container networking systems. They provide a proof-of-concept framework to mitigate these threats and enforce proper container networking policies. CloudCover provides access control a layer above container networking, at the service mesh perspective, and addresses domain-relevant threats within the microservices domain. However, CloudCover may be combined with these previous methods to provide a defense-in-depth approach. Directly related to CloudCover, work by Li, *et al*. [3] (AutoArmor) has explored inherent trust relationships within microservice systems and how this practice may be exploited by attackers residing within compromised services. CloudCover expands upon this work by extending the threat model with new adversarial capabilities and an analysis of two critical threats to the microservices domain.

Similar access control investigations have been conducted in related domains, such as serverless computing [30]–[33]. Specifically, work by Datta, *et al*. [33] (Valve) and Jegan, *et al*. [32] (Kalium) examined the enumeration of serverless computing network calls via dynamic analysis techniques. In contrast to this approach, CloudCover leverages static analysis of source code, enumerating all of the possible network requests that a piece of code *could* create, rather than collecting dynamically exercised requests during a time period of testing. Further, as the authors note within Kalium, their approach fails to address the need for concurrent network requests, while CloudCover handles concurrent requests by default through the use of a designated verification oracle that resides external to the deployed microservices. Work by Sankaran, *et al.* [30] examined an access control approach within serverless computing architectures that describes a method for policy enforcement of a "workflow" that may traverse many functions. CloudCover extends upon the ideas presented in this work by first providing the means for extracting the necessary policies from microservice source code, but also implementing an enforcement mechanism tailored to the service mesh domain upon which, large-scale microservice applications are often deployed. Lastly, the requirements of third-party/cloud provider cooperation for implementation, as found in [32], or source code modification, as found in [31], are not necessary in this work due to the core architecture and design decisions present within the CloudCover *verification-in-the-loop* approach.

Microservice Code Analysis: As previously mentioned, AutoArmor [3] is a current state-of-the-art work that aims to apply source code analysis techniques to microservice policy generation. Through their methodology, they identify network request invocations and extract program slices associated to requests. From these slices, they catalog requests between services and generate access control policies. Our work significantly expands upon these techniques by performing similar initial request extraction, but considering larger, system-wide relationships that extend beyond single-hop dependencies. Additionally, we create a domain-aware, verification-in-the-loop approach to implementing these multi-hop, system-wide policies for enforcement.

Other works have also studied the practice of applying code analysis to generate access control policies, among which, Access-Control Explorer (ACE) [34] combines static and dynamic analysis for this purpose. Lachmund, *et al.* [35] use a static code analysis approach to generate policies that abide by the principle of least privilege with the differentiation of resource accesses initiated by an application from those initiated by a user. Our work similarly attempts to apply principle of least privilege by identifying multi-hop network requests from microservice source code to reduce the system's attack surface. Additionally, CloudCover enforces complex, multi-hop access control policies through our verification-in-the-loop approach.

4. Threat Model

A range of concerns and known issues plague the emerging microservices domain. Among these 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 [36], inherent trust among containers [37], and latent misconfigurations within containers [25], [26] have been previously found in research and reported to developers. Due to these factors, deployed microservices should be untrusted during operation and should have only the minimum necessary permissions and access.

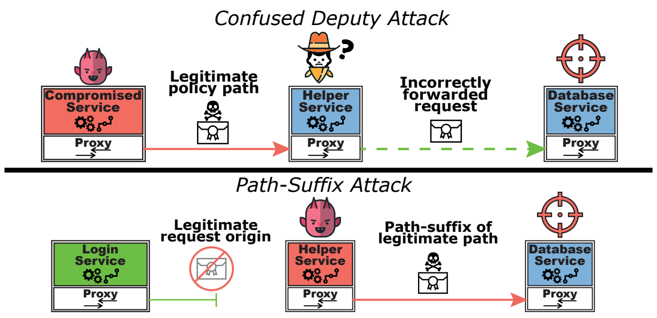


Figure 2: **Microservice Dependency Attacks** – Overview of confused deputy and path-suffix attacks.

4.1. Model Assumptions and Constraints

As part of this work, we make the assumption that the previously discovered issues and vulnerabilities within the microservice domain have the potential to manifest within real-world deployments. Further, the vast size and complexity of microservice systems necessitates domain-specific security and verification mechanisms to constrain microservice activities. From the perspective of an adversary, we consider threats against microservice code to be within scope and possible for attackers to exploit. Whether these exploits take the form of software supply-chain attacks, application-level vulnerabilities, or container compromise, we assume that microservices deployed within these environments are vulnerable and that attackers may leverage these weaknesses to gain full access over the containers in which microservice source code operates. From this standpoint, the adversary may modify application behavior within the compromised container, generate arbitrary network requests, or attempt to modify payloads leaving the application hosted within the container. With these capabilities, the adversary may attempt to create unauthorized requests to other microservices that violate the intended policies constructed by system administrators. It is important to note that we do not consider attacks against message integrity (leaving this as a task for checks and safeguards within microservice application code) or confidentiality to be within scope of this work. Further, we posit that sidecars and architectural elements of the overlay service mesh are out-of-scope for adversaries due to their code bases being relatively small, open-source and highly vetted technologies across a number of large enterprises and US government agencies [38]–[41]. Instead, we focus upon limiting the impact and horizontal actions of adversaries and their influence upon the broader system by detecting unintended actions occurring within the microservice deployment and raising alerts to system administrators for targeted remediation and isolation. While this threat model assumes a sophisticated attacker with knowledge and skills necessary to execute such attacks, we believe the domain of microservice systems and the large amount of resources dedicated to these types of deployments make them highly valuable targets for adversaries to attempt to exploit. Additionally, this threat model was previously presented and

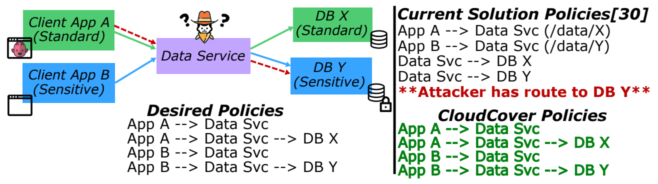


Figure 3: **Confused Deputy Analysis** – Under current access control policies in service meshes, the confused deputy attack is enabled. Adversaries may leverage multi-hop dependencies and exploit existing systems. CloudCover prevents these actions by capturing multi-hop dependencies and enforcing these policies with verification-in-the-loop access control.

leveraged in work by Li, *et al*. [3]. This work expands upon this previously analyzed threat model by considering two domain-enabled threats to microservices that have been considered by past work, but not directly addressed within the microservices domain. Namely, these threats are *confused deputy attacks* and *path-suffix attacks*. We note that while these may not represent a comprehensive list of multi-hop attacks available to adversaries, we highlight these as representative of a class of threats that have gone previously unaddressed within the domain of microservices and require domain-informed solutions to combat. Next, we elaborate on these threats and the characteristics of microservices that enable their existence.

4.2. Microservices Attacks

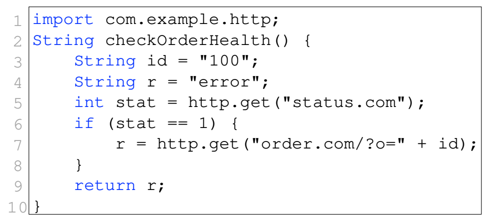
Due to the extremely complex and interconnected nature of microservice deployments, it is highly challenging to accurately describe and map all of the potential connections that microservices may make between one another. As such, we identify a previously uncharacterized structure that exists within microservice deployments that we refer to as a *multi-hop* dependency, or "dependency chains". In this sense, for the microservice system to fulfill the original request, a new request must be serviced before ultimately returning the final response. For example, a frontend service A may make a request for data retrieval from service B which then coordinates the actual data query with service C before returning final results to service A. This type of structure is shown in Figure 3 through the connections of `App A` to `Data Service` at path `/data/X` and ultimately to `DB X`. Through this multi-hop service request process, complex dependencies are naturally created and necessary within microservice architectures. Further, *all* access control schemes and state-of-the-art service mesh implementations for microservices fail to capture or address these complex relationships.

Confused Deputy Attack: In the context of this work, we refer to the "Confused Deputy Attack" as one that occurs when a benign, uncompromised service is leveraged for malicious purposes via a legitimate intermediary hop for which they have a valid access policy. Current access control methods in service mesh tools are unable to verify multi-hop relationships as traffic propagates through the system. Rather, current state-of-the-art approaches examine requests in isolation, enforcing defined policies only at the recipient microservice, rather than at a system-wide monitor. These isolated, single-hop connection policies are shown for an example system within Figure 3 under the "Current Solution Policies" heading. This shortcoming allows an opening for adversaries to traverse legitimate paths and manipulate the intermediary, or "broker" 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 behalf of the adversary. The first scenario depicted in Figure 2 shows how a compromised service traverses a legitimate path between them and the deputy in order to make illegitimate requests against a victim service. Further, Figure 3 demonstrates how such restrictive policies can fail to address this phenomenon in practical systems. Despite the strength of such an attack and the complexity of reasoning about these relationships, CloudCover allows us to detect and mitigate such threats in microservice systems (more details in Section 6).

Path-Suffix Attack: With intricate relationships between microservices in a deployment, we analyze an additional threat within the domain where an attacker bypasses the full, legitimate multi-hop dependency path. In other words, the entirety of a multi-hop path should be secured and access should only be provided in the case where the origin service generates the beginning request of a service chain. As the second scenario in Figure 2 shows, if an adversary has presence and control of a microservice that exists along a multi-hop service path, illegitimate requests may be sent to target services that exist along the suffix of a multi-hop path. Again, due to the fact that the current service mesh design only accounts for isolated, single-hop connections, this complex dependency relationship goes unaddressed. In this way, the adversary may bypass the requirement of an originator to start a service request chain and may instead simply replay or generate malicious requests towards the target service, as shown in the second scenario of Figure 2. Figure 3 expands upon this idea by highlighting the policies generated by CloudCover that prevent such attacks. Under the "CloudCover Policies" heading, any request destined for a `DB` service within the environment must originate by one of the `App` services and may not originate directly from the `Data Svc`. By accounting for the entirety of the service request path and enforcing proper prerequisite approval for service requests, CloudCover is able to prevent such threats. CloudCover helps to ensure that "zero-trust" networking is maintained and malicious presence within a microservice deployment is limited.

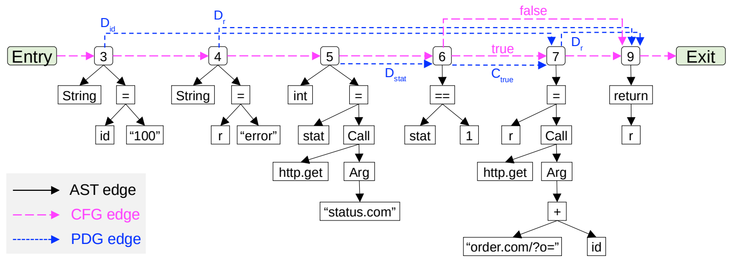
5. Microservice Code Analysis

The goal of microservice code analysis is to extract the requests between different microservices, which will serve as the foundation for later access control policy enforcement. With that, we can construct a request dependency graph that can help verify the validity of the real-time requests. In the

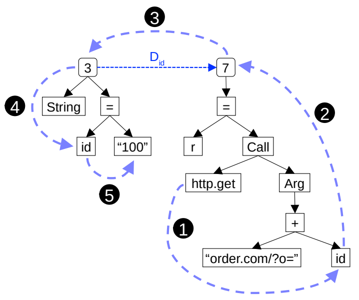


```
import com.example.http;
String checkOrderHealth() {
    String id = "100";
    String r = "error";
    int stat = http.get("status.com");
    if (stat == 1) {
        r = http.get("order.com/?o=" + id);
    }
    return r;
}
```

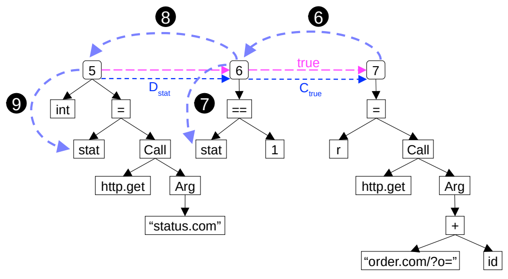
(a) An example service code.



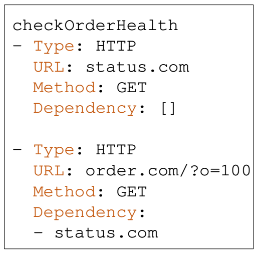
(b) The CPG of the code in (a).



(c) Service request extraction.



(d) Dependency-aware request extraction.



```
checkOrderHealth
- Type: HTTP
  URL: status.com
  Method: GET
  Dependency: []

- Type: HTTP
  URL: order.com/?o=100
  Method: GET
  Dependency:
  - status.com
```

(e) Extracted result.

Figure 4: **Dependency Analysis Methodology** – A simplified example of our service request extraction method. (a) denotes an example service code, (b) shows its CPG, (c) explains our service request extraction method, (d) explains our dependency-aware request extraction method, and (e) shows the extracted result.

following, we will discuss the unique challenges of analyzing microservice code, and our methods for addressing them. We also provide a comparison with the state-of-the-art in the Appendix.

5.1. Challenges of Microservice Code Analysis

We identify three unique challenges in analyzing microservice code: cross-language, accurate extraction of requests, and dependency across microservices.

Cross-Language: A microservice application typically involves multiple services, each of which provides well-defined APIs that are functionally independent from other services. Each service can be implemented in a variety of languages. Different combinations of these languages pose a challenge to code analysis due to different syntax, semantics, libraries, and frameworks. In addition, some communication protocols have flexible formats that can be customized by users, e.g., the remote procedure call (RPC) [42]. Furthermore, tedious human efforts to replicate code analysis techniques between languages are not scalable. To address this challenge, we design our code analysis technique on top of a *language-agnostic representation*, i.e., code property graph (CPG). We discuss this in more detail in Section 5.2.

Accurate Extraction of Service Requests: To provide the ground truth for allowed access policies, accurate extraction results are necessary. However, it is challenging to accurately extract these results since the correct context information is required to find the exact requests and parse variables in said requests. Additionally, configuration files must be included as important key values might be explicitly defined. To address this challenge, we design a control and data flow graph backtracking algorithm that can accurately extract and parse the requests. We elaborate on this in Section 5.2. Alternative approaches to performing request extraction have examined using dynamic analysis to track the behaviors of source code during operation, however, dynamic analysis may not provide comprehensive coverage of all possible network requests if they are note appropriately exercised. In contrast, our static analysis approach extracts all *possible* network requests, whether they are commonly exercised or not, due to analysis of source code directly.

Dependency Across Services: Though a group of given services are independent of each other, microservices are often required to work together to complete a task. For example, before a request is made to an `order` microservice, we must check user validation and retrieve product details by making requests to the `user` and `product` microservices, respectively. This opens up the possibility for dependencies across microservices, which must be analyzed. To address this challenge, we design a dependency-aware extraction technique. We discuss this in more detail in Section 5.3.

5.2. Language-Agnostic Code Analysis

To address the cross-language challenge, we leverage a language-agnostic code analysis technique. In particular, we represent different types of microservice code as code property graphs (CPGs), and design an accurate service request extraction algorithm.

A **code property graph (CPG)** is a unified data structure that represents the code in a graph-based format [43].

Algorithm 1: CPG-based service request extraction.

```
Input: $G$ - Code property graph of the microservice;
       $M$ - Methods of sending requests to others;
Output: $R$ - Extracted requests
1  $Q = \emptyset$
2  for each method $m \in G$ do
3      if $m \in M$ then
4          $R, D = \emptyset$
5          for $arg \in m.ast$ do
6              find($arg$, $R$, $D$)
7          parse the requested service $s$ through $R$
8          $Q.push(s)$
9          extract dependencies based on $D$ and $Q$
```

Algorithm 2: Find the value of a variable.

```
Function find($v, R, D$):
1  if $v$ is constant or pre-defined in configuration files then
2      $R$.push($v$) and return
3  if $v$ is a user-defined input argument then
4      $R$.push($*$)
5  if $v$ uses other variables then
6      for each variable $a$ used by $v$ do
7          find($a, R, D$)
8  if $v$ uses the return values of another method $b$ then
9      find($b, R, D$)
10     if method $b \in M$ then
11         $D.push(b)$
12 if $v$ is a parameter of a method $c$ then
13     find($c, R, D$)
```

CPGs provide the foundation for our static analysis methodology. We use CPGs in this scenario for two main reasons. (i) *CPG is language-agnostic* as it focuses on capturing the relationships and dependencies between code elements rather than specific language details. In particular, a popular CPG framework, Joern [44], has developed specific front-end tools to convert more than ten languages to CPGs. (ii) *CPG enables deep static analysis processes* required for the follow-up code analysis in microservice applications.

A CPG includes three types of subgraphs: abstract syntax tree (AST), control flow graph (CFG), and program dependence graph (PDG). An AST encodes the syntactic information in a hierarchical tree representation, a CFG represents the execution flow between instructions in a graph representation, and a PDG captures both data and control dependencies between the program statements.

Figure 4(b) denotes the CPG of an example code in Figure 4(a). Line 7 is denoted as node 7 in CPG. It is further decomposed into an AST rooted in 7, which includes complete syntax of this statement. Also, line 7 has a control flow dependency of line 6 based on the condition of whether variable `stat` equals 1. Further, variable `id` in line 7 has a data dependency on its definition in line 3.

CPG-based Service Request Analysis: Algorithm 1 presents our service request analysis algorithm. Given the CPG of the service to be analyzed, we aim to find out which services it requests. To achieve that, we first identify the methods of sending requests in the CPG (lines 2-3). Then, we parse the parameters, find out their specific values, and merge them to get the exact services. In particular, we traverse the method's AST (line 5). For each variable on the AST, we call the `find` algorithm (Algorithm 2) to find out the values of these variables (line 6). Next, we parse the values of that request using a variable value queue (line 7).

Algorithm 2 illustrates the `find` algorithm, which finds the value of a variable by recursively backtracking. Given a variable v, there are five different cases we need to consider. (i) If the variable value is constant, e.g., a string, or it is pre-defined in the configuration file, we use it as the value for variable v (lines 1-2). (ii) If the variable is a user-defined input argument, that means we cannot get its specific value as it is determined during runtime, we use a wildcard to represent it (lines 3-4). (iii) If the variable uses other variables, we will call the `find` algorithm to recursively determine the value of other variables (lines 5-7). (iv) If the variable uses the return value of another method, we will call the `find` algorithm to recursively find the return value of this method (lines 8-9). (v) If the variable is a parameter of a method, the `find` algorithm recursively finds its value (lines 12-13).

Framework-Agonostic RPC Analysis: RPC (Remote Procedure Call) is a communication protocol that allows a program to execute a procedure in a different address space, typically on a remote system [42]. There are different RPC frameworks and they define their own APIs and store them in different configuration files, e.g., `.proto` and `.thrift` for gRPC [45] and Apache Thrift [46], respectively. This poses a challenge for analyzing different RPC frameworks. To address this challenge, we propose a framework-agnostic RPC analysis method. In particular, we first scan all the method definitions from the CPGs of the services to extract all the RPC-related APIs. In this step, we locate a method as the combination of method name, parameters, and return values. After that, we apply the same CPG-based service request analysis to capture the requests.

Example: Figure 4(c) presents an example of extracting a service request. Starting from a method called `http.get`, which is known to be used for sending requests, we find it uses a string constant and another variable `id`. With that, we follow ❶ to find the value of variable `id`. With that, we backtrack the node `id` to node 7 following ❷. Then, as node 7 has a data dependency on node 3, we backtrack it to node 3 following ❸. Realizing node 3 is the root node of an AST subtree, we traverse it to find the definition of variable `id` and its value "100" as shown by ❹ and ❺. To this end, we are able to recover the value of the service request at line 7 as "`order.com/?o=100`", shown in Figure 4(e).

5.3. Dependency-Aware Analysis

Though the language-agnostic code analysis method can identify service requests, it misses the dependency across different services as discussed in Section 5.1, which is also a major limitation in existing microservice code analysis methods, e.g., AutoArmor [3]. To address this challenge, we design a dependency-aware analysis method by identifying three types of dependencies as follows.

(i) *Data dependency* refers to the case that there exists a data dependency between the two services. For example, the value of one service request can be used to construct another service request. To capture that, we add an extra step in extracting service requests. In particular, when the value of variable v is affected by another method, we not only recursively find its value, but also save this dependency into a queue D (lines 10-11 in Algorithm 2). When it is finished, we further extract the dependencies based on D and service request queue Q.

(ii) *Control dependency* refers to the case where there exists a control flow among different service requests. For example, the result of one service request is used in the condition of making another service request. In the example service code in Figure 4(a), the service request in line 7 is controlled by the condition in line 6, while the variable `stat` in line 6 is defined by another service request in line 5. To find the control dependency, we backtrack the service requests based on the control flow edges. If any service request-related variables are identified, we backtrack them with an algorithm similar to Algorithm 2 to recover values. Note our method can capture both dependencies even if they appear simultaneously.

(iii) *Architecture dependency* refers to the dependency relationship between different services from the architecture design of a specific microservice application. Different from the previously discussed data and control dependencies which are mainly used to analyze one service, the architecture dependency aims to capture the overall architecture of all the services in a system. In particular, based on the extracted relationship from the service requests, data dependency, and control dependency, we construct the overall architecture of a service by backtracking. Additionally, we identify the originator services, which are critical to thwarting our identified threats, especially the path-suffix attack. Specifically, a service is regarded as an originator if it either (i) only makes requests without accepting any requests; or (ii) accepts requests from other services, while there is no control or data dependency between them.

Example: Figure 4(d) shows an example of capturing the data and control dependency. Starting from the service request at node 7, we find it has a control flow dependency to node 6. Then, we backtrack to node 6 by following ❻. Similarly, we traverse the AST rooted at node 6 and find a variable named `stat` by following ❼. As its value is unknown, we backtrack it to node 5 by following the data flow edge (❽). By traversing the AST rooted at node 5, we realize the value of `stat` is defined by the service request "`status.com`" (❾). To this end, we are able to identify this dependency across different services. As shown in Figure 4(e), we extract out metadata corresponding to dependencies to compile a comprehensive list of connection flows within analyzed microservice systems.

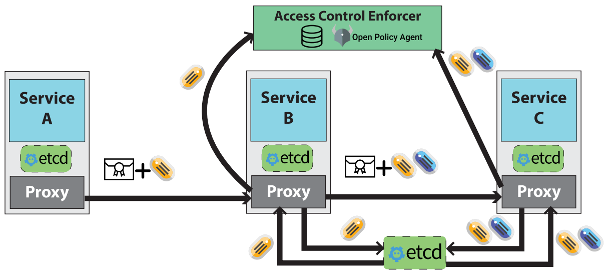


Figure 5: *Verification-in-the-loop* involves service proxies consulting with a verification oracle to approve all service-to-service requests within a deployment. We utilize `Open Policy Agent` [47] and `etcd` [48] to facilitate this process. We test with `etcd` configurations to investigate performance benefits.

To provide a direct comparison of our methodology against the current state-of-the-art tooling, AutoArmor [3], we provide a detailed description of our techniques in comparison to this previous approach and present it in the Appendix. Additionally, as part of our evaluation in Section 7, we examine five example microservice deployments and compare our results against the results previously presented in the AutoArmor work [3].

6. CloudCover

Utilizing the current state-of-the-art technologies in the DevOps toolset and service meshes, we design and implement CloudCover to address the shortcomings in access control enforcement for microservice deployments.

6.1. Architecture and Design

Due to observed issues within the state-of-the-art in microservice access control, holistic visibility and control over microservice behavior is critical in thwarting the domain-critical threats noted in Section 4. As such, we design CloudCover to operate as a supplemental access control system alongside existing tooling used for deploying and operating microservices. CloudCover leverages three key elements in its defense: a list of microservice dependencies, a capability to intercept traffic inbound/outbound from a microservice, and an external service, we term a "verification oracle", that renders acceptance decisions for requests based upon extracted policies (single-hop and multi-hop). Access control requests are received from service proxies deployed at the network boundary of all microservices. Instead of utilizing the built-in access control methods provided by Istio [49], we re-route all access control checks to the verification oracle for approval. By re-routing this traffic to a standalone entity within the deployment, observation of microservice behavior across multi-hop dependencies can be captured and system-wide visibility into the security concerns of deployments are possible.

Another factor in this process 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 and historical context needed to perform multi-hop access control verification tasks. To accomplish this, we first assign a unique name and key tuple to each unique microservice that is attached to requests that exit the microservice boundary. Next, we deploy etcd [48] storage elements for temporary storage of previous request metadata (list of target services with corresponding access keys from originating services) so that a record or "chain" of previous microservice request information can be passed in request headers. The storage of request headers are stored external to microservices, residing securely within the CloudCover system elements and are transparent to the underlying microservice code. In this manner, microservice code changes are not necessary and attackers residing within a compromised microservice have no knowledge of the CloudCover system or metadata critical to verification processes. When a microservice receives a request, the Istio and etcd components first extract this name+key tuple and forward it for verification before allowing the request access to the target microservice. The verification oracle examines the tuple that is sent and the corresponding policy entry for a target microservice to validate the originating microservice's claim to target resources. For multi-hop connections, the process remains the same, however, when the CloudCover metadata tuples are appended as the number of hops grows and the verification oracle examines all tuples and their targets for validity before approving the request. If the verification decision is to accept the traffic, the request is stripped of any CloudCover details and verification-in-the-loop tokens, and the request payload is passed to the target microservice.

While this design necessitates new network connections be generated by the service mesh elements for access control verification, separating the access control 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 7, we provide theoretical and experimental data to support the feasibility and scalability of CloudCover in real-world environments.

6.2. Implementation

We implement our previously described approach using industry-standard technologies, namely Kubernetes [50] for container orchestration and management of workloads. Atop Kubernetes, we leverage the Istio service mesh [49] for service proxy capabilities and service management and discovery. Next, we make use of Open Policy Agent (OPA) [47] for providing the mechanism of enforcement for microservice requests made within the deployment and we utilize etcd [48] for storage of request metadata and tokens from previous connections. The combination of these technologies and the overall structure of deployment (under both tested configurations) is provided in Figure 5.

Next, we deploy experimental workloads of our design to a privately 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 [51] server was used for Kubernetes control-plane purposes to coordinate the deployment 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.

7. Evaluation

The evaluation of CloudCover is comprised of three components: a performance investigation of extracting microservice requests from source code, an analysis of the network overhead imposed by CloudCover, and experimental microservice deployments under realistic load. This approach is intended to show both the security-performance tradeoff and the system costs when making use of CloudCover for microservice access control enforcement.

7.1. Performance of Service Request Extraction

This experiment studies the performance of service request extraction. We implement the code analysis task of CloudCover on top of a popular code property graph framework, i.e., Joern [44]. We implement it to support five programming languages that are frequently used in microservice applications, including C++, Java, JavaScript, Ruby, and Python. The experiments are performed on a server with two Intel Xeon Silver 4309Y CPUs running Rocky Linux 8.6. We compared it with AutoArmor [3], which is the state-of-the-art code analysis-based access control framework for microservices.

We tested five microservice applications as summarized in Table 1, including Bookinfo [4], Online Boutique [5], Sock Shop [52], Social Network [53], and Media Service [53], The first three are commonly tested by existing works, e.g., AutoArmor [3], the other two are from DeathStarBench [53], which is an open-source benchmark suite to evaluate the performance of large-scale cloud microservices-based applications.

In particular, Bookinfo [4] is a simple bookstore application that displays the description and reviews of a book. Online boutique [5] is a web-based e-commerce application where users can browse items, add them to the cart, and purchase them. Sock shop [52] simulates the user-facing part of an e-commerce website that sells socks. Social

TABLE 1: **Dependency Extraction** – Results of service request extraction for five microservice applications. LoC denotes lines of code. Service requests column denotes the number of requests extracted by CloudCover, AutoArmor, and ground truth, respectively. Time column denotes the runtime of CloudCover and AutoArmor, respectively. Speedup denotes CloudCover's extraction performance relative to AutoArmor. *Note: AutoArmor was not evaluated against DeathStarBench's examples, therefore, no comparison for service extraction, time required, or speedup is provided.*

Application	Service	Language	LoC	Service requests	Time (s)	Speedup
BookInfo	productpage	Python	788	**5**:3:5	**6**:21	3.5×
	details	Ruby	197	**1**:1:1	**1**:4	4×
	reviews	Java	211	**1**:1:1	**5**:27	5.4×
	ratings	JavaScript	275	**2**:2:2	**2**:27	13.5×
Online Boutique	currencyservice	JavaScript	692	-	**1**:25	25×
	paymentservice	JavaScript	668	-	**1**:26	26×
	emailservice	Python	1,317	-	**1**:20	20×
	recommendationservice	Python	1,389	**1**:1:1	**1**:28	28×
	adservice	Java	659	-	**2**:29	14.5×
SockShop	front-end	JavaScript	3,503	**33**:33:33	**126**:125	0.99×
	orders	Java	1,710	**8**:8:8	**9**:55	6.1×
	carts	Java	1,541	**7**:7:7	**23**:48	2.1×
	shipping	Java	594	**2**:2:2	**6**:34	5.7×
	queue-master	Java	667	**2**:2:2	**4**:31	7.8×
Social Network (DeathStarBench)	composepostservice	C++	632	**14**:-:14	**3**:-	-
	hometimelineservice	C++	427	**6**:-:6	**1**:-	-
	mediaservice	C++	103	-	**1**:-	-
	poststorageservice	C++	718	-	**1**:-	-
	socialgraphservice	C++	1,123	**48**:-:48	**7**:-	-
	textservice	C++	235	**4**:-:4	**1**:-	-
	uniqueidservice	C++	222	-	**1**:-	-
	urlshortenservice	C++	264	-	**1**:-	-
	usermentionservice	C++	304	-	**1**:-	-
	userservice	C++	1,042	**14**:-:14	**2**:-	-
	usertimelineservice	C++	513	**3**:-:3	**1**:-	-
	writehometimelineservice	C++	215	**2**:-:2	**1**:-	-
Media Service (DeathStarBench)	castinfoservice	C++	429	-	**1**:-	-
	composereviewservice	C++	881	**18**:-:18	**3**:-	-
	movieidservice	C++	430	**12**:-:12	**2**:-	-
	movieinfoservice	C++	540	-	**1**:-	-
	moviereviewservice	C++	463	**3**:-:3	**1**:-	-
	pageservice	C++	251	**8**:-:8	**1**:-	-
	plotservice	C++	328	-	**1**:-	-
	ratingservice	C++	171	**4**:-:4	**1**:-	-
	reviewstorageservice	C++	423	-	**1**:-	-
	textservice	C++	123	**4**:-:4	**1**:-	-
	uniqueidservice	C++	283	**4**:-:4	**1**:-	-
	userreviewservice	C++	461	**3**:-:3	**1**:-	-
	userservice	C++	1,081	**12**:-:12	**2**:-	-
Total	39 Unique Services	5 Languages	25,873	221:60:221	-	-

network [53] is a broadcast-style social network with unidirectional follows relationships. Media service [53] is an online website for browsing movie information, as well as reviewing, rating, renting, and streaming movies. In total, there are 39 unique services crossing five programming languages, including C++, Java, JavaScript, Ruby, and Python.

Table 1 summarizes our results, providing two interesting conclusions. First, CloudCover is able to identify all the service requests inside five microservice applications. In particular, it uncovers all 221 requests, matching the results of manual ground-truth analysis, from the 39 unique services, with an average of 5.7 requests per service. For the first three applications, AutoArmor is able to capture most of the requests, i.e., 60 out of 62. It failed to capture two from the `productpage` service of the `Bookinfo` application caused by ignoring the multiple requests for the same service (more details in Section 7.3). Also, the `front-end` service of `SockShop` sends up to 33 requests, which clearly demonstrates the frequent communication between services.

Second, CloudCover runs much faster than AutoArmor. That is, CloudCover achieves 11.6× speedup over AutoArmor on average of the 14 commonly tested services from the first three applications, where the speedup is calculated by the runtime of AutoArmor relative to ours. CloudCover achieves the highest speedup for the `recommendationservice` in `Online boutique`, which is 28×. Such a high speedup derives from our design of directly extracting requests on CPGs, instead of running program slicing first and then extracting requests. CloudCover runs similarly for one service, `front-end`. This is because AutoArmor only identifies the requests, while we need to analyze the dependencies after we identify the requests. In addition, CloudCover is also fast for the other two applications in DeathStarBench [53], i.e., 1.5 seconds per service.

7.2. Network Overhead

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 deployment that facilitate CloudCover implementation. 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 deployed alongside 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` [54] Kubernetes plugin that deploys an instance of `Wireshark` [55] at the Kubernetes pod boundary to collect and report network traffic that passes through the pod (pairing of microservice container with CloudCover component).

CloudCover Standalone Configuration: Based on our experiments and observations, in the standalone configuration, CloudCover creates five new network connections for our verification-in-the-loop system. We mathematically represent the cost of these network connections in Equation 1.

$$O_s = (O_{rq(e+OPA)} + O_{rp(e+OPA)} + O_{ack(e+OPA)}) * n$$
$$O_s = (2,673 + 2,992 + 360) * n \tag{1}$$

In the above equation, total (system) overhead cost is represented as O_s comprised of sub-components ($1.O_{rq(e+OPA)}$, $2.O_{rp(e+OPA)}$, $3.O_{ack(e+OPA)}$, and n). These components represent: (1) requests (O_rq) for verification with network calls to `etcd` and `OPA` components; (2) responses (O_rp) from verification components with network responses from `etcd` and `OPA`; (3) and finally the acknowledgement (O_ack) packets from CloudCover components `etcd` and OPA. n represents the total number of verification requests occurring in the system over a period of time. Experimental results show the total cost of this process is five network connections, totaling 6.025 KB exchanged in TLS handshakes and payloads between elements.

Using this experimental data, we calculate an estimation of overall network load imposed upon deployments with CloudCover. The results of CloudCover's cost in large-scale systems are provided in Figure 6. The first three bars of each cluster represent component costs of CloudCover's implementation (described in Equation 1) and are summed, resulting in the final bar of each cluster (O_s). As shown, under 10,000 service-to-service requests per second, CloudCover's components introduce a network load of 36.54 MB per second (over the baseline microservice deployment), which we believe to be an acceptable cost for large, performant networks. The other component of cost considered with respect to network load when using CloudCover is response latency imposed by our verification-in-the-loop approach. These details are discussed below in Section 7.3 where we perform load testing upon CloudCover.

CloudCover Sidecar Configuration: To offset cost and improve system performance, we design and implement an alternative configuration for CloudCover that makes use of `etcd` [48] as a sidecar container alongside microservices, similar to service proxies. By adapting CloudCover, we save four inter-pod network connections, relative to the baseline implementation that invokes connections external to the pod, now containing the microservice container and CloudCover `etcd` component.

$$O_s = (999 + 162 + 72) * n \tag{2}$$

Equation 2 represents the optimized cost of CloudCover when operating in the sidecar configuration which significantly decreases network load. 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`. Under these circumstances, a load of 10,000 service-to-service requests per second results in additional network load of 12.25 MB per second atop the network cost of the microservices. Details regarding network latency costs imposed under this configuration and the performance of CloudCover relative to a baseline of no security mechanisms are provided below in Section 7.3.

7.3. Case Study: Istio Bookinfo

The Bookinfo example application [4], created by the authors of Istio is intended to be a small-scale representation of a microservice environment. It contains 6 microservice definitions that collaborate to provide the functionality of a simple bookstore application. Even in small-scale and limited scope, the Bookinfo example contains multi-hop dependency relationships demonstrating that such relationships manifest in real-world deployments, thus requiring domain-specific access control policies for these connections.

To analyze the Bookinfo application, we perform language-agnostic code analysis and dependency detection upon the microservice source code independently. Using all known methods of sending network requests (i.e., Python's `requests` library), we extract all requests and dependency relationships between unique microservices. Additionally, the Bookinfo microservices have conditions that guard certain network requests. In comparison to AutoArmor [3], we capture dependencies that their analysis misses. Following initial dependency extraction, we apply our dependency-aware analysis to extract architectural dependencies. To accomplish this, backtracking of the initially collected dependencies is performed and multi-hop dependencies are constructed. For example, $ProductPage$, $Reviews$, and $Ratings$ constitute a multi-hop dependency chain.

Table 1 enumerates all service requests that occur within the sampled open-source microservice systems. We show the number of service requests extracted from source code by CloudCover, relative to a ground truth manual analysis and the comparative work of AutoArmor [3].

To capture the practical overhead imposed upon a microservice deployment when using CloudCover, we conduct a series of performance trials using the Locust [56] load testing framework. We first examine our case study microservice example deployment, Bookinfo and conduct experiments within our testbed environment as discussed previously in Section 6. We analyze the results of these experiments based upon total number of requests answered, average latency, 90th percentile latency (90% latency), and requests per second, the results of which are presented in Table 2. As seen by the results, the Bookinfo deployment

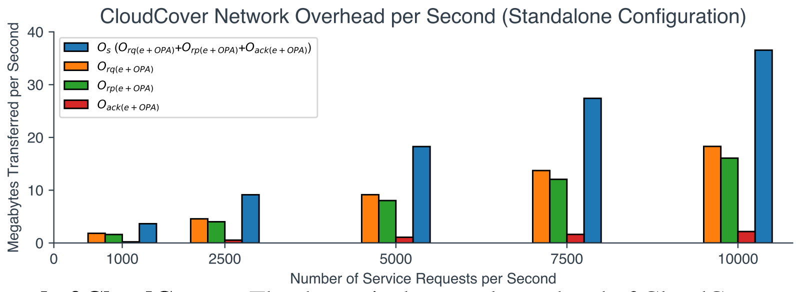


Figure 6: **Network Overhead of CloudCover** – The theoretical network overhead of CloudCover components (O_{rq}, O_{rp}, and O_{ack}) introduced by CloudCover, with etcd deployed as a standalone service in the environment, is shown. The summation of component costs is the final bar on the right of each cluster (O_s). Through packet captures and calculations, we compute the *additional* overhead to a microservice system introduced by CloudCover, separate from baseline microservice operations. In the worst case of 10,000 service-to-service requests generated per second, CloudCover imposes a network overhead of 36.54 MB/sec. Detailed analysis of network overhead of component elements (OPA and etcd) are provided in the Appendix.

Trial Type	**Total Reqs**	**Avg. Latency (ms)**	**90%ile 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 2: **BookInfo Experiment Results** – Experiment trials conducted over 10mins of full load using Python Locust.Our results show CloudCover, in both deployment scenarios, performing as well as scenarios with Istio's default access control enabled and no access control enabled.

behaves nearly identically in all trials with varying security configurations. We believe that this is due to the processing time associated with requests being consumed by microservices themselves, rather than the security components associated with deployments. The outcome of these trials indicate that CloudCover is not the source of significant latency and processing requirements, even in very simple microservice deployments. Expanding upon this finding, we present a series of detailed experiments under these "worst-case" conditions where microservices do very little processing of requests in the Appendix.

8. Discussions and Limitations

When considering whether to centralize or decentralize access control in distributed environments, it is critical to consider possible tradeoffs, such as performance for security.
CloudCover Deployment Strategies: CloudCover's implementation relies upon an external "verification oracle" to validate requests between microservices. CloudCover extends the threat model presented by [3] to include system-wide relationships spanning "chains" of microservice requests. Without enforcement being made by an external element, the context of multi-hop connections are lost and the risk of a "false originator" emerges. Through our experimental observations of microservice systems under load, we observe performance loss due to the use of CloudCover. However, leveraging cloud deployment techniques such as sidecar containers and replication, large portions of the cost of CloudCover may be offset. These performance improvements are illustrated in the Appendix. A further area of exploration would be to deploy the verification oracle as a "DaemonSet" [57]. In the context of Kubernetes, this means that an instance of the verifier will be deployed on each of the physical machines hosting microservices, providing load-balancing and close network proximity.
Security vs. Performance: As with any security solution, it is important to consider the security-performance tradeoff created by introducing new systems. As discussed previously, CloudCover is the *only* access control solution for microservice architectures that addresses the domain-specific threats analyzed in this work. As microservice systems grow and become increasingly difficult to manage, strong security techniques grow in importance as well. CloudCover significantly strengthens the security posture of these deployments without the need for administrator intervention or effort.

9. Conclusions

CloudCover represents a critical step in securing microservice systems by providing holistic access control for the lifetime of microservice requests. Through our static analysis dependency extraction methods and verification-in-the-loop implementation, CloudCover is able to detect and enforce multi-hop dependencies in microservices. Our proof-of-concept system is able to defend against two previously unaddressed threats to microservices that we highlight as part of this work. Additionally, CloudCover is able to sustain high-throughput and strict access control policies in environments with heavy request transaction load, demonstrating feasibility in real-world environments.

10. Acknowledgements

We are grateful to our anonymous reviewers and shepherd for their thoughtful and constructive comments. This work was supported by the National Science Foundation

(NSF) Awards 2143393 and 2319975. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in SoutheastCon 2016. IEEE, 2016, pp. 1–5.

[2] D. Taibi, V. Lenarduzzi, and C. Pahl, "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, 2018.

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

[4] Istio Authors, "Bookinfo application (accessed 05/2024)," https://istio.io/latest/docs/examples/bookinfo/, 2024.

[5] G. C. Platform, "Online Boutique (accessed 05/2024)," 2024, https://github.com/GoogleCloudPlatform/microservices-demo.

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

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

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

[9] R. Chandramouli and Z. Butcher, "Building secure microservices-based applications using service-mesh architecture," NIST Special Publication, vol. 800, p. 204A, 2020.

[10] Istio, "Architecture," 2022, https://istio.io/latest/docs/ops/deployment/architecture/ (accessed 07/2022).

[11] ——, "Istio / security faq (accessed 01/2020)," 2020, https://istio.io/latest/about/faq/security/.

[12] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," National Institute of Standards and Technology, Tech. Rep., 2020.

[13] A. W. Services, "Serverless computing – amazon web services (accessed 09/2024)," 2024, https://aws.amazon.com/serverless/.

[14] ——, "Lambda runtimes - aws lambda (accessed 09/2024)," 2024, https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html.

[15] G. C. Platform, "Serverless — google cloud (accessed 09/2024)," 2024, https://aws.amazon.com/developer/tools/.

[16] T. K. Authors, "Home - knative (accessed 09/2024)," 2024, https://knative.dev/docs/.

[17] A. Ellis, "Home — openfaas - serverless functions made simple (accessed 09/2024)," 2024, https://www.openfaas.com/.

[18] T. L. Foundation, "Knative — cncf (accessed 09/2024)," 2024, https://www.cncf.io/projects/knative/.

[19] Istio, "Authorization for HTTP Traffic (accessed 06/2020)," 2020, https://istio.io/latest/docs/tasks/security/authorization/authz-http/.

[20] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," Faculdade de Engenharia da Universidade do Porto, Portugal, 2009.

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

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

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

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

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

[26] Z. Jian and L. Chen, "A defense method against docker escape attack," in Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, 2017, pp. 142–146.

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

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

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

[30] A. Sankaran, P. Datta, and A. Bates, "Workflow integration alleviates identity and access management in serverless computing," in Proceedings of the 36th Annual Computer Security Applications Conference, 2020, pp. 496–509.

[31] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein, "Secure serverless computing using dynamic information flow control," Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1–26, 2018.

[32] D. S. Jegan, L. Wang, S. Bhagat, and M. Swift, "Guarding serverless applications with kalium," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4087–4104.

[33] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in Proceedings of The Web Conference 2020, 2020, pp. 939–950.

[34] P. Centonze, R. J. Flynn, and M. Pistoia, "Combining static and dynamic analysis for automatic identification of precise access-control policies," in Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). IEEE, 2007, pp. 292–303.

[35] S. Lachmund, "Auto-generating access control policies for applications by static analysis with user input recognition," in Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, 2010, pp. 8–14.

[36] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "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. Springer, 2020, pp. 23–43.

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

[38] N. Chaillan and D. E. D. I. Co-Lead, "How did this department of defense move to kubernetes and istio," 2020.

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

[40] C. (CNCF), "With kubernetes, the u.s. department of defense is enabling devsecops on f-16s and battleships," 2020, https://www.cncf.io/blog/2020/05/07/with-kubernetes-the-u-s-department-of-defense-is-enabling-devsecops-on-f-16s-and-battleships/.

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

[42] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," ACM Transactions on Computer Systems (TOCS), vol. 2, no. 1, pp. 39–59, 1984.

[43] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in 2014 IEEE Symposium on Security and Privacy. IEEE, 2014, pp. 590–604.

[44] joern.io, "Overview — joern documentation (accessed 05/2024)," https://docs.joern.io/, 2024.

[45] gRPC Authors, "Introduction to grpc (accessed 05/2024)," https://grpc.io/docs/what-is-grpc/introduction/, 2024.

[46] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Facebook white paper, vol. 5, no. 8, p. 127, 2007.

[47] O. P. Agent, "Open policy agent (accessed 10/2023)," 2023, https://openpolicyagent.org.

[48] etcd, "etcd (accessed 05/2024)," 2024, https://etcd.io/.

[49] Istio, "Istio (accessed 05/2024)," 2024, https://istio.io.

[50] Kubernetes, "Kubernetes - Production-Grade Container Orchestration (accessed 05/2024)," 2024, https://kubernetes.io/.

[51] 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.

[52] Weaveworks, "Microservices demo: Sock shop (accessed 05/2024)," https://github.com/microservices-demo, 2024.

[53] S. G. Cornell University, "Deathstarbench: Open-source benchmark suite for cloud microservices (accessed 05/2024)," https://github.com/delimitrou/DeathStarBench, 2024.

[54] eldadru, "ksniff – kubectl plugin to ease sniffing on kubernetes pods using tcpdump and wireshark (accessed 10/2023)," 2023, https://github.com/eldadru/ksniff.

[55] W. Foundation, "Wireshark (accessed 05/2024)," 2024, https://www.wireshark.org/.

[56] L. Authors, "Locust – a modern load testing framework (accessed 10/2023)," 2023, https://locust.io.

[57] T. K. Authors, "Daemonset — kubernetes (accessed 09/2024)," 2024, https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[58] F. E. Allen, "Control flow analysis," ACM Sigplan Notices, vol. 5, no. 7, pp. 1–19, 1970.

[59] F. E. Allen and J. Cocke, "A program data flow analysis procedure," Communications of the ACM, vol. 19, no. 3, p. 137, 1976.

[60] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in 24th USENIX Security Symposium (USENIX Security 15). Washington, D.C.: USENIX Association, Aug. 2015, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ming

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

Appendix – Code Analysis Techniques

Significant work has been conducted in the domain of static code analysis for the purposes of detecting bugs and vulnerabilities in software. Below, we elaborate upon previous work, covering the commonly used techniques of control flow analysis, data flow analysis, and taint analysis.

Control Flow Analysis analyzes the ordering of program statements and flow of control within a program by identifying issues or defects in program flows (e.g., dead code, infinite loops, and incorrect control flow structures) [58].

Data Flow Analysis tracks the flow of data within a program with a focus on understanding how variables, values, and data dependencies propagate throughout code. Through data flow analysis, potential issues and defects in code can be identified. Namely, data handling issues, such as uninitialized or unused variables, and data inconsistencies can be discovered [59].

Taint Analysis is a technique facilitated by data flow analysis. It can track the flow of tainted data (e.g., untrusted data) within a program. Taint analysis can be used to understand the impact of specific data, and how tainted data can potentially influence the behavior and security of the software or system [60].

Appendix – Code Analysis Comparison with AutoArmor

Compared to the state-of-the-art work, i.e., AutoArmor [3], CloudCover outperforms in four aspects. (i) CloudCover analyzes the service requests on top of a language-agnostic representation, i.e., CPG. However, AutoArmor implements service request extraction for each language, which requires tedious human efforts and is not scalable to new languages. Instead, our design provides generalizability to support all programming languages in theory.

(ii) To extract the service request, AutoArmor uses a two-step solution with program slicing first and request extraction next. Though program slicing helps to improve the runtime of downstream request extraction, it takes a much longer time. On the contrary, CloudCover directly implements the service request extraction on the CPG, where unnecessary codes are inherently excluded. With this, CloudCover achieves 11.6× speedup over AutoArmor (more details in Section 7.1).

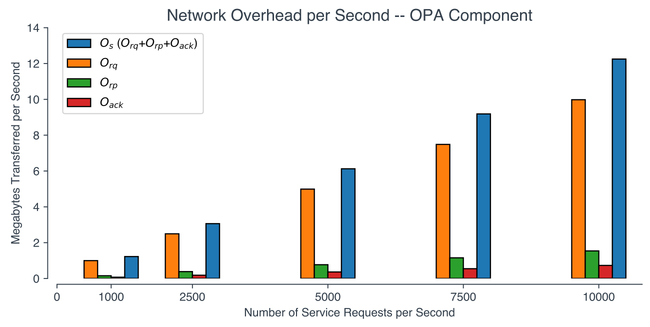


Figure 7: **Network Overhead of OPA Connections** – `OPA` is utilized for CloudCover's *verification-in-the-loop* process. We plot the *additional* overhead to a microservice system introduced by the required network connections to the `OPA` component of CloudCover in deployments. In the worst case, we observe that the `OPA` network requests impose 12.25 MB/sec upon a deployment under 10,000 requests/sec load.

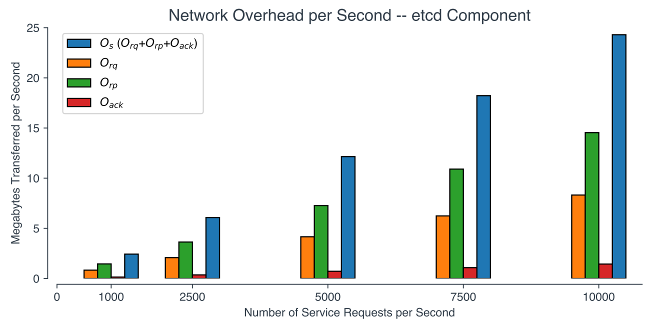


Figure 8: **Network Overhead of etcd Connections** – `etcd` is utilized for CloudCover's temporary token storage process in verification. We plot the *additional* overhead to a microservice system introduced by the required network connections to the `etcd` compoenent of CloudCover in deployments. In the worst case, we observe that the `etcd` network requests impose 24.15 MB/sec upon a deployment under 10,000 requests/sec load.

(iii) CloudCover is able to identify dependencies between different services, which is critical to defending against our identified threats and is unaddressed by previous work. This can help identify the invalid accesses that might appear valid under pair-based access control policies, e.g., AutoArmor. For the example code in Figure 4(a), existing works will allow direct access from service "`checkOrderHealth`" to "`order.com/?o=100`". This might be invalid as they ignore the service "`status.com`", which serves as a prerequisite before visiting "`order.com/?o=100`".

(iv) Together with data, control, and architecture dependency, CloudCover can identify the multi-hop request chains among different microservices. By adding the multi-hop request chains in the policies, CloudCover can thwart both path-suffix attacks and confused deputy attacks. For example, as shown in Figure 3, although an attacker can access the "`Data Service`" service from the "`Client App A`" service, it is not allowed to further access the "`DB Y (Sensitive)`" service because such a multi-hop request is not in the allowed policies.

Appendix – Network Overhead

With the inclusion of CloudCover into microservice systems, the primary source of new overhead imposed upon these environments is network load. The design of CloudCover necessitates additional network interactions and requests made amongst system components to provide for verification-in-the-loop access control. However, at the cost of these additional network requests, the microservice system receives stable security benefits not currently offered by state-of-the-art tools.

As previously described in Section 6, CloudCover is designed with two key elements that are added into microservice deployments. The first of which is the verification oracle component which is built upon `OPA`. Figure 7 illustrates the theoretical network overhead imposed by the addition of `OPA` into the microservice environment. Particularly, the theoretical network overhead required to facilitate network connections made to `OPA` that provide the verification oracle functionality. As microservice requests are made amongst deployed microservices, each time a new service request is received, `OPA` is used to verify that the request is legitimate and has a valid policy to access the requested resource. In this way, one new network connection is required to perform this transaction. The recipient microservice bundles the request metadata in a new request to `OPA` which examines the contents and returns a verification decision of approval or denial. This transaction is quite small in size, resulting in a worst-case network overhead of 12.25 MB/sec under 10,000 microservice requests per second.

The other key element of CloudCover's design is the ability for CloudCover to maintain request state temporarily during microservice requests in order to construct the multi-hop connections that `OPA` reasons about. To facilitate this functionality, we utilize `etcd` as temporary metadata storage. `Etcd` provides fast, key-value storage, which aligns well with the temporary metadata storage needs of CloudCover. As part of our experimentation and design, we consider two methods of utilizing `etcd`: first, as a standalone service that microservices make requests to for storage and extraction of request metadata; second, as a sidecar container to each of the deployed microservices for faster request and retrieval. Despite these deployment options, both scenarios require two new network requests (one for storage of inbound request metadata, one for retrieval of outbound request metadata). After calculation, we find that under a load of 10,000 microservice requests per second, the overhead imposed upon the deployment by the `etcd` component is 24.15 MB/sec. As observed, this is roughly twice the overhead imposed by `OPA`, which is logical due to

Trial Type	Total Reqs	Avg. Latency (ms)	90%ile Latency (ms)	Reqs/sec	△ Reqs/sec relative to NoSec
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 3: **Multi-Hop Fake Service Experiment Results** – Experiment trials conducted over 10mins of full load using Python Locust. **CC - CloudCover*.

the fact that `etcd` requires two small network transactions relative to `OPA`'s singular network transaction. We plot the full results of these calculations in Figure 8.

As mentioned previously, `etcd` provides the flexibility for different deployment options within CloudCover. Namely, we explore the implementation of `etcd` as a standalone service that microservices target with metadata storage transactions, but we also examine `etcd`'s ability to be deployed as a sidecar container alongside all microservices. While both options still necessitate two new network requests be made to facilitate proper functionality, their performance does differ. The first option of `etcd` deployment, as a standalone service, provides the ability for `etcd` to be replicated to provide load-balancing functionality for microservices. However, when `etcd` is deployed as a sidecar container, the network distance between the microservices and `etcd` is incredibly short, providing for high speed storage/retrieval actions which improve throughput of the overall system. Deploying `etcd` as a sidecar container alongside microservices is very low cost because `etcd` itself is a very small application, requiring a relatively small footprint on the host. Both standalone and sidecar container options were explored and the results of these experiments are elaborated further below.

Appendix – Experimental Throughput

Along with the theoretical network overhead analysis of CloudCover, we examine the response latency incurred by our proof-of-concept system through a series of load testing experiments. We accomplish these load testing experiments through sample microservice deployments and leveraging Python Locust [56], a load testing library written in Python. Our load tests were conducted in our experimentation platform described in Section 6. To generate load within the environment, we make use of five instances of load generators targeting a sample deployment to place the system under heavy load. These tests are aimed at seeing the max throughput of the environment and finding the worst-case performance of CloudCover under these circumstances.

Through these experimental trials, we examine how multiple dependency hops affect the latency of the system. We develop experiments that compare baseline microservice deployments with no security deployed against: default Istio authorization policies, CloudCover deployed in "standalone" mode where the verification oracle and `etcd` are both configured to be centralized instances, and finally CloudCover deployed with a standalone verification oracle and `etcd` deployed as sidecar containers for each of the microservices within the environment. The results of these experimental trials are shown in Table 3. As Table 3 shows, as the number of dependency hops increase in the deployed system, the number of requests able to be serviced by every form of access control decreases and the response latency suffers as a result. However, when CloudCover is deployed with `etcd` as a sidecar container for the environment, the performance of CloudCover increases significantly. From these experiments, we developed the load balancing and replication capabilities of CloudCover further in an attempt to improve performance.

In order to explore the replication and load balancing performance of CloudCover, we design additional experiments that vary the levels of replication for CloudCover components and compare these results. These experimental trials contain a five dependency hop microservice deployment targeting baseline microservice deployment performance against: Istio default access control policies, CloudCover with `OPA` and `etcd` deployed as standalone services with varying levels of replication of each component, and CloudCover with `etcd` deployed as a sidecar container and varying levels of `OPA` replication. We present the results of these experimental trials in Table 4. As our results show, when CloudCover is deployed in a load balancing fashion with multiple instances of the `OPA` verification oracle handling verification requests *and* `etcd` deployed as a sidecar container, the performance of CloudCover is highly com-

Trial Type	**Total Reqs**	**Avg. Latency (ms)**	**90%ile Latency (ms)**	**Reqs/sec**	**△ Reqs/sec relative to NoSec**
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 4: **Multi-Hop Fake Service Replication Experiment Results** – Experiment trials conducted over 10mins of full load using Python Locust. **CC - CloudCover.*

parable to Istio's default access control implementation. In this manner, CloudCover is able to provide superior security by defending against previously discovered threats, but is also able to maintain performance that is comparable to the current state-of-the-art in service meshes. Due to this, we believe that CloudCover is feasible for deployment in real-world environments.

Appendix – Fake Service Experiments

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` application [61]. Under this configuration, the microservices themselves are extremely lightweight web server applications, performing little to no processing of incoming requests. Additionally, we experiment with increasing numbers of service dependency hops to represent "deep" microservice deployments where there are many dependent services waiting for upstream service calls before returning. Table 3 presents the results of these experiments. As shown in the table, default Istio authorization policies perform very near the baseline of no security enabled, however, Istio authorization policies only consider isolated, single-hop dependencies. This limited view of dependencies may leave deployments open to the confused deputy and path-suffix attacks discussed in Section 4. Default Istio authorization policies are enforced by the service mesh directly, not requiring new network connections that leave the pod network boundary, resulting in little overhead over baseline performance. In comparison, CloudCover configurations introduce new network connections to the verification oracle, but effectively mitigate the domain-relevant threats, providing a stronger, more holistic access control approach within microservice systems.

To offset the cost of verification-in-the-loop network connections, we explore various configurations and deployment strategies for CloudCover. Access control in service meshes is vital to securing microservice systems and administrators have incentives to ensure performant applications at the cost of additional computing nodes/hosts. Due to this, we deployed replicated CloudCover elements and investigated the performance effects. These results are presented in Table 4. A notable comparison within these scenarios is the baseline performance of default Istio authorization policies relative to CloudCover when deployed with `etcd` as sidecars and `OPA` with 5 replications. Under this model, CloudCover achieves comparable performance to Istio policies and baseline performance. Due to the simple nature of the deployed workloads, we believe that these results show promise for CloudCover to be adopted in real-world deployments to secure realistic microservice systems.