Expressive Policies For Microservice Networks

Karuna Grewal
Cornell University
kgrewal@cs.cornell.edu

P. Brighten Godfrey
University of Illinois
Urbana-Champaign and VMware
pbg@illinois.edu

Justin Hsu
Cornell University
justin@cs.cornell.edu

ABSTRACT

Microservice-based application deployments need to administer safety properties while serving requests. However, today such properties can be specified only in limited ways that can lead to overly permissive policies and the potential for illegitimate flow of information across microservices, or ad hoc policy implementations.

We argue that a range of use cases require safety properties for the flow of requests across the whole microservice network, rather than only between adjacent hops. To begin specifying such expressive policies, we propose a system for declaring and deploying *service tree policies*. These policies are compiled down into declarative filters that are inserted into microservice deployment manifests. We use a light-weight dynamic monitor based enforcement mechanism, using ideas from automata theory. Experiments with our preliminary prototype show that we can capture a wide class of policies that we describe as case studies.

CCS CONCEPTS

• **Networks** → **Network manageability**; • **Security and privacy** → **Logic and verification**;

KEYWORDS

Mircroservices; Service-mesh; Security Automata.

ACM Reference Format:
Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. 2023. Expressive Policies For Microservice Networks. In *Proceedings of The 22nd ACM Workshop on Hot Topics in Networks (HotNets'23).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626111.3628181

1 INTRODUCTION

Microservice-based application design offers separation of concerns between the components of the application. This design enables flexible deployment with the configuration changing and scaling on the fly, and independent development of microservices by different teams. Each microservice exposes its functionalities over a well-defined interface for other microservices to consume, and a typical request processing involves many such interactions among a fleet of microservices. This communication pattern makes microservice interface a suitable granularity to express and enforce communication policies.

Several systems are used today to implement communication policies between microservices. Kubernetes clusters use a container-network interface (CNI) plugin (e.g., Calico [13], Cilium [4]) to set up network connectivity, e.g., allocating IP addresses to containers and setting up any necessary virtual communication tunnels. These CNI features can also be used to enforce policies by restricting communication between API endpoints. Service meshes act as a layer 7 proxy, intercepting all incoming and outgoing service traffic, and thus have access to more information. Specifically, the service mesh's sidecar proxies (e.g., Envoy [5]) understand HTTP and can see API endpoints and parameters – and can thus allow or block communication between microservices using finer grained conditions on the specific API call.

The above systems can, however, only control *single-hop* interactions: clients sending requests to certain API endpoints. Single-hop policies do not consider the rich structure of microservice interactions: a request produces a *service tree* of API calls to other microservices, which in turn may trigger several other requests. As a result, some desirable policies may not be possible to specify using single hop policies, missing the opportunity to protect against illegitimate flow of requests or concisely express the actual intent.

This paper explores the need for more expressive safety properties for network communication in microservice deployments. In addition to single-hop policies, we consider policies over the service tree associated with an API call. We present and discuss several example service tree policies: indirect connectivity, intermediate service interactions, and fine-tuned traffic management. Such fine-grained policies can be used by cluster administrators to improve safety of deployments, while also specifying higher-level policies (e.g., "no request from Internet clients should ever lead to a WRITE call to a certain database") that do not refer to specific one-hop microservice-to-microservice interactions.

HotNets'23, November 28-29, 2023, Cambridge, Massachusetts
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0415-4/23/11.
https://doi.org/10.1145/3626111.3628181

To show that service tree policies are feasible to specify, we propose a regular expression based language to specify properties over inter-service interactions. Regular expressions have been well-studied as a specification language for path-based network properties [12]. However, inter-service interactions in microservices yield a tree of service interactions, so our work adapts ideas from regular expressions to reason about properties over service trees.

Today there is no easy way to enforce such policies. One option is to directly implement the safety checks for inter-service interactions into the microservice application code. This approach is invasive for existing applications and lacks the benefits of separating policies from the application. Separating policies from the application offers modularity and convenience to application developers, who might rely on more experienced security team for the security policies. Another alternative is to enforce information flow control type safety properties using taint tracking systems, where the taints carried by requests and responses are used to prevent illicit flow of information; this was recently proposed for serverless applications [1]. However, it requires designing a security lattice to specify policies, which as we describe later is not feasible for many properties.

To address these limitations, we present a framework that compiles policies into traffic filters that configure the Istio [6] service mesh and filter the incoming requests. Our solution's policy enforcement is decoupled from the application besides assuming that the application propagates request headers (i.e., headers from incoming requests are copied by the service onto resulting outgoing requests). This is feasible for applications as it is a common pattern used for monitoring that can be implemented using common distributed tracing frameworks, like Jaeger [7] and Zipkin [15].

In summary, this paper begins to lay out the subtleties and scope of safety properties in microservice networks. The history of policies in the network layer may be instructive: safety policies and other configurations originally emerged in low level device configurations, resulting in management limitations until the advent of SDN. To avoid such pitfalls and take advantage of the unique opportunities of the service layer, our goal is to initiate a discussion of the right set of policies for the emerging domain of service layer networking.

2 MOTIVATING SAFETY POLICIES

Microservices background. Microservice application deployments are typically partitioned across several containers. The services running in these containers interact by querying API endpoints using communication protocols, for instance REST APIs or gRPC running over HTTP.

Our work focuses on safety properties enforceable at the granularity of the API calls between microservices, which

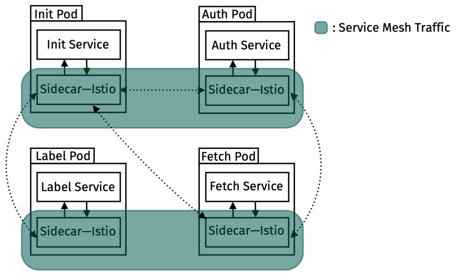


Figure 1: Example Photo Gallery Application

has become a useful place for cluster administrators to control communication policy. For example, Kubernetes container network interfaces (CNIs) typically allow administrators to define which microservices can talk with each other [8]. More granular policies are possible if the application is deployed on a *service mesh* [2]. A service mesh offloads common networking functions from applications into *sidecars*, which are supporting containers for the microservice's main deployment container. Each microservice's sidecar acts like a proxy that can handle all request flow in and out of the microservice instance. Since they understand HTTP and other protocols, sidecars, like Envoy [5] support L7 policies, whitelisting or blocking a client from reaching certain other microservice or API endpoints in some microservice.

2.1 Safety policy use cases

The distributed nature of microservices raises new challenges for safety properties, which often span multiple microservices. To ground the discussion, we describe a set of safety policies that might be useful for application deployments.

Photo gallery application. Consider a Flickr-like photo gallery application in Figure 1. On uploading an image, the frontend service invokes the face labeling service for identifying the faces in the image. The application might be implemented using the following four services:

(1) **InitFrontend** (INIT). Initiates the image post-processing.
(2) **LabelFaces** (LABEL). Labels faces in an image.
(3) **FetchFaceTags** (FETCH). Fetches the set of faces that an authenticated user is authorized to access. We assume that authorized faces are the ones that have been previously labeled by the user, so that they cannot get information about faces that they didn't already know.
(4) **Authenticate** (AUTH). Implements an additional layer of two-factor authentication around the FETCH service.

In this application, a request to the INIT service initiates a data scrubbing routine before invoking the LABEL service. The data scrubbing routine, which scrubs unauthorized set of face labels, starts with first stage of authentication. For authentication, the INIT service invokes the AUTH service with the userID and an authentication stage parameter set to `first`. This is followed by a request from the INIT service to the FETCH service, which in turn invokes the AUTH service to execute the second stage of authentication before responding with the set of authorized face labels. INIT will then invoke LABEL with these face labels.

We now describe several policies that the application or cluster administrators may want to enforce.

Policy 0: Service-to-service direct connectivity. A policy at this granularity will block any direct API call from a given service to a specified service. For example, an application or security team might disallow the INIT service to directly call the FETCH service to minimize unnecessary access.

Policy 1: Service-to-service indirect connectivity. Such policies will block any API call to an endpoint A if it was triggered by a request flow originating from a certain endpoint B. For example, a compliance team might disallow access to the LABEL service for users from a certain geographical region if the LABEL service's database storage policies are not compliant with the regional data protection policies.

Policy 2: Intermediate service interactions. Commonly, microservice-based applications are backed by databases, which might store personally identifiable information. In such settings, we may want some data to be scrubbed before letting an untrusted service U access it. In this case, a *service-to-service indirect connectivity* policy that blocks the database from serving any request that originated at U will be restrictive. This policy will not allow the database to serve data scrubbed requests that originated at U. This limitation motivates the need for policies that condition if a request from a service can be served by another based on intermediate inter-service interactions.

For instance, consider the data scrubbing routine comprising of a sequence of requests to the first authentication stage, then the FETCH service, and finally the second authentication stage. The data scrubber will ensure that the LABEL service labels only authorized faces in an image uploaded by an authenticated user. The data scrubbing policy should require that a request from INIT can indirectly reach LABEL only after the sequence of invocations to AUTH, followed by FETCH, and then AUTH. This policy is similar to the network layer waypointing policies.

Policy 3: Traffic Management. Currently traffic management applications, like A/B testing or load-balancing support only service-to-service policies. The above *intermediate service interactions* policies can be used for fine-tuned traffic management. For example, a security team wants to test the correctness of the beta version of the previously described data scrubbing routine by dog-fooding its component services on the internal users such that all gateway requests from internal users that get served by the beta version of the FETCH service should also have been served by the beta version of the AUTH service.

2.2 How can we enforce these policies?

Direct implementation in services. An ad hoc means to enforce these policies is by direct implementation in the service. For instance, a developer could add conditional checks before serving requests to an endpoint to filter them based on the source service. However, it is difficult to manage policy changes with this strategy because this approach will require application-wide changes. Ideally, the policy enforcement can be decoupled from the application so that separate networking or security team can define application-wide policies independent of the development team.

Enforcement in existing service meshes. Kubernetes container-network interfaces (CNIs) and service meshes commonly let users express policies for filtering requests, access control, or traffic splitting. However, they only support coarse policies, like in Policy 0, and are not capable of enforcing more complex policies.

Taint tracking and information flow control. Enforcement of security and safety properties have been explored through taint tracking and information flow control (IFC) systems. Intuitively, taint tracking is a method to track what data sources a piece of data, like request or response messages may depend on by carrying an additional *taint* information in its header. To use taint tracking, a user specifies the policy by designing a *security lattice*, a set of security labels with a partial order $x \leq y$ saying that label x is less secure than y. Endpoints are assigned labels from the lattice based on the data with which they interact, and taint tracking systems prohibit illict flows of data through static or dynamic enforcement. For example, an endpoint with label x can only serve incoming requests or responses that carry taint y if $y \leq x$, and outgoing requests or responses from an endpoint labeled with x will be tainted as x.

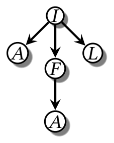
IFC systems provide strong guarantees, but have some general drawbacks. First, developers usually think of security-properties in terms of path-based properties, and it is not obvious how to design the required security lattice. Furthermore, the service graphs must be fixed for the application and the lattice cannot be dynamically modified. However, dynamic changes in microservice applications are central to support the features of elasticity and flexible deployment.

Finally, since security lattices can only represent partial order relations, they cannot express some of our properties that describe a non-partial order relation on the requests.

For example. The data-scrubbing policy in Policy 2 cannot be enforced by lattice-based IFC. An IFC system would label the service INIT, AUTH, FETCH, and LABEL with some lattice's labels, say, i, a, f, and l respectively. Requests or responses going out of a service will be labeled by the service's label. A service can serve requests with security label lower than its own label. However, designing a lattice is not feasible for this policy. This policy requires the information from INIT to not flow into LABEL, so their labels should satisfy $i \not\leq l$. Since INIT can send a request to AUTH, their labels should satisfy $i \leq a$; while $i \not\leq f$ as INIT should not bypass AUTH. Since a request from INIT tainted with a can go to FETCH, we have $a \leq f$. Similarly, since requests may flow from FETCH to AUTH, we need $f \leq a$. This implies that $a = f$. However, this will let INIT bypass AUTH and invoke FETCH directly because $i \leq a = f$, violating our intended policy.

3 AN EXPRESSIVE POLICY FRAMEWORK

Service tree policies. Consider the photo gallery application in Section 2. If we refer to INIT as I; AUTH as A; FETCH as F; and LABEL as L, the request flow for a request to INIT service in that application can be described as the service tree below, where nodes denote endpoints. The tree can be parsed from root to leaf to give various request flow paths. A *happens-before* relation can be defined on these requests in terms of the service tree, such that a request to a given endpoint happens-before the requests to any endpoint that occur after it in the pre-order traversal of the service tree. For instance, the following service tree describes three root to leaf request paths (a) INIT $\rightarrow$ AUTH, (b) INIT $\rightarrow$ FETCH $\rightarrow$ AUTH, and (c) INIT $\rightarrow$ LABEL, where requests in (a) happen before those in (b), which happen before those in (c).



We consider the following form of policies over requests,

$$regex \text{ in } (start \text{ to } final)$$

where *start* and *final* are two endpoints in our application, and *regex* is a regular expression over the set of all endpoints in the application besides *start* and *final*. *regex* specifies an acceptable temporal ordering (or happens-before relation) on the API calls in the request flow path from *start* to *final*. The above policy specifies that a request originating at *start* can arrive at *final* only after the temporal ordering of the API calls in its request flow path between the call to *start* and *final* matches the ordering specified by *regex*.

Example policy. We will now formally describe an example access control policy in our language for the data-scrubbing scenario described in Policy 2. The data-scrubbing policy in our language will be AFA in $(I$ to $L)$. This policy specifies that for a request that reaches L and happens after I, the children requests of I should satisfy A happens-before F, F happens-before A and A happens-before L.

The above policy for requests from I toward L can be relaxed to allow services besides L to be invoked before the mandatory data scrubbing routine, AFA. The regular expression (not $L)^*$ denotes zero or more occurrence of endpoints besides L. Using this, we can specify the relaxed policy as

$$(\text{not } L)^* AFA \text{ in } (I \text{ to } L).$$

Table 1 summarizes common use cases for the policies of the form *regex* in $(I$ to $L)$.

4 POLICY ENFORCEMENT

The central idea of our policy enforcement mechanism is that a request carries around a special header with context about the requests that happened before it. The policy violation checks, which are based on the incoming request's context header are offloaded from the microservice into its sidecar proxy using the service mesh layer.

To ease the translation from policy to enforcement, we propose a compilation procedure that generates traffic filtering extensions for each microservice's sidecar proxy. The proxy, which can intercept incoming and outgoing microservice traffic, runs these extensions to filter or update the context headers of the requests. The incoming and outgoing response headers remain unaffected.

Role of declarative filters. A *declarative filter* of the form

$$\begin{aligned} \text{ServiceName} ::= \text{match } hdr \text{ with} \\ &\mid u_i \longrightarrow v_i \\ &\mid \ldots \\ &\mid u_n \longrightarrow v_n \end{aligned}$$

pattern matches on the header field hdr of an incoming request to the service ServiceName. A successful match of the hdr value with u_i modifies the value of hdr to v_i. The hdr value carries the context about requests that happened before the given request; we write $\mathcal{L}$ for the set of context values. A declarative filter describes rules for updating this context. We rely on service code instrumentation for context propagation of headers between responses and outgoing requests. If an incoming request has no outgoing child request, its response carries the same hdr as the incoming request; otherwise its response carries the hdr from the response of the last outgoing child request of the incoming request. If an outgoing request is the first child request of its parent, it carries the same hdr as its parent; otherwise (its parent's) previous child request's response hdr flows into its hdr.

Example: Filters for data scrubbing. As an example, consider the policy in Section 3. We take the following set of contexts $\mathcal{L} = \{q_1, q_2, q_3, q_4, q_5, \bot, \top\}$, where q_1 records that

Table 1: Possible scenarios of safety properties enforced by *regex* in (I to L) **policy, where** I, L **are two services.**

Action	Scenarios
Access Control	Request from I should reach L after matching a sequence of data-scrubbing services given by *regex*.
Traffic Management	Requests from I should be served by beta-L after being served by beta version of services in *regex*.
Retry Policy	Limit the number of retries to L for external requests (or requests from I) if it matches a *regex*.

$\mathbf{I}$ ::= match ctx with
  $\mid * \rightarrow q_1$

$\mathbf{L}$ ::= match ctx with
  $\mid q_1, q_2, q_3, q_5 \rightarrow \top$
  $\mid q_4, \bot \rightarrow \bot$

$\mathbf{F}$ ::= match ctx with
  $\mid q_1, q_3, q_4, q_5 \rightarrow q_5$
  $\mid q_2 \rightarrow q_3$
  $\mid \bot \rightarrow \bot$

$\mathbf{A}$ ::= match ctx with
  $\mid q_1 \rightarrow q_2$
  $\mid q_3 \rightarrow q_4$
  $\mid q_2, q_5 \rightarrow q_5$
  $\mid \bot \rightarrow \bot$
  $\mid q_4 \rightarrow q_5$

Figure 2: Filters

the given request is the child of a request to I; q_2 records that the given request is a child of a request to A, which was a child request of I; q_3 records that the request flow path $I \rightarrow A \rightarrow F$ happened immediately before the given request; q_4 records that the request flow path $I \rightarrow A \rightarrow F \rightarrow A$ happened immediately before the given request; q_5 records that the I happened before the given request, but the request flow path between I and this request violated the *regex*. The context $\bot$ records that the given indirect request has reached the endpoint B while satisfying the property; $\top$ records that the given request has violated the policy. $\mathcal{L}$ has two special labels: $\bot$ resets or erases the context from the request, and $\top$ describes that the request should be blocked. As we will discuss in the next section, the set of contexts $\mathcal{L}$ can be automatically generated from the policy. Assuming that the context is carried in a custom HTTP header ctx, the filters at services I, A, F, L are described in Figure 2.

To see how these filters enforce the policy, consider a request that has the following request flow path, $I \rightarrow A \rightarrow L$. This request *should not* be allowed to label the requested image because it has not been scrubbed in accordance with the data scrubbing routine. We explain how the filters block such a request by describing the modifications to the ctx header applied by the filters. A request will start out in the $\bot$ or empty context. The sequence of ctx modification will be: (at service I) $\bot \rightarrow q_1$, (at service A) $q_1 \rightarrow q_2$, and (at service L) $q_2 \rightarrow \top$, which means that L blocks the request.

Compiling policy to declarative filters. So far, we have considered how enforcement works, given the contexts $\mathcal{L}$ and the declarative filters. To generate the contexts and filters, we use an idea from automata theory: our safety policies are regular expressions, and any regular expression can be algorithmically converted to an equivalent *deterministic finite automata* that accepts only the behaviors that follow the regular expression [11]. The set of contexts $\mathcal{L}$ can be read off from the set of automata states, and the declarative filters can then be obtained from the automata transitions.

We outline our compilation procedure, which requires a few more ideas. To compute the transition relations, the first step is to convert the policy into a regular expression, for instance, policy AFA in (I to L) becomes $IAFAL$, and generating its deterministic finite automata. However, this automaton will accept only those requests that start at I and end at L and match AFA between I and L in the pre-order traversal. But we should be able to accept requests that start at, say, I and do not reach L. Therefore, we augment the original automaton, so that the transition relations we extract from it will accept all, but only those requests arriving at L that had an I happen-before them, and the substring with I and L in pre-order traversal did not match with AFA. Finally, the set of transitions in the augmented automaton on a given service name or endpoint gives the match cases for the filters at that service.

5 PROTOTYPE IMPLEMENTATION

We implemented a compiler in 1kLoC in python that takes the policy and generates the declarative filters for each service in the format described in Section 4. These declarative filters are converted into configuration files that are used to extend the functionality of Istio's sidecar proxy, Envoy. These filters are run whenever the sidecar proxy intercepts a request at the service. Since Istio does not support mapping incoming request to outgoing requests, we rely on minimal application instrumentation for context propagation of headers from parent to children request and response. We have tested our current prototype on regular expression based policies for enforcing access control.

6 DISCUSSION

Why enforce safety at inter-service interactions? On the one hand, the application layer provides the finest granularity to enforce safety policies as it offers means to control the flow of information at the site of its origin. However, these policies are difficult to manage due to the enforcement being closely tied to the application code. On the other extreme, L4 network layer offers a non-invasive enforcement mechanism with the downside being the coarse policies.

Enforcing safety at inter-service interactions strikes a balance: policies have visibility into inter-service interactions, while treating the services as black-boxes. The policies are high-level enough to describe properties over HTTP requests, while being minimally invasive for the application. Moreover, this decoupling of policy enforcement from application helps eliminate the source of incorrectness from application layer to that of inter-service interactions.

Other potential policy languages. For simplicity, we have used regular expression based language to specify our policies, but there are many potential policy languages that can be compiled and enforced using our strategy. There is likely a tradeoff: richer policy languages could potentially specify more precise safety policies, but possibly require more elaborate mechanisms to enforce. Exploring this tradeoff is an interesting direction for future work.

Managing safety policies. Current single-hop policies can be unwieldy, since in a system with n microservices, operators need to define which of the n^2 possible microservice pairs may interact, which in turn are susceptible to changes in the application architecture. Multi-hop policies are a step toward easy-to-use and maintainable microservice policy systems. Although, multi-hop policies could add additional complexity, they can compactly express goals independent of specific microservices. For instance, external queries should never directly or indirectly access a secure database.

7 RELATED WORK

Safety property enforcement for microservice applications has been explored from the following three directions: by extending the functionalities of common cloud-native solutions, language-based enforcement, and distributed tracing. On the cloud-native front, Istio [6] is a popular service mesh that extends the cloud-native infrastructure to support traffic management policies. However, as described in Section 2, these policies are very coarse.

Trapeze [1] and Fabric [9] are IFC systems for serverless applications and distributed systems respectively, both of which have similarities to microservices. Both these work offer strong security guarantees, and because all application code needs to be checked statically, the code does not need to be trusted (though it does need to be available). However, their implementations are tied to a language-runtime and require application changes and designing the security lattice. Our work addresses these limitations and can handle a class of safety properties broader than IFC properties.

Distributed tracing frameworks, like Jaeger [7] and Zipkin [15] have been used to get visibility into end-to-end application telemetry, service dependencies, and for root-cause analysis. A downside to such frameworks is the instrumentation requirement in the service code for trace collection.

Regular expressions have been used for specifying path invariants in networks [3]. For instance, FatTire [10] and Merlin [12] use regular expressions to specify per-packet forwarding rules for fault tolerance and bandwidth allocation policies. In contrast, NetQRE [14] uses regular expression to filter flows of packet for quantitative network monitoring.

8 FUTURE DIRECTIONS

Parallel/non-deterministic vs sequential API calls. Asynchronous APIs that are commonly used for running background functions add non-determinism to the API call service tree. Intuitively, an occurrence of a non-deterministic API call in a service tree can be seen as spawning an independent service sub-tree. Therefore, instead of having a single sequence of API calls, like with synchronous APIs, we will have a set of such sequence of calls. We see future possibility to explore safety properties on the set of independent service trees generated while using an asynchronous API.

Context propagation. As highlighted in Section 4, currently the application needs to be instrumented and trusted for context propagation. This is a standard requirement not only for our work but also for tracing the execution tree of a request across microservices. It is generally difficult for other entities (e.g., the service mesh's sidecars) to propagate context because the application is a black box and it is not clear which requests entering the application caused the application to generate other requests. Somehow removing the context propagation requirement would reduce the need to trust the application.

Efficiency. Similar to most dynamic enforcement mechanism, policy enforcement at service mesh is on the critical path of the application traffic. Therefore, making more efficient hybrid enforcement methods that can combine dynamic and static enforcement are a possibility.

9 CONCLUSION

This paper describes the subtleties and scope of safety properties in microservice networks. To this end, it proposed the specification for a class of regular expression based properties and its enforcement using service meshes.

Acknowledgments. This work benefited from discussions with Fred Schneider and Rolph Rector. We thank the reviewers for their constructive feedback. This work is supported by NSF grants #2152831 and #2312714.

REFERENCES

[1] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 118, 26 pages. https://doi.org/10.1145/3276488

[2] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (HotNets'21)*. Association for Computing Machinery, New York, NY, USA, 229–236. https://doi.org/10.1145/3484266.3487379

[3] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. 2014. An Assertion Language for Debugging SDN Applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 91–96. https://doi.org/10.1145/2620728.2620743

[4] Cilium. 2023. Cilium. https://cilium.io/. (June 2023). Accessed: 2023-06-30.

[5] Envoy. 2023. Envoy Proxy. https://docs.cilium.io/en/stable/security/network/proxy/envoy/. (June 2023). Accessed: 2023-06-30.

[6] Istio. 2023. Service Mesh. https://istio.io/. (June 2023). Accessed: 2023-06-30.

[7] Jaeger. 2023. Jaeger Tracing. https://www.jaegertracing.io/. (June 2023). Accessed: 2023-06-30.

[8] Kubernetes. 2023. Service Traffic Policy. https://kubernetes.io/docs/concepts/services-networking/service-traffic-policy/. (June 2023). Accessed: 2023-06-30.

[9] Jed Liu, Owen Arden, Michael D. George, Andrew C. Myers, Toby Murray, Andrei Sabelfeld, and Lujo Bauer. 2017. Fabric: Building Open Distributed Systems Securely by Construction. *Journal of Computer Security* 25, 4–5 (Jan 2017), 367–426. https://doi.org/10.3233/JCS-15805

[10] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: Declarative Fault Tolerance for Software-Defined Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. Association for Computing Machinery, New York, NY, USA, 109–114. https://doi.org/10.1145/2491185.2491187

[11] Michael Sipser. 1997. *Introduction to the theory of computation*. PWS Publishing Company.

[12] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/2674005.2674989

[13] Tigera. 2023. Project Calico. https://www.tigera.io/project-calico/. (Oct. 2023). Accessed: 2023-10-22.

[14] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/3098822.3098830

[15] Zipkin. 2023. Zipkin. https://zipkin.io/. (June 2023). Accessed: 2023-06-30.