SafeTree: Expressive Tree Policies for Microservices*

KARUNA GREWAL, Cornell University, USA
BRIGHTEN GODFREY, University of Illinois at Urbana-Champaign, USA
JUSTIN HSU, Cornell University, USA

A microservice-based application is composed of multiple self-contained components called *microservices*, and controlling inter-service communication is important for enforcing safety properties. Presently, inter-service communication is configured using microservice deployment tools. However, such tools only support a limited class of *single-hop* policies, which can be overly permissive because they ignore the rich *service tree* structure of microservice calls. Policies that can express the service tree structure can offer development and security teams more fine-grained control over communication patterns.

To this end, we design an expressive policy language to specify service tree structures, and we develop a *visibly pushdown automata*-based dynamic enforcement mechanism to enforce *service tree* policies. Our technique is non-invasive: it does not require any changes to service implementations, and does not require access to microservice code. To realize our method, we build a runtime monitor on top of a *service mesh*, an emerging network infrastructure layer that can control inter-service communication during deployment. In particular, we employ the programmable network traffic filtering capabilities of Istio, a popular service mesh implementation, to implement an online and distributed monitor. Our experiments show that our monitor can enforce rich safety properties while adding minimal latency overhead on the order of milliseconds.

CCS Concepts: • **Theory of computation → Formal languages and automata theory**; • **Networks → Cloud computing**; **Network monitoring**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Microservices; Servicemesh; Visibly Pushdown Automata.

ACM Reference Format:
Karuna Grewal, Brighten Godfrey, and Justin Hsu. 2025. SafeTree: Expressive Tree Policies for Microservices. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 349 (October 2025), 27 pages. https://doi.org/10.1145/3763127

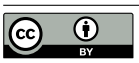
1 Introduction

Large-scale cloud-based applications are often implemented using the *microservice* design paradigm, where the application is decomposed into multiple microservices—that is, services that are loosely coupled and individually have narrowly-defined roles. This design offers separation of concern between the services: individual services can be developed by independent teams, exposing their service's functionality over well-defined API interfaces. Each service runs in isolation in its own runtime-environment called a *container*, listening for incoming traffic on a dedicated port and IP address. Inter-service communication happens over a communication protocol like HTTP or gRPC.

Controlling inter-service communications is important for enforcing safety properties in microservice applications. For example, a service deployment team may want to split the flow of requests to two versions of their service for A/B testing. In a security-critical setting, a team might

*This is the conference version of the paper. We defer technical details and proofs to the full version of the paper [18] .

Authors' Contact Information: Karuna Grewal, Cornell University, Ithaca, USA, kgrewal@cs.cornell.edu; Brighten Godfrey, University of Illinois at Urbana-Champaign, Urbana-Champaign, USA, pbg@illinois.edu; Justin Hsu, Cornell University, Ithaca, USA, justin@cs.cornell.edu.



ACM 2475-1421/2025/10-ART349
https://doi.org/10.1145/3763127

want to restrict communication between certain services to enforce security guidelines of their company. Similarly, an audit or a data-compliance team may want to enforce data-protection regulations, like GDPR [1] and HIPAA [28], by controlling inter-service exchange of information.

1.1 Challenges

Limited expressiveness of existing policies. Today, developers can control inter-service communications with two primary methods: coarse-grained control, and fine-grained control. The first method is exemplified by Kubernetes, the most widely used container orchestration framework, which uses a container network interface (CNI) to control which containers (i.e., services) can communicate with each other. While effective, this kind of policy offers very limited expressivity: communication between services is either fully unrestricted, or entirely prohibited.

The need to offer more fine-grained control of inter-service communications has motivated the use of *service meshes* (e.g., Istio [19]), which handle request-level communications on behalf of services. The data plane of the service mesh (often realized as the Envoy [13] proxy, with one instance of Envoy paired with each service instance) essentially forms a layer between the application and transport protocols, handling incoming and outgoing requests to provide API-level access control, encryption, visibility, load balancing, etc. Service mesh can be used for monitoring inter-service communication at the API call granularity and enforcing policies, for instance, allowing or rejecting a request based on some HTTP source header value.

Both the above methods only support *single-hop* policies, specifying if a pair of endpoints can communicate (e.g., if service A is allows to call a certain API of another service B). However, in a microservice application, a single initial request results in a *service tree* of requests—services make API calls to multiple other services, which in turn call other services. Expressing this service tree structure can offer even more fine-grained control over the communication patterns.

For example, consider a hospital management application where a request for medical test involves interactions between three services: (1) Test, which receives the request; (2) De-identify, which de-identifies patient information; and (3) Lab, which sends the patient records to an external lab. HIPAA's data-protection regulations mandate that to preserve the privacy of a patient, unnecessary personal health information must be de-identified [29]. Therefore, a compliance team might require that the test functionality calls an De-identify service before Lab. However, this property cannot be specified as a single-hop policy between the Test and the Lab services—it requires reasoning about intermediate service interactions between these services. In particular, we need De-identify and Lab services to be invoked by Test, and in that order. Such applications necessitate policies that can specify the structure of a service tree.

Enforcement requirements. Not only are there challenges in expressing rich policies, enforcing policies in this setting is also challenging. For example, safety properties are often specified and maintained by teams, like compliance or deployment, that do not have access to the service code. Therefore, services in a microservice application will often appear as blackboxes to teams, and a policy enforcement mechanism should be *non-invasive*, *i.e.,* not require code changes, and *blackbox*, *i.e.,* not require access to code. Furthermore, the inter-service communication patterns of an application can change due to dynamic updates of service code or due to to elastic scaling of the application, where service containers can spin up or down in response to the load on the application. Thus, the enforcement mechanism should be able to cope with microservice applications that change dynamically, rather than being fixed from the outset.

1.2 Our Approach

In this work, we consider the question: *"How can we specify and enforce policies over the rich service tree structure at runtime without invasive changes to the blackbox service implementation?"*

Our solution consists of three parts: a policy language, an automata-based enforcement mechanism, and a distributed runtime monitor. The black-box and non-invasive aspects of our solution are crucial for usability. For instance, our policies can be fully decoupled from the application code, enabling them to be written and maintained by teams that do not have access to the service code (e.g., deployment or compliance teams). Accordingly, our solution supports polyglot applications.

Expressive policy language for service trees. First, we design a policy language called SafeTree for specifying allowed service tree structure. Our language offers constructs to specify constraints on the children, siblings, and subtrees of a service in the tree. SafeTree also allows multi-hop policy over a linear sequence of API calls, without any reference to the tree structure.

An automata-based enforcement mechanism. Second, we design an automata-based distributed runtime monitor for SafeTree policies. The key idea is that a service tree can be represented as a *nested word* [5], so that policies correspond to sets of nested words. Accordingly, we define a compilation procedure from a policy into a *visibly pushdown automaton* (VPA) [4] that accepts the set of valid nested-words described by the policy. Our VPA-based monitor can be implemented in a fully distributed manner with no need for a centralized authority, by carrying the VPA state in a custom configuration header along with the requests and simulating the VPA transitions locally at the services.

A distributed runtime monitor. Lastly, to implement our policy checking mechanism, we develop a prototype implementation of our monitor on top of the Istio service mesh. In Istio, each service container is paired with a *sidecar* container running an Envoy proxy that can be programmed to specify custom traffic filtering logic involving operations on HTTP headers. We exploit this customizability to implement a runtime monitor locally at services, simulating the VPA in a distributed fashion.

Outline. We first motivate a policy language for service trees (section 2) and introduce background on nested words (section 3). We then introduce our primary technical contributions:

(1) a policy language with constructs to model the service tree structure and a nested-word semantics of the policies (section 4);
(2) a translation of our policies into VPA and a translation of a VPA into a distributed monitor, along with a proof of soundness (section 5);
(3) a broad range of case studies demonstrating real-world policies that can be expressed in our policy language (section 6);
(4) and an implementation and evaluation of an efficient and distributed runtime monitor over service mesh (sections 7 and 8).

Finally, we survey related work (section 9) and conclude with future directions (section 10).

2 A Tour of Service Tree Policies

This section motivates service tree policies in the context of a hospital management application.

2.1 Running Example: Hospital Management Application

Consider a microservice-based hospital management application offering functionalities to request (a) a payment of bills, and (b) a medical test from an external lab. These functionalities are implemented using the following services: **Frontend**, F: requests for an appointment or a medical test; **Test**, T: requests for a lab test; **De-identify**, I: de-identifies personal health information; **Lab**, L: sends patient records to an external lab; **Payment**, P: charges a patient for a service; **Database**, D: writes to a payment database; **EventLog**, E: logs database access events.

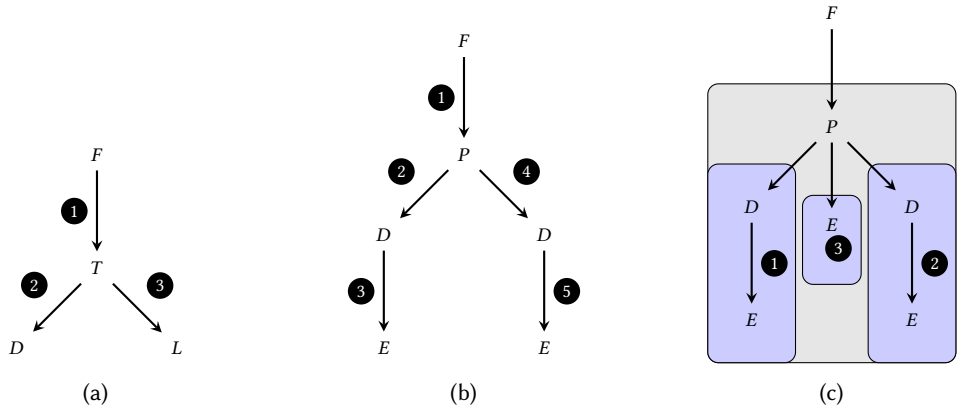


Fig. 1. Service tree for: (a) "lab test" request; (b) "appointment" request; (c) tree with annotated paths.

In a microservice setting, services communicate using API calls over some communication protocol, like HTTP or gRPC, etc. In this paper, we focus on *synchronous* HTTP-based APIs, which follow a request/response pattern. An API request initiates calls to a fleet of services, which may themselves call further services. The runtime execution trace of the application when serving a single request to a service can be viewed as an ordered *service tree* rooted at a node corresponding to the requested API. Each API invoked in a single execution trace is represented as a node in the tree; an edge from a parent to a child node implies the parent endpoint called the child endpoint; the children of a node are ordered according to the order in which they are called by the parent. Since this runtime behavior can depend on arguments to the call, responses from child calls, local state at the microservice, etc., a call to a given API might lead to multiple possible service trees.

To illustrate, let us consider possible service trees for two Frontend functionalities: requesting a medical test, or requesting a payment.

(1) ***Medical test functionality***: the execution trace when Frontend sends a request to Test is represented as a tree in fig. 1a. First, the Frontend invokes Test: this is represented as the edge ❶ between a node labeled with Frontend and Test. Further API calls invoked by Test while serving this incoming request are represented as outgoing edges from this Test node. In this case, Test first calls ❷ De-identify and then Test calls ❸ Lab. The left-to-right ordering of the children of Test in this tree reflects the order of children calls.

(2) ***Payment functionality***: the service tree in fig. 1b describes the execution trace when Frontend requests Payment to charge for two bills: first, ❶ Frontend invokes Payment. This service then invokes ❷ Database, which then invokes ❸ EventLog. Then Payment invokes ❹ Database for the second time, before Database invokes ❺ EventLog.

2.2 Safety Properties

In our example application, we can imagine several safety policies motivated by regulatory, business, and security concerns.

(1) ***Deployment team: A/B testing.*** Suppose there are two versions of Payment and EventLog services, v1 and v2. For A/B testing, a deployment team may want Payment requests from beta testers to be served by v2 of the Payment service and any direct or indirect EventLog invoked by v2 of Payment should also be of version v2. The labeling of beta users is specified

in the Frontend service, while the services undergoing A/B testing can be multiple hops deeper in the call tree.

(2) ***Security team: Log database access.*** If Payment service accesses the Database (as shown in fig. 1b) to update patient's payment details, this sensitive update should be logged by Database calling EventLog.

(3) ***Data compliance team: HIPAA compliance.*** In compliance with the HIPAA guidelines to protect patient privacy from an external lab, the hospital's data-compliance team might want to ensure that a request to Test service (as shown in fig. 1a) should be processed by first calling De-identify and then calling the external Lab .

These safety properties are often specified and maintained by teams that do not have direct access to the service code. Therefore, services in the application will often appear as blackboxes to teams, who may only have visibility into an application's execution by observing inter-service communication patterns. Since examining application code is outside the scope of service meshes, we can only specify policies at the granularity of inter-service communication. For example, the data compliance policy above may not be able to enforce the specific data that is passed from De-identify to Lab.

Existing microservice frameworks allow control over inter-service communication via *single-hop* constraints, which control calls between single pairs of services. However, the above policies cannot be directly expressed in terms of such policies since they involve constraints on the service tree structure, e.g., descendants, subtrees, etc. Thus, we need a policy language and enforcement mechanism that is more expressive than current CNI and service mesh policies.

2.3 Service Tree Policies

To address the above issue, we design SafeTree—a policy language for specifying service trees to increase the expressiveness of the policies that can be enforced strictly using the inter-service communication information available at the service mesh layer. To get a flavor of our language, let us specify the payment database logging policy (2), which requires that any requests from Payment to Database must call EventLog. In our language, this policy can be specified as:

$$\underbrace{\textbf{match}\ \text{Payment}}_{\text{1. root}} \overset{\forall\text{-path}}{\Longrightarrow} \underbrace{(\text{Database EventLog \_}) + (!\text{Database})^*}_{\text{2. path regex}}$$

For brevity, we defer some syntactic details to section 4. For now, we will step through the above specification to understand how different parts of the policy have been expressed:

- Since the property is described with respect to a subtree rooted at Payment (as marked in the tree in fig. 1c), we need to express the root of the subtree. This is specified as the expression "**match** Payment" on the left.
- The condition that Database invoked by Payment should always invoke EventLog is described in terms of a regular expression over the application endpoints. This is specified as the right-hand side expression (Database EventLog _) + (!Database)*. Here, _ denotes any sequence of the application endpoints and !Database denotes any endpoint besides Database. This regular expression describes the valid sequence of API calls in a path from Payment's children to any leaf node. For instance, in the tree in fig. 1c, this regular expression matches the paths labeled with ❶, ❷, and ❸.

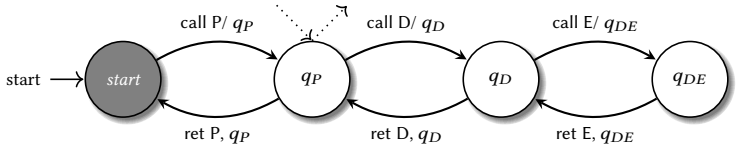


Fig. 2. Visibly pushdown automaton, which accepts a tree where Payment (P) invokes Database (D) as its child, and this D invokes EventLog (E) as its children. Omitted transitions are marked in dotted. The initial and final state is *start*.

2.4 Policy Enforcement

The blackbox treatment of services calls for a non-invasive enforcement mechanism that does not require code modification. In our work, the service tree policies are enforced using a distributed runtime monitor based on *visibly pushdown automata* (VPA) [4].

Overview: visibly pushdown automata. A VPA is a restricted type of pushdown automaton where the stack operation is determined by the symbol being read. For instance, the VPA in fig. 2 has transitions with two kinds of labels: (a) "call *service*/ *push*" and (b) "ret *service*, *pop*", where *service* is the name of the service endpoint and *push* and *pop* are values to be pushed and popped on the stack respectively. A *service* symbol tagged with "call" corresponds to an HTTP request to *service*, and similarly "ret" tags HTTP responses from *service*. Intuitively, the VPA reads a string over call/ret tagged symbols, say "call P; call D; . . . ; ret P", from left to right. On a "call" symbol, the VPA pushes a value on the stack wand transitioning to the next state; on a ret symbol, the VPA pops the top of the stack and moves to the next state. For instance, in fig. 2, the edge labeled "call P/ q_P" between *start* and q_P represents that upon reading call P at *start* state, q_P is pushed on stack and the VPA transitions to state q_P. Similarly, the transition at state q_P on symbol "ret P" in fig. 2 indicates that the next state will be *start* if top of the stack is q_P while making that transition.

Service tree—a nested-word view. By viewing each node in the service tree as a call and return to/from the labeled endpoint, the service tree can be modeled as a sequence of nested calls and returns forming *nested word* [5]. For instance, each node in fig. 3a corresponds to a call and ret symbol in its nested-word view described in fig. 3b. The call and ret symbol of a child call is nested between its parent's call and ret symbols because a synchronous API waits for the response from any API that it invokes before resuming its execution. For instance, in fig. 3b, the call and ret of the two D in fig. 3a are nested between the call and ret symbols of their parent P.

By viewing service trees as nested words, we can view policies as sets of allowed nested words. By carefully designing our policy language, we can ensure that our policies correspond to languages accepted by VPA, which can check if a given service tree is permitted. For instance, the service tree in fig. 3a is accepted by the VPA in fig. 2 because the VPA accepts the nested word view (described in fig. 3b) of the service tree. After starting the VPA run at the initial state, *start*, the VPA arrives at the final state *start*, which is an accepting state.

Runtime policy checking. To check service tree policies, we develop a compiler to translate SafeTree policies into a VPA that accepts the valid set of nested words, and then we extract a distributed runtime monitor from this VPA. The motivation behind a distributed design is the lack of a global view of service trees at services. Section 5 and section 7 will detail the design principle behind the distributed monitoring, but the key is that instead of a centralized monitor, we employ local sub-monitors at services to simulate VPA transitions on their respective HTTP requests/responses. To facilitate local VPA simulation while giving the illusion of having a centralized VPA, the current VPA configuration is carried in a custom HTTP header.

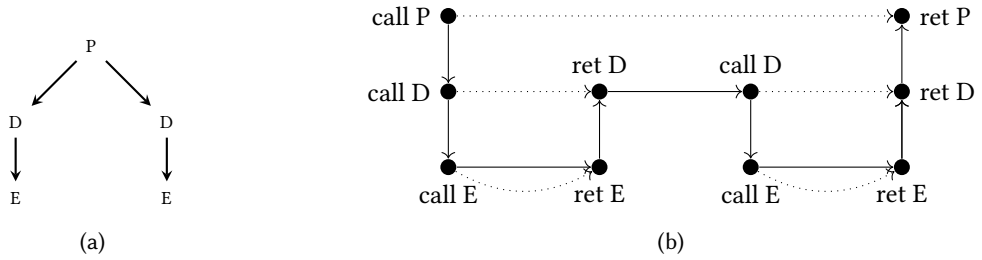


Fig. 3. Service tree for an appointment request: (a) tree view; (b) nested-word view.

To implement this distributed monitor without invasive changes to the service implementation, we build on top of Istio, a popular service mesh implementation [19]. Service mesh is a network infrastructure layer that abstracts away inter-service communication logic from the service implementation and offloads it into a sidecar container. Istio pairs each service with a sidecar to manage the service's traffic using the Envoy proxy [13] running in the container. The proxy intercepts all incoming/outgoing HTTP traffic into the service container. The Envoy proxy can be programmed with custom logic to perform reads and writes to HTTP headers and control access to the service. Section 7 will detail how we use this programmability to simulate VPA transitions in Envoy.

3 Background: Service Tree as a Nested-Word

We give a refresher on the standard definition of nested words [5] and introduce some custom notations that will be useful in later sections for encoding service trees as these nested words.

3.1 Nested Words Refresher

We start by recalling basic concepts and notation for nested words.

Definition 3.1 (Call-Return Augmented Alphabet). $\Sigma = \Sigma_c \cup \Sigma_r$ is a *call-return augmented alphabet* for some $\tilde{\Sigma}$ if $\Sigma_c = \{\langle s \mid s \in \tilde{\Sigma}\}$, and $\Sigma_r = \{s\rangle \mid s \in \tilde{\Sigma}\}$.

Intuitively, $\langle s$ and $s\rangle$ correspond to an HTTP request or call to an endpoint s and an HTTP response from an endpoint s respectively.

Definition 3.2 (Indexed Symbols). An *indexed symbol* $a = (e, i)$ is a pair of some symbol $e \in \Sigma$ and an index $i = \iota(a) \in \mathbb{N}^+$, where ι is the index projection. Here, Σ is a call-return augmented alphabet for some $\tilde{\Sigma}$. The symbol a is said to be a *call symbol* if $e \in \Sigma_c$, and a *return symbol* if $e \in \Sigma_r$. We write $C(a)$ when a is a call symbol and $R(a)$ when a is a return symbol.

Definition 3.3 (Nested Word). A nested word $n = (w, \nu)$ over $\Sigma = \Sigma_c \cup \Sigma_r$ (a call-return augmented alphabet for some $\tilde{\Sigma} \ni s$) is a pair of: a word $w = a_1 \dots a_l = (a_i)_{1 \le i \le l}$ such that $\iota(a_1) = k$ for some $k \in \mathbb{N}^+$ and $\iota(a_i) + 1 = \iota(a_{i+1})$ for any $1 \le i < l$; and a *matching relation* $\nu \subseteq \{-\infty, k, \dots, k+l-1\} \times \{k, \dots, k+l-1, +\infty\}$ that associates call symbols with corresponding return symbols and satisfies:

(1) *Each symbol occurs in only one pair.* For any $a_c = (\langle s, i) \in w$, there exists a unique $j \in \{k, \dots, k+l-1, +\infty\}$ such that $\nu(i,\ j)$ and if $j \neq +\infty$ then there exists a return symbol $a_r \in w$ with index $\iota(a_r) = j$. For any $a_r = (s\rangle, j) \in w$, there is a unique $i \in \{-\infty, k, \dots, k+l-1\}$ such that $\nu(i,\ j)$ and if $i \neq -\infty$ then there exists a call symbol $a_c \in w$ with index $\iota(a_c) = i$.
(2) *Edges go forward.* If $\nu(i,\ j)$ then $i < j$.
(3) *Edges do not cross.* If $\nu(i,\ j)$, $\nu(i',\ j')$ and $i < i'$ then either the two edges are well-nested (*i.e.,* $j' < j$) or disjoint (*i.e.,* $j < i'$).

Let $c \in \Sigma_c$ and $r \in \Sigma_r$. A nested word is *well-matched* if for any $a_c = (c, i) \in w$, there exists an $a_r = (r, j) \in w$ such that $v(i,\ j)$; and for any $a_r = (r, j) \in w$, there exists an $a_c = (c, i) \in w$ such that $v(i,\ j)$. Here, a_i is a call whose *matched return* is a_j.

Example 3.4. Fig. 4 formally describes the well-matched nested word representation for the service tree in fig. 3a. Here, $\tilde{\Sigma} = \{\mathsf{P}, \mathsf{D}, \ldots\}$ is the set of all endpoint names, and the call-return augmented alphabet for $\tilde{\Sigma}$ is $\Sigma = \{\langle \mathsf{P},\ \langle \mathsf{D},\ \ldots\} \cup \{\mathsf{P}\rangle, \mathsf{D}\rangle,\ \ldots\}$. In the nested word representation, each API call in the service tree will be unfolded into a matched call and return pair. In this case, the nested word is $n = (a_1 a_2 a_3 a_4 a_6 a_7 a_8 a_9 a_{10}, v)$, where $a_1, \ldots, a_{10}$ are indexed symbols; we will sometimes call the first symbol a_1 the *root* of the nested word, since it is root of the service tree.

The matching relation v is described using the dotted lines. For instance, the first call to the Payment service or a_2 is matched with the return symbol a_5. Note that the first and the last symbol in the nested word, *i.e.,* a_1 and a_{10} respectively form a matched call-return pair; due to the inherent request/response protocol, nested words corresponding to service trees in a microservice application will always be of this form.

A nested word n can also be sliced into smaller nested words corresponding to various subtrees of the service tree.

Definition 3.5 (A sub-nested word). Given $n = (a_1 \ldots a_l, v)$ and $\iota(a_1) \leq i < i' \leq \iota(a_l)$, let $a_j, a_{j'} \in n$ be such that $\iota(a_j) = i$, and $\iota(a_{j'}) = i'$. We define a sub-nested word $n[i, i'] = (a_j \ldots a_{j'}, v[i, i'])$. Here, $v[i, i']$ is the restricted matching sub-relation:

$$v[i, i'] = \{v(p,\ q) \mid i \leq p, q \leq i'\} \cup \{(p, +\infty) \mid v(p,\ q),\ i \leq p \leq i',\ q > i'\}$$
$$\cup \{(-\infty, q) \mid v(p,\ q),\ i \leq q \leq i',\ p < i\}$$

Example 3.6. Consider the service tree in fig. 3a. Its first subtree rooted at D consists of the first D and first E. In terms of the nested word (in fig. 4) for this service tree, the subtree under consideration is given by the sub-nested word $n[2, 5] = (a_2 a_3 a_4 a_5, v')$, where $v' = \{(2, 5),\ (3, 4)\}$.

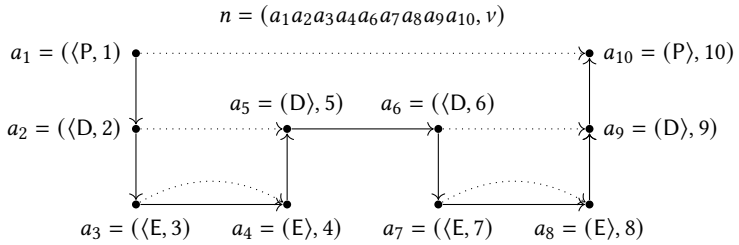


Fig. 4. Well-matched nested word for the service tree in fig. 3a. The dotted line from a node a_i to a_j represents the match relation $v(\iota(a_i),\ \iota(a_j))$. The projection on the second element of any indexed symbol a_i is $\iota(a_i)$.

3.2 Service Tree Concepts

To define our policies, we first define nested-word equivalents for different parts of a service tree.

First, we define a set of words that correspond to all paths from the root of the service tree representation of a nested word. Well-matched nested words are said to be *rooted* if the first and the last symbol are matched.

Definition 3.7 (Set of Paths). For a rooted well-matched nested word $n = (a_1 \dots a_l, \nu)$ over some Σ, the set of all *call sequences* $Seq(n)$ from the root is given by:

$$Seq(n) = \{\, \pi(n, a_m) \mid C(a_m),\ a_m \in n \,\},$$

where $\pi(n, a_m)$ is a word $w = (a_i)_{i \in I}$ and

$$I = \{i \mid a_i \in n,\ \iota(a_i) \leq \iota(a_m),\ C(a_i),\ \nu(\iota(a_i),\ j),\ j > \iota(a_m)\}.$$

The word w is a *path* from the root to some subsequent call $a_m \in n$.

Example 3.8. In the running example in fig. 4, the set of all paths in the nested word n is given by $Seq(n) = \{a_1a_2,\ a_1a_2a_3,\ a_1a_6a_7,\ a_1a_6\}$. A leaf node is some call symbol that is immediately followed by a return symbol. The set of all path to leaves in n is given by $SeqLeaf(n) = \{a_1a_2a_3,\ a_1a_6a_7\}$.

To reason about the children of a node in the tree, we define a set of symbols corresponding to children of a nested word's root:

Definition 3.9 (Child of a root). For a rooted well-matched nested word $n = (a_1 \dots a_l, \nu)$, a call $a_m \in n$ is a *child* of root a_1 if $a_1a_m \in Seq(n)$. The set of children $Child(n)$ is given by:

$$Child(n) = \left\{\, a_m \;\middle|\; a_1a_m \in Seq(n) \,\right\}$$

In the nested word n in fig. 4, symbols a_2 and a_6 are the children of the node a_1. Similarly, a_3 is the child of a_2. The sub-nested words $n[2, 5]$ and $n[6, 9]$ form a set of sub-trees rooted at a_1's children. We formally define the set of *child subtrees of the root* of a nested word.

Definition 3.10 (Child subtree of a root). Given a rooted well-matched nested word $n = (a_1 \dots a_l, \nu)$ with a_m as one of the children of a_1 and some $x \in \mathbb{N}^+$ such that $\nu(\iota(a_m),\ x)$, a *subtree* rooted at a_m is a rooted well-matched nested word $n' = ((a_j)_{j \in I}, \nu_m)$, where $I = \{\iota(a_m), \dots, x\}$ and $\nu_m = \nu[\iota(a_m), x]$. The set of subtrees is given by:

$$Subtree(n) = \left\{ ((a_j)_{j \in I}, \nu_m) \;\middle|\; \begin{array}{l} a_m \in Child(n), \\ \exists\, x \in \mathbb{N}^+ \text{ such that } \nu(\iota(a_m),\ x) \end{array} \right\}$$

4 SafeTree: A Service Tree Policy Language

Now that we have introduced notation for nested words, we describe our policy language SafeTree, and define an interpretation as sets of nested words.

4.1 Syntax

The policy syntax is presented in fig. 5. SafeTree policies make extensive use of regular expressions (*reg*) defined over the set of endpoints $\tilde{\Sigma}$.

Top-level SafeTree policies are of the form **start** S : *inner*, where $S \subseteq \tilde{\Sigma}$, and *inner* can be two kinds of sub-policies: *p* over the hierarchical structure or *seq* over the linear structure of the tree. Intuitively, top-level policies specifies that "the *inner* sub-policy is satisfied on any subtree rooted at an endpoint in S that has no ancestor in S". For instance, if $S = \{\mathrm{A}\}$, *inner* should be satisfied on all subtrees whenever A is first encountered. We write **start** $\star$: *inner* as convenient notation for **start** $\tilde{\Sigma}$: *p*; this policy specifies that *inner* should be satisfied at the root of the entire tree.

Inner: Hierarchical Policies. After specifying the starting symbol of a policy, a SafeTree policy needs to specify a sub-policy to express constraints on hierarchical or the linear structure of a service tree. All hierarchical policies shown in fig. 5 have "**match** reg" expression to the left of the annotated $\Longrightarrow$ symbol. This expression specifies that there exists a path from the root of the service tree to some descendant a_i such that the path matches *reg*. (Note that *reg* in the **match** expression should not match the empty string.) We will step through the syntax of hierarchical

Regular Expressions

$a \in \tilde{\Sigma}$

$S \subseteq \tilde{\Sigma}$

$reg ::= a \mid \epsilon \mid \emptyset \mid reg_1 + reg_2 \mid reg_1 reg_2 \mid reg^*$

Service Tree policy

$policy ::= \textbf{start}\ S : p \mid \textbf{start}\ S : seq$

Hierarchical Policy

$p_l ::= \textbf{match}\ reg_1 \overset{\forall\textbf{-path}}{\Longrightarrow} reg_2$

$p_a ::= \textbf{match}\ reg \overset{\forall\textbf{-child}}{\Longrightarrow} p$

$p_e ::= \textbf{match}\ reg \overset{\exists\textbf{-child}}{\Longrightarrow} p_1\ \textbf{then} \ldots \textbf{then}\ p_k$

$p ::= p_l \mid p_a \mid p_e$

Linear Sequence Policy

$seq ::= \textbf{call-sequence}\ reg$

Fig. 5. SafeTree syntax

policies to understand the purpose of different right side expressions in these policies in terms of the service tree.

The simplest policy is $\textbf{match}\ reg_1 \overset{\forall\textbf{-path}}{\Longrightarrow} reg_2$, where reg_1, reg_2 are regular expressions. It expresses the existence of a path from the root to some descendant a_i that matches reg_1. Also, all the paths from any of a_i's children to the leaves in the tree match reg_2. The path between the tree's root and a_i should be the *shortest* match of reg_1, *i.e.,* there should be no prefix of the path that matches reg_1.

Example. Consider the service tree previously defined in fig. 3a. Suppose _ is a syntactic sugar for the regular expression notation for $\tilde{\Sigma}^*$. This tree satisfies the policy $\textbf{match}\ \text{PD}^* \overset{\forall\textbf{-path}}{\Longrightarrow} \_$, which requires existence of a path from the root of the tree, *i.e.,* P that matches exactly one P and any number of D. If we consider P as both the root and the descendant a_i, we get a path of length one that satisfies the requirement. The two paths DE from the children of P satisfy the regular expression _ on the right. Therefore, the service tree satisfies the policy. Although there are more paths, like PD in the tree that could have matched the regular expression on the left, we care about the path which does not have a prefix in the language of the regular expression PD^*.

To specify that all subtrees of some node in a tree should satisfy a policy p, we can use the policy $\textbf{match}\ reg \overset{\forall\textbf{-child}}{\Longrightarrow} p$. This policy specifies that there exists a node a_i such that the path from the root of the tree to a_i matches reg and all subtrees rooted at children of a_i satisfy policy p. This policy specifies a universal condition on a node's subtrees.

To specify existential conditions on the subtrees, the policy $\textbf{match}\ reg \overset{\exists\textbf{-child}}{\Longrightarrow} p_1\ \textbf{then} \ldots \textbf{then}\ p_k$ can be used. It specifies that there exists a node a_i such that the path from the root of the tree to a_i matches reg. Also, a_i has a subtree that satisfies p_1 followed by another subtree somewhere after it that satisfies p_2, and so on till p_k. Suppose c_i and c_{i+1} are the root node of the subtrees matching p_i and p_{i+1} respectively, where $1 \leq i < k$. Let C be the parent of c_1 and c_2. The node C can have children older than c_i and younger than c_{i+1}.

Inner: Linear Sequence Policy. SafeTree offers a **call-sequence** construct to specify the desired sequence of API calls as a regular expression, while disregarding the hierarchical tree structure. For instance, in a tree starting at P, say, we want to express that D happens after P, without any specific details about the subtrees or paths in the tree where D should occur. This will be specified

as **call-sequence** P_D_, where _ in the regular expression denotes any sequence of symbols. This policy specifies that the depth-first traversal of the tree should match the given regular expression.

$$\boxed{policy = \textbf{start}\ S : seq}$$

$$\mathsf{NW}[\![policy]\!] = \left\{ n = (a_1 \dots a_l, v) \;\middle|\; \begin{array}{l} \forall\, \pi(n, a_i) = a_1 \dots a_i \in Seq(n), \\ \text{if } \pi(n, a_i) \in FirstMatch(n, (\tilde{\Sigma})^* S) \text{ then} \\ \exists\, x \text{ such that} \\ v(\iota(a_i),\ x) \text{ and } n[\iota(a_i), x] \in \mathsf{Sequence}[\![seq]\!] \end{array} \right\}$$

$$\boxed{policy = \textbf{start}\ S : p}$$

$$\mathsf{NW}[\![policy]\!] = \left\{ n = (a_1 \dots a_l, v) \;\middle|\; \begin{array}{l} \forall\, \pi(n, a_i) = a_1 \dots a_i \in Seq(n), \\ \text{if } \pi(n, a_i) \in FirstMatch(n, (\tilde{\Sigma})^* S) \text{ then} \\ \exists\, x \text{ such that } v(\iota(a_i),\ x) \text{ and } n[\iota(a_i), x] \in \mathsf{Tree}[\![p]\!] \end{array} \right\}$$

$$\boxed{p_l = \textbf{match}\ reg_1 \overset{\forall\textbf{-path}}{\Longrightarrow} reg_2}$$

$$\mathsf{Tree}[\![p_l]\!] = \left\{ n = (a_1 \dots a_l, v) \;\middle|\; \begin{array}{l} \exists\, \pi(n, a_i \in n) = a_1 \dots a_i \in Seq(n) \text{ such that} \\ \pi(n, a_i) \in FirstMatch(n, reg_1), \\ \exists\, x \text{ such that } v(\iota(a_i),\ x) \text{ and} \\ \quad \forall n_s \in Subtree(n[\iota(a_i), x]), \\ \quad\quad \forall \pi' \in SeqLeaf(n_s),\ Calls(\pi') \in \mathcal{L}(reg_2) \end{array} \right\}$$

$$\boxed{p_a = \textbf{match}\ reg \overset{\forall\textbf{-child}}{\Longrightarrow} p}$$

$$\mathsf{Tree}[\![p_a]\!] = \left\{ n = (a_1 \dots a_l, v) \;\middle|\; \begin{array}{l} \exists\, \pi(n, a_i \in n) = a_1 \dots a_i \in Seq(n) \text{ such that} \\ \pi(n, a_i) \in FirstMatch(n, reg), \\ \exists\, x \text{ such that } v(\iota(a_i),\ x) \text{ and } Subtree(n[\iota(a_i), x]) \subseteq \mathsf{Tree}[\![p]\!] \end{array} \right\}$$

$$\boxed{p_e = \textbf{match}\ reg \overset{\exists\textbf{-child}}{\Longrightarrow} p_1\ \textbf{then} \dots \textbf{then}\ p_k}$$

$$\mathsf{Tree}[\![p_e]\!] = \left\{ n = (a_1 \dots a_l, v) \;\middle|\; \begin{array}{l} \exists\, \pi(n, a_i \in n) = a_1 \dots a_i \in Seq(n) \text{ such that} \\ \pi(n, a_i) \in FirstMatch(n, reg), \\ \exists\, x \text{ such that } v(\iota(a_i),\ x) \text{ and} \\ \quad \exists\, \{t_1, \dots, t_k\} \subseteq Subtree(n[\iota(a_i), x]) \text{ such that} \\ \quad\quad \text{for any } 1 \le j < k,\ \iota(a_1^j) < \iota(a_1^{j+1}), \text{and} \\ \quad\quad \text{for any } 1 \le j \le k,\ t_j \in \mathsf{Tree}[\![p_j]\!], \\ \text{where } \forall 1 \le j \le k, t_y = (a_1^j \dots, v^j) \end{array} \right\}$$

$$\boxed{seq = \textbf{call-sequence}\ reg}$$

$$\mathsf{Sequence}[\![seq]\!] = \{n = (a_1 \dots a_l, v) \mid Calls((a_x)_{x \in I}) \in \mathcal{L}(reg),\ \text{where } I = \{\iota(a_j) \mid C(a_j),\ a_j \in n\}\}$$

Fig. 6. SafeTree semantics (where *Subtree* is defined in definition 3.10)

4.2 Nested Word Interpretation for Service Trees

Formally, we interpret a SafeTree policy as a set of rooted well-matched nested word. Since matching paths with regular expressions is common across SafeTree policies, we define a nested word variant for a set of shortest paths matching some regular expression.

Definition 4.1 (First or Shortest Match Path). Given a nested word $n = (a_1 \dots a_l, v)$ over Σ, which is a call-return augmented alphabet for some $\tilde{\Sigma}$, a path $\pi(n, a_m) \in Seq(n)$ is the *first match* for some regular expression reg over $\tilde{\Sigma}$ (or $\pi(n, a_m) \in FirstMatch(n, reg)$) if $w = Calls(\pi(n, a_m))$ is matched by reg, and there is no smaller prefix of w that matches reg.

$$FirstMatch(n, reg) = \left\{ \pi(n, a_m) \in Seq(n) \;\middle|\; \begin{array}{l} \text{for all } 1 \le j < m, \text{ the following holds} \\ s_1 \dots s_j \notin \mathcal{L}(reg),\ w \in \mathcal{L}(reg), \\ \text{where } w = Calls(\pi(n, a_m)) = s_1 \dots s_m \end{array} \right\}$$

Here, $Calls$ of a path rewrites every augmented call symbol with its counterpart in $\tilde{\Sigma}$.

Definition 4.2 (Calls Projection of a Path). Given a nested word $n = (a_1 \dots a_l, v)$ over base alphabet $\tilde{\Sigma}$ and a path $\pi(n, a_m) = a_1 \dots a_m$, we define

$$Calls(a_1 \dots a_m) = s_1 \dots s_m, \text{ where } s_i = \text{proj}(a_i) \text{ for any } 1 \le i \le m.$$

Here, proj(a) projects the base symbol of an indexed symbol, *i.e.,* for any $a = (\langle s, i)$ or $a = (s\rangle, i)$ (with some index i), we define proj(a) $= s$.

The nested word interpretation $\mathsf{NW}[\![p]\!] \subseteq NestedWords(\Sigma)$ for a service tree policiy p is defined in fig. 6. Here, $NestedWords(\Sigma)$ is the set of all nested words over Σ. For a nested word $n = (a_1 \dots a_l, v)$ to be accepted by a service tree policy **start** S : *inner*, the *inner* policy should be satisfied on every sub-nested word of n that is rooted at some symbol $a_i \in S$ and the path from the root of the nested word to the symbol a_i should contains no symbol in S besides a_i. In the $\mathsf{NW}[\![p]\!]$ definition in fig. 6, this condition is formally expressed by requiring the path between the root and a_i to be a first match of regular expression $(\tilde{\Sigma})^* S$.

The nested word interpretation $\mathsf{Tree}[\![p]\!] \subseteq NestedWords(\Sigma)$ in fig. 6 for a hierarchical policy first defines an existential constraint on a path from the root to some node, followed by specific constraints on either path until the leaves, or on sub-nested words (or subtrees). As defined in fig. 6, a linear policy accepts a nested word $n \subseteq \mathsf{Sequence}[\![p]\!]$ if the sequence of call symbols in n match the given regular expression in p.

5 Enforcement

The SafeTree policies are enforced by a visibly pushdown automaton (VPA)-based monitor. For this, we define a compiler from our policy language to a VPA. The VPA is then used to check if a service tree is valid. Before we detail our VPA-based enforcement mechanism, we first give a refresher on the standard VPA model:

Definition 5.1 (Visibly pushdown automaton). A (deterministic) *visibly pushdown automaton* (VPA) is defined as $\mathcal{M} = (Q, q_{init}, F, \Sigma, \Gamma, \bot, \delta_c, \delta_r)$, where:

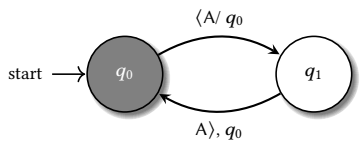
- Q is the set of all states, $q_{init} \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states,
- Σ is the alphabet, where $\Sigma = \Sigma_c \cup \Sigma_r$ consists of call symbols Σ_c and return symbols Σ_r,
- Γ is the set of stack symbols with a special bottom of stack symbol $\bot \in \Gamma$,
- $\delta_c^p : Q \times \Sigma_c \to Q \times (\Gamma - \{\bot\})$ is the call transition function,
- $\delta_r^p : Q \times (\Gamma - \{\bot\}) \times \Sigma_r \to Q$ is the return transition function.

Note that we do not need the conventional *internal symbols* of a VPA to monitor our policies; extending our policies to use internal symbols is an interesting avenue for future work.

Example 5.2. Fig. 7 shows a two state VPA, which is defined over $\Sigma = \{\langle \mathsf{A}, \mathsf{A}\rangle\}$. In this case, $Q = \{q_0, q_1\}$, $q_{init} = q_0$, $F = \{q_0\}$, and $\Gamma = \{\bot,\ q_0\}$.

A VPA configuration (q, θ) is a pair of its current state q and the current stack θ. The stack θ is a sequence of stack symbols with only one occurrence of $\bot$ at the starting. For instance, a possible

configuration for the VPA in fig. 7 can be $(q_1, \perp q_0)$. This configuration denotes that the VPA is at state q_1 and its stack has only q_0.



Nested word: $n = (a_1 a_2, \nu)$, where
$a_1 = (\langle \mathrm{A}, 1), \ a_2 = (\mathrm{A}\rangle, 2)$, and
$\nu = \{(1, 2)\}$

Run: $(q_0, \perp) \xrightarrow{\langle \mathrm{A}} (q_1, \perp q_0) \xrightarrow{\mathrm{A}\rangle} (q_0, \perp)$

Fig. 7. (a) A two state VPA over $\Sigma = \{\langle \mathrm{A}, \mathrm{A}\rangle\}$ with $Q = \{q_0, \ q_1\}$, $q_{init} = q_0$, $F = \{q_0\}$, and $\Gamma = \{\perp, q_0\}$; (b) Run of the VPA on $n = (a_1 a_2, \nu)$.

5.1 Semantics of a VPA

The run $\rho(w) = (q_1, \theta_1) \ldots (q_k, \theta_k)$ of a VPA $\mathcal{M}$ on some nested word w is a sequence of configurations. Nested word w is accepted by $\mathcal{M}$ if q_n is a final state of $\mathcal{M}$. The nested word $w \in \mathcal{L}(\mathcal{M})$ is in the language of $\mathcal{M}$ if w is accepted by $\mathcal{M}$. Intuitively, when a VPA $\mathcal{M}$ transitions from one configuration to another on a call symbol, say $a \in \Sigma_c$, it pushes a value on the stack and moves to the next state. While transitioning on a return symbol $a \in \Sigma_r$, the top of the stack is popped and the configuration state is updated.

For example in fig. 7, the transition $\langle \mathrm{A}/q_0$ is a call transition that pushes q_0 on the stack upon reading the symbol $\langle \mathrm{A}$. The transition labeled with $\mathrm{A}\rangle, q_0$ is a return transition on $\mathrm{A}\rangle$ and it pops the top of the stack q_0.

The valid transitions allowed by a VPA can be defined as follows:

Definition 5.3. Let $\mathcal{M} = (Q, q_{init}, F, \Sigma, \Gamma, \perp, \delta_c, \delta_r)$ be a deterministic VPA with states Q, $\Sigma = \Sigma_c \cup \Sigma_r$. Let μ be the set of all stacks and a be some symbol in Σ then $\xrightarrow{a}: Q \times \mu \to Q \times \mu$ is defined as follows:

(1) if $a \in \Sigma_c$ then $(q', \theta') \xrightarrow{a} (q, \theta)$ if there exists $(q', a, q, s) \in \delta_c$, where $s \in \Gamma$ and $\theta = \theta' s$,
(2) if $a \in \Sigma_r$ then $(q', \theta') \xrightarrow{a} (q, \theta)$ if there exists $(q', s, a, q) \in \delta_r$, where $s \in \Gamma$ and $\theta s = \theta'$.

Example 5.4. Let us look at the run of the VPA in fig. 7 on the nested word $n = ((\langle \mathrm{A}, 1)(\mathrm{A}\rangle, 2), \nu)$, where $\nu = \{(1, 2)\}$. As shown in the figure, on the first symbol, the VPA goes to state q_1 and pushes q_0 on stack. On the next symbol, the top of the stack is q_0, so the VPA goes from the state q_1 to q_0 and the stack value q_0 is popped.

5.2 Compilation Sketch

We define a VPA interpretation $\mathrm{VPA}[\![.]\!] : \mathit{Policy} \to \mathit{VPA}$ for SafeTree policies. Here, *Policy* is the set of all policies and *VPA* is the set of all VPAs. Here, we provide a sketch of the compilation; the detailed rules and further discussion can be found in the full paper. Below, we write $\mathcal{M}$ for the target policy's VPA. We first consider sequential policies.

VPA of call-sequence *reg.* $\mathcal{M}$ simulates *reg*'s DFA $\mathcal{A}$ on all call symbols and ignores the return symbols. The nested word is accepted if $\mathcal{A}$ accepts the sequence of calls.

We now turn to hierarchical policies, which are of the form "**match** $reg \overset{\cdots}{\Longrightarrow} inner$". For such policies, $\mathcal{M}$ first simulates *reg*'s DFA $\mathcal{A}$ (on call symbols) to find a path that matches *reg*. During this phase, $\mathcal{M}$ maintains a stack of $\mathcal{A}$'s run on the path. Suppose the above path ends with some symbol $\langle s$, $\mathcal{M}$ checks if *inner* is satisfied on the subtree rooted at $\langle s$. If *inner* is not satisfied, the policy can still be satisfied if there exists another path from the root that matches *reg* leading to a

subtree satisfying *inner*. To implement this behavior, the following *retry steps* are taken: on return symbols, $\mathcal{M}$ backtracks $\mathcal{A}$'s run using the stack history of the run; $\mathcal{M}$ searches for another path by simulating $\mathcal{A}$ to go forward on call symbols; and then running the checks for *inner*.

Now, we detail the other steps of policy checking.

VPA of match $reg \overset{\forall\text{-}\textbf{path}}{\Longrightarrow} reg'$. While reading the paths in the subtree rooted at $\langle s$, $\mathcal{M}$ simulates the DFA $\mathcal{A}'$ of reg' on the call symbols; and on return symbols, backtracks $\mathcal{A}$'s run using the stack history. If $\mathcal{A}'$ accepts each path, $\mathcal{M}$ accepts the nested word; otherwise it searches for another path matching *reg*.

VPA of match $reg \overset{\forall\text{-}\textbf{child}}{\Longrightarrow} p$. $\mathcal{M}$ simulates p's VPA $\mathcal{M}_p$ on all the subtrees rooted at $\langle s$'s children. If $\mathcal{M}_p$ accepts each of $\langle s$'s child subtree, $\mathcal{M}$ accepts the nested word; otherwise it searches for another path matching *reg*.

VPA of match $reg \overset{\exists\text{-}\textbf{child}}{\Longrightarrow} p_1$ ***then*** $\ldots$ ***then*** p_k. Let any policy p_i's VPA be $\mathcal{M}_{p_i}$. $\mathcal{M}$ simulates $\mathcal{M}_{p_1}$ on $\langle s$'s first child subtree; if the subtree is not accepted by $\mathcal{M}_{p_1}$, this simulation is repeated on the next child subtree of $\langle s$. When $\mathcal{M}_{p_1}$ accepts such a subtree, $\mathcal{M}$ simulates $\mathcal{M}_{p_2}$ on the next child subtree, and so on. $\mathcal{M}$ accepts the word if after repeating these steps, $\mathcal{M}$ finds a subtree accepted by $\mathcal{M}_{p_k}$; otherwise $\mathcal{M}$ retries starting from the search for a path matching *reg*.

VPA of start S : *inner*. First, $\mathcal{M}$ simulates the DFA $\mathcal{A}$ (on the call symbols) that accepts paths from the root to some symbol in S that does not have any ancestor in S. Similar to the hierarchical policies, $\mathcal{M}$ then simulates *inner*'s VPA $\mathcal{M}_{in}$ on the subtree rooted at $\langle s$. $\mathcal{M}$ continues to apply retry steps (similar to the hierarchical policies) to the search for other paths matching $\mathcal{A}$ and checking if $\mathcal{M}_{in}$ accepts the relevant subtrees for each of those paths.

As expected, our compilation is sound with respect to SafeTree's nested word semantics.

Theorem 5.5 (Soundness). *Let p be a policy and $\mathcal{L}(VPA[\![p]\!])$ be the set of rooted well-matched nested words accepted by its visibly pushdown automaton $VPA[\![p]\!]$. Then,*

(1) $NW[\![p]\!] = \mathcal{L}(VPA[\![p]\!])$ *if p is a service tree policy,*
(2) $Tree[\![p]\!] = \mathcal{L}(VPA[\![p]\!])$ *if p is a hierarchical policy, and*
(3) $Sequence[\![p]\!] = \mathcal{L}(VPA[\![p]\!])$ *if p is a linear sequence policy.*

All three equivalences in the above theorem are proved by induction on the structure of the policies. The proof is given in the full paper.

Finally, we consider complexity. We can bound the size of the compiled VPA in terms of the size of the policy as follows:

Theorem 5.6. *Suppose the DFA of every regular expression in a policy p has at most R states; each sub-policy has at most k immediate sub-expressions, i.e., fan-out at most k; and (nesting) depth d. Then the VPA $\mathcal{M}(p)$ such that $NW[\![p]\!] = \mathcal{L}(\mathcal{M})$ has $O((k+1)^d R)$ states.*

The proof goes by structural induction on the policy p. The proof and definitions of depth and fan-out are given in the full paper. Regarding the quantity R, note that the worst case size complexity of a DFA is known to be exponential in the length of the regular expression [31], but in the common cases this quantity is often much smaller.

5.3 Runtime Monitor for a VPA

We will now formalize the monitor implementation for a VPA. This section first describes the centralized interpretation, which follows immediately from the standard VPA semantics, and then introduces a new distributed interpretation. All the compiled VPAs are deterministic and complete, so we treat the VPA transition relations as functions below.

Definition 5.7. A VPA $\mathcal{M} = (Q, q_{init}, F, \Sigma, \Gamma, \bot, \delta_c, \delta_r)$ can be interpreted as $[\![\mathcal{M}]\!]_{central} : (Q \times \mu) \to NestedWord(\Sigma) \to (Q \times \mu)$, which takes: (a) an initial VPA configuration (q, θ)—a pair of a state and some stack; (b) and a nested word $w \in NestedWord(\Sigma)$, and returns a final VPA configuration. We define $[\![\mathcal{M}]\!]_{central}$ inductively on the length of the nested word w:

(1) Case $w = (\epsilon, \phi)$:

$$[\![\mathcal{M}]\!]_{central}\ (q', \theta')\ w = (q', \theta')$$

(2) Case $w = (a_1 \dots a_k a_{k+1}, v)$, where $a_{k+1} = (e, i)$ for some $e \in \Sigma$:

$$[\![\mathcal{M}]\!]_{central}\ (q', \theta')\ w = (q, \theta), \text{ where } [\![\mathcal{M}]\!]_{central}\ (q', \theta')\ (a_1 \dots a_k, v[\iota(a_1), \iota(a_k)]) \xrightarrow{e} (q, \theta)$$

Intuitively, a distributed monitor is a set of call transition functions of the type call : $Q \to (Q \times \Gamma)$ and return transition functions of the type ret : $(Q \times \Gamma) \to Q$. Here, Q is a set of states and Γ is a set of stack elements. In the following definition of the distributed monitor, *Call* and *Ret* represent the set of all call and return functions.

Definition 5.8. A distributed monitor $\mathcal{D} : \Sigma \to (Call \times Ret)$ maps symbols in some alphabet Σ to a pair of call and return transition functions, where *Call* and *Ret* represent the set of all call and return functions of the type call : $Q \to (Q \times \Gamma)$ and ret : $(Q \times \Gamma) \to Q$. Here, Q is the set of states and Γ is a set of stack elements. We write *DistMon* for the set of distributed monitors.

Now, we present a translation function dist_monitor : $VPA \to DistMon$ to convert a VPA $\mathcal{M}$ to a distributed monitor $\mathcal{D} \in DistMon$.

Definition 5.9. The distributed monitor for some VPA $\mathcal{M} = (Q, q_{init}, F, \Sigma, \Gamma, \bot, \delta_c, \delta_r)$, where $\tilde{\Sigma}$ is the base alphabet that gets call-return augmented into Σ, as:

$$\text{dist\_monitor}(\mathcal{M}) \triangleq \left\{ (a, (\text{call}, \text{ret})) \;\middle|\; \begin{array}{l} a \in \tilde{\Sigma}, \\ \text{if } \delta_c(q, \langle a) = (q', s) \text{ then } \text{call}(q) = (q', s) \text{ and} \\ \text{if } \delta_r(q', s, a\rangle) = q \text{ then } \text{ret}(q', s) = q \end{array} \right\}$$

For any $a \in \tilde{\Sigma}$, the mappings in the call and return transition functions in $\mathcal{D}(a) = (\text{call}, \text{ret})$ can be viewed as the transition rules in δ_c, δ_r on the symbols $\langle a$ and $a\rangle$.

Operationally, a distributed monitor takes a pair of state and stack as an initial configuration, a nested word and returns a final configuration. Before defining the operational semantics for a distributed monitor, we introduce a single step transition operator $\xrightarrow{x}_{dist}$, where x is a symbol in some alphabet Σ.

Definition 5.10 (Single Step Transition). Let $\mathcal{D}$ be a distributed monitor defined over a set of base alphabet $\tilde{\Sigma}$ such that for any $e \in \tilde{\Sigma}$, the pair of call-return transition mapped to e is $\mathcal{D}(e) = (\text{call}, \text{ret})$. Let Σ be the call-return augmented alphabet of $\tilde{\Sigma}$ and x be some symbol in Σ. Let Q be the set of states, μ be the set of stacks, and the stack $\theta \in \mu$. The single step function $\xrightarrow{x}_{dist}$: $Q \times \mu \to Q \times \mu$ is defined as follows:

(1) if $x = \langle e$ for some $e \in \tilde{\Sigma}$ then $(q, \theta) \xrightarrow{x}_{dist} (q', \theta')$, where $\text{call}(q) = (q', s)$ and $\theta' = \theta s$,
(2) if $x = e\rangle$ for some $e \in \tilde{\Sigma}$ then $(q, \theta) \xrightarrow{x}_{dist} (q', \theta')$ where $\text{ret}(q, s) = q'$ and $\theta' s = \theta$.

When a distributed monitor $\mathcal{D}$ reads a symbol $a_i = (x, \iota(a_i))$ from a nested word $w = (a_1 \dots a_k, v)$, it selects a pair of call-return transition functions $\mathcal{D}(e)$ such that $x = \langle e$ or $x = e\rangle$. The pair $\mathcal{D}(e)$ is referred as a sub-monitor in the distributed monitor $\mathcal{D}$. Based on the tag of the symbol x, the call or return transition of $\mathcal{D}(e)$ is applied to the input configuration of $\mathcal{D}$. Formally:

Definition 5.11 (Operational Semantics). A VPA $\mathcal{M}$'s distributed monitor $\mathcal{D}$ = dist_monitor($\mathcal{M}$) can be interpreted as $[\![\mathcal{D}]\!]_{dist} : (Q \times \mu) \to NestedWord(\Sigma) \to (Q \times \mu)$, where $NestedWord(\Sigma)$ is the set of nested words on Σ. The run of a distributed monitor $\mathcal{D}$ on some nested word w starting at some initial state q and a distributed stack μ can be inductively defined on the length of the nested word w:

(1) Case $w = (\epsilon, \phi)$:

$$[\![\mathcal{D}]\!]_{dist}\ (q, \theta)\ w = (q, \theta)$$

(2) Case $w = (a_1 \dots a_k a_{k+1}, v)$, where $a_{k+1} = (e, i)$ for some $e \in \Sigma$:

$$[\![\mathcal{D}]\!]_{dist}\ (q, \theta)\ w = (q', \theta'), \text{ such that } [\![\mathcal{D}]\!]_{dist}(q, \theta)\ (a_1 \dots a_k, v[\iota(a_1), \iota(a_k)]) \xrightarrow{e}_{dist} (q', \theta')$$

Finally, we can show that running the distributed and the centralized variant of a VPA are equivalent, i.e., they accepted the same nested words:

Theorem 5.12. *Given a nested word w over some call-return augmented alphabet Σ and a VPA $\mathcal{M}$ whose run on w is $\rho(w) = (q_1, \theta_1) \dots (q_n, \theta_n)$ then:*

(1) *the run of $\mathcal{M}$'s centralized monitor is $[\![\mathcal{M}]\!]_{central}(q_1, \theta_1)\ w = (q_n, \theta_n)$, and*
(2) *the run of $\mathcal{M}$'s distributed monitor $[\![\mathcal{D}]\!]_{dist}(q_1, \theta_1)\ w = (q_n, \theta_n)$.*

The proof goes by induction on the length of the nested word w.

In section 7, we will see how to use these distributed monitors to enforce policies in an implementation. But first, we consider some example policies that can be expressed in our policy language.

6 Case Studies

This section motivates real-world service tree policies that are relevant to three teams: data-compliance or audit teams, security teams, and deployment teams. All case studies are presented in the context of the running example of the hospital management application from section 2.

Notation. We write Endpoint as shorthand for the set {Endpoint}, and $S = \{a_1, \dots, a_k\}$ as shorthand for the regular expression $a_1 + \dots + a_k$. We write Any = $\tilde{\Sigma}$ and !Endpoint for the set $\tilde{\Sigma}$ − {Endpoint}.

6.1 Deployment Team Policies

Thorough testing is a key step in the development process of a microservice application. Therefore, we describe case-studies about testing scenarios of varying complexities.

Case study 1: A/B testing. Say a deployment team wants to test the interaction of a small subset of beta testers, labeled as Beta, with a new version v2 of the Database service. For example, the label Beta might have been assigned to a random subset of users by the frontend, or it could be assigned to internal users which will test the service before it is released publicly. So the deployment team requires all traffic coming from Beta to be served by Database-v2 instead of Database-v1.

Let us model requests labeled as Beta to be requests from some endpoint labeled as Beta. The above A/B testing policy is specified as:

start Beta : **call-sequence** Beta(!Database-v1)*.

Since this policy specifies a constraint on subtrees starting at Beta, the policy is of the form **start** Beta : *inner*. The *inner* policy needs to be matched on a subtree starting at Beta. The policy *inner* has to match the sequence of API calls in the subtree rooted at Beta with a regular expression that prevents calls to v1 of Database.

Case study 2: Factorial testing. Deployment teams often want to test the interactions between all recently updated services. For instance, suppose a deployment team wants each request to either use the old v1 version of all services, or the latest v2 version of all services; this is a *factorial testing* scenario. More concretely, consider that the deployment team wants to prevent Test-v2 from invoking v1 versions of the De-identify and Lab services. This can be expressed as the policy:

$$\textbf{start}\ \text{Test-v2} : \textbf{call-sequence}\ (\text{Any} - \text{De-identify-v1} - \text{Lab-v1})^*.$$

Here, (Any − De-identify-v1 − Lab-v1) is the set of all endpoints besides v1 of De-identify and Lab.

Case study 3: Regional access control. Sometimes certain services need to have restricted regional access. For instance, suppose the hospital application wants to prevent EU users from accessing the main Database service to avoid inadvertent violation of GDPR guidelines. Supposing EU users are labeled at the frontend as Frontend-EU, we can express this policy as:

$$\textbf{start}\ \text{Frontend-EU} : \textbf{call-sequence}\ \text{Frontend-EU}(!\text{Database})^*.$$

Case study 4: External requirement. A deployment team might want to specify some business logic involving services with global effects, like a write service that updates the database or an account creation service that creates a new user. For example, new appointments should be saved in the appointment database by invoking some Database service. This policy can be expressed as:

$$\textbf{start}\ \text{Appointment} : \textbf{call-sequence}\ (\text{Appointment}\ \_\ \text{Database}\_).$$

The pattern _ matches any sequence of API calls. The Database service can be replaced by, say, a Log service to log admin access to some resource.

6.2 Security Team Policies

Case study 5: Payment logging. Suppose a security team wants all Payment requests to call payment Database at least once, and Database to send requests only to EventLog. The team can state this requirement as "Payment invokes at least one Database and this Database invokes EventLog as all its children," and specify this policy as:

$$\textbf{start}\ \text{Payment} : \textbf{match}\ (\text{Payment Database}) \overset{\forall\textbf{-path}}{\Longrightarrow} (\text{EventLog})\_.$$

The **match** (Payment Database) matches a subtree rooted at Payment that invokes Database as its child. To specify that all children of this Database are EventLog, the right side sub-expression of $\overset{\forall\textbf{-path}}{\Longrightarrow}$ matches all outgoing paths from Database with the regular expression (EventLog) _, where the pattern _ matches any sequence of API calls.

Case study 6: Data Vault—no outgoing calls, a constraint on the leaves. Consider that the hospital application has a data Vault service to store patient records, and a security team wants to restrict Vault from invoking any endpoints to prevent it from sharing confidential data with any third-party services. This property can be expressed by requiring the Vault service to have no children in the service tree:

$$\textbf{start}\ \text{Any} :\ \textbf{match}\ \text{Any} \overset{\forall\textbf{-path}}{\Longrightarrow} (!\text{Vault})^*(\text{Vault} + \epsilon).$$

This can also be expressed as a singleton sequence policy: **start** Vault : **call-sequence** Vault.

Case study 7: Resource pricing. Suppose that patients can invoke Test service to request a batch of medical tests. For each medical test, the Test service sends a child request. To ensure correct billing, a security team might require every child call of Test to directly or indirectly invoke Payment. This is policy can be specified as:

$$\textbf{start}\ \text{Test} : \textbf{match}\ \text{Test} \overset{\forall\text{-}\textbf{child}}{\Longrightarrow} (\textbf{match}\ \_\ \text{Payment} \overset{\forall\text{-}\textbf{path}}{\Longrightarrow} \_)\,.$$

6.3 Compliance Team Policies

Case study 8: Data compliance. Data protection laws, like HIPAA and GDPR, mandate compliance teams to responsibly handle customer data. For instance, personal health information, like a patient's name, should be de-identified for privacy. Since compliance teams do not have direct insight into the implementation of the application, they would like to check compliance by monitoring inter-service communications. To encode this kind of requirement as a service tree policy, they can require that a service sending a request to external parties should have previously run the de-identification service. This policy can be specified as:

$$\textbf{start}\ \text{Test} :\ \textbf{match}\ \text{Test} \overset{\exists\text{-}\textbf{child}}{\Longrightarrow} p_1\ \text{then}\ p_2\,,$$

where $p_1 = \textbf{match}\ \text{De-identify} \overset{\forall\text{-}\textbf{path}}{\Longrightarrow} \epsilon$ and $p_2 = \textbf{match}\ \text{Lab} \overset{\forall\text{-}\textbf{path}}{\Longrightarrow} \epsilon$.

The right sub-expression of $\overset{\exists\text{-}\textbf{child}}{\Longrightarrow}$ needs to specify the existence of two subtrees satisfying sub-policies p_1 and p_2. Here, p_1, p_2 will specify the existence of a call to De-identify and Lab respectively. The **match** sub-expression in p_1 and p_2 are "**match** De-identify" and "**match** Lab" respectively because they need to match a subtree rooted at De-identify and Lab. In the above policy, the right side expression in p_1 and p_2 is an empty string because the team wants De-identify and Lab to not invoke any APIs. This regular expression prevents the subtrees rooted at De-identify and Lab from having any calls to children. Note the regular expression on the right of policy p_1 and p_2 could be replaced by _ if the compliance team wanted to allow De-identify and Lab to invoke other services.

Add-on policy. Suppose the compliance team wants each Test request to send one Lab request. Additionally, it wants Test to de-identify the patient records before invoking Lab. This property specifies constraints on the sequence of API calls in subtrees starting at Test, so the policy is:

$$\textbf{start}\ \text{Test} :\ \textbf{call-sequence}\ (!\text{Lab})^*\text{De-identify}(!\text{Lab})^*\text{Lab}(!\text{Lab})^*\,.$$

The above regular expression specifies that De-identify is called before Lab.

Case study 9: Data Proxy or Middleware. Sometimes the access to a service needs to be managed by a proxy, such as a firewall to secure a data source, a load balancer, or an authenticator. Consider the compliance team wants Test to invoke Lab via an authentication service Auth to avoid creating unauthorized lab requests. More formally, we need to express that the subtree rooted at Test should have an Auth descendant, and Lab should be directly or indirectly invoked by Auth, meaning Lab is Auth's descendant. This requirement can be specified as the following policy:

$$\textbf{start}\ \text{Test} :\ \textbf{match}\ \text{Test} \underbrace{\overset{\exists\text{-}\textbf{child}}{\Longrightarrow}}_{(1)} (\textbf{match}\ (!\text{Lab})^*\text{Auth} \underbrace{\overset{\exists\text{-}\textbf{child}}{\Longrightarrow}}_{(2)} (\textbf{match}\ \_\text{Lab} \underbrace{\overset{\forall\text{-}\textbf{path}}{\Longrightarrow}}_{(3)} \_))\,.$$

First, we specify Test as the start endpoint. Subtrees starting with Test certainly need the root to be Test. This is described in the regular expression associated with **match** condition of the policy labeled with (1). The inner policy to the right of (1) checks for the existence of a subtree that has

a path to Auth. The regular expression (!Lab)*Auth on the left of (2) permits Auth, but no Lab in the path. After matching this path, the inner policy of (2) checks for the existence of Lab as a descendant, which is similarly expressed as the expression **match** _Lab on the left of (3). Since Lab can be followed by any endpoint, the regular expression on the right of the label (3) is _.

This policy can be strengthened to require that Auth is called before the only call to Lab, as follows:

start Test : **call-sequence** (!Lab)*Auth(!Lab)*Lab(!Lab)*.

7 Implementation

To demonstrate our design, we developed a prototype implementation in $\sim 2k$ lines of Java. Our tool compiles a policy into a VPA, and then extracts a runtime monitor that runs on top of the Istio service mesh. At its core, the runtime monitor enforces the policy by simulating a VPA in a distributed manner, using the distributed monitors in definition 5.9. Here, we discuss aspects of our monitor's design that are particular to the Istio [19] service mesh framework.

Istio-based implementation. In the service mesh framework, each microservice container instance is paired with a *sidecar* container implemented as an Envoy proxy [13]. The sidecar can intercept all incoming and outgoing HTTP traffic for its corresponding service container, and the Envoy proxy running at the sidecar can perform a variety of useful functions orthogonal to our work, such as load balancing, service discovery, failover, etc. In addition, Envoy can be programmed with custom logic to inspect HTTP headers and perform actions, like adding/removing/updating the headers, or allowing/denying HTTP traffic, etc. Envoy proxies maintain an in-memory state for each request/response pair for the duration of the request's lifetime. Therefore, any metadata saved in the in-memory state during the request processing can be retrieved during its response's processing. While not a core feature of the framework, this capability turns out to enable some important optimizations for our enforcement method, which we will discuss below.

Local monitors as Envoy filters. We implement our local monitors as Envoy filters, which are custom traffic filtering Lua scripts that simulate VPA transitions on the service (symbol) from the current VPA configuration carried in the HTTP header. Since in our setting, the service trees are rooted well-matched nested word, the top of the stack symbol read by a response message is the same as the value pushed by its matched request. Therefore, instead of carrying the stack in the HTTP header, we save the stack symbol locally in the proxy's in-memory state. This reduces the memory overhead of propagating stack information along with requests, which can be significant. With our design, only the VPA state is carried in the HTTP header.

Extracting local monitors. Given a VPA $\mathcal{M}$ and its distributed monitor $\mathcal{D} = \text{dist\_monitor}(\mathcal{M})$ as defined in definition 5.9, our compiler extracts the filter for a service $s \in \tilde{\Sigma}$ from the call and ret function mapped to s, *i.e.,* $(\text{call}, \text{ret}) = \mathcal{D}(s)$. The call and ret functions are essentially the VPA's call and return transitions on the service s. The filter comprises two callback functions: OnRequest implements call, while OnReponse implements ret.

For example, the two callbacks for Payment's (P) local monitor for the VPA in fig. 2 are given in fig. 8. The OnRequest function implements the call transition on P as a conditional block that updates the state header and the custom in-memory metadata, local_stack to q_P if the current state is $start$. Similarly, OnResponse implements the return transition on P that updates state to $start$ if the current state header and local_stack are q_P.

Local monitor execution. When a request arrives at a service, the service's co-located proxy executes the filter's OnRequest callback to run a call transition. For instance, if the state header

Payment

```
callback OnRequest(){
  if (state == start) then
      state = q_P; local_stack = q_P
}
```

```
callback OnResponse(){
  if (state == q_P && local_stack = q_P) then
      state = start
}
```

Fig. 8. Payment's (P) local monitor for the VPA in fig. 2: OnRequest and OnResponse simulate VPA transition on requests and responses respectively. The top of the stack symbol is locally saved as local_stack metadata in the in-memory state of P's proxy and state is carried in the HTTP header. Here, qp, $start$ are VPA states.

of an incoming request at P is set to $start$, OnRequest updates state to q_P and writes q_P to the (custom) local_stack metadata in P's proxy's in-memory state (corresponding to this request's session).

Likewise, the OnResponse callback is executed on intercepting a response. For instance, if the current state header of an outgoing response from P is q_P and the in-memory local_stack saved during its corresponding request's processing was q_P, OnResponse updates the state header to $start$. Thus, the callback implements our distributed monitor's single-step transition (as defined in definition 5.10).

Context propagation. We require that the SafeTree header carrying the VPA state is propagated from an *incoming* request to any resulting *outgoing* request. Although Envoy can see requests entering and leaving the service, the service itself is a black box from its perspective, so Envoy does not directly know which incoming (parent) requests produced which outgoing (child) requests. We therefore require that the application copy the header from incoming to outgoing requests. This functionality is known in the microservice community as *context propagation*; it is anyway required for other purposes—in particular, distributed tracing to monitor application behavior and track the cause of request failures and performance issues—and libraries exist to help implement it [20, 38]. SafeTree does not assume anything more about microservice applications beyond this standard requirement.

Rejecting invalid service trees. For simplicity, our implementation logs any policy violation after the request's entire service tree has been processed, rather than actively blocking requests. It should be possible to extend the prototype to block a request as soon as we know the policy must be violated—depending on the policy this can happen early or later in the service tree. For example, for policies of the form **start** S : $inner$, which require subtrees starting at the symbols in S to satisfy $inner$, the response from the root of such subtrees can be early blocked if the $inner$ policy is violated on the subtree. For certain policies, it is possible to block a request if transitioning on it will send the VPA into a state that's sufficient for it to never accept the service tree. For instance, for a policy of the form **start** S : **call-sequence** reg, a request can be blocked if transitioning on it will violate reg. Similarly, for policies of the form **start** $S : reg \overset{\forall\text{-path}}{\Longrightarrow} reg$ and **start** $S : reg \overset{\forall\text{-child}}{\Longrightarrow} p$, a request can be blocked if the start symbol in the start set S is not the same as the first symbol of all the words in the language of reg. For instance, **start** A : B $\overset{\forall\text{-path}}{\Longrightarrow}$ B.

8 Evaluation

We evaluate two aspects of the SafeTree monitor: its performance overhead and its memory footprint by considering the following research questions:

- *RQ1: How much header space is required for the context headers?*
- *RQ2: How much latency overhead does the monitor add?*

Setup. Our experimental setup consists of two microservice applications written in Go: a hotel reservation application from the DeathStarBench [14] and a simple hospital application that we wrote to exhibit the call structure of the application described in section 2. The average number of nodes in the service trees of both the applications are 4.5 and 6 respectively.

While the service implementations here—especially in the hospital application—are rather simple, the specific logic inside the application does not affect the overhead of the SafeTree monitor since SafeTree runs in the service mesh outside the application, so its performance is not affected by application logic. SafeTree's overhead does, however, depend on the topology of API calls and the policies being checked, which we will study in the evaluation.

The microservice applications are deployed on a minikube cluster enabled with Istio sidecar injection. The cluster runs locally on a machine with 16 GB of RAM, an i7 processor, and Ubuntu 22.04 operating system. The application's inter-service communication is managed by the Envoy proxy running in the Istio version 1.23.2. The case study policies used in our experiments are listed in the full paper.

8.1 RQ1: Header Space Overhead

Maintaining the current state of the VPA-based runtime monitor and its stack configuration is central to our enforcement mechanism. As described in section 7, the stack configuration is saved locally at sidecar proxies, but the current state of the VPA is propagated alongside requests in a custom HTTP header. Table 1 presents the total number of call and return transitions for each policy's VPA (in columns #call and #return); total number of VPA states (in #state column); and the number of bits needed to encode the maximum number of VPA states (in #bits column). The context headers for all our policies are at most six bits long, which is minimal compared to the available space in HTTP headers (on the order of kilobytes).

Table 1. Policies prefixed with "Hotel" are evaluated on the hotel application, and the remainder are evaluated on the hospital application. Main findings: (1) context headers can encode the VPA state in a small number of bits, and (2) policy checking adds minimal latency, on the order of a millisecond to the application.

	VPA transitions		VPA states		Latency	Policy	Nesting
Scenario	#call	#return	#states	#bits	overhead (ms)	class	#levels
A/B Testing	6	30	6	3	0.700	Linear	NA
Factorial Testing	11	80	11	4	0.420	Linear	NA
Access Control	12	100	12	4	0.448	Linear	NA
Update	25	544	25	5	0.433	Hierarch.	2
Data-compliance	38	1326	38	6	0.958	Hierarch.	2
Data Proxy	36	1184	36	6	1.117	Hierarch.	3
Encryption	23	454	23	5	0.460	Hierarch.	1
Data Vault	20	346	20	5	0.370	Hierarch.	1
Resource pricing	25	562	25	5	0.457	Hierarch.	2
Hotel Encryption	23	454	23	5	0.216	Hierarch.	1
Hotel Data Proxy	36	1184	36	6	0.443	Hierarch.	3
Hotel Compliance	38	1326	38	6	0.278	Hierarch.	2

We can understand how the size of VPAs vary across different classes of policies if we look at the #states column and the "Policy class" column, which describes if a policy is linear or hierarchical. We observe that the linear policies compile to a VPA with fewer states than the hierarchical policies. If we look at the "Nesting #levels" column, we can further observe that among the hierarchical

policies, more deeply nested policies tend to have more VPA states. The data-compliance policy appears to be an outlier as it has the greatest number of states even though other policies have deeper nesting, but this is because the policy specifies multiple existential condition on subtrees. ***To summarize, the SafeTree monitor requires only a few bits of extra HTTP header space to compactly encode contextual information about the service tree structure.***

8.2 RQ2: Latency Overhead

Another aspect of the SafeTree monitor's evaluation is to understand its effect on the application's performance. Accordingly, we compare the latency of requests when the application is being monitored versus when it is not, on a workload of 200 requests for all the user-facing endpoints of the application. Latency benchmarking in a microservice application is prone to variance due to several network factors, like congestion, application's concurrency, elastic scaling, etc. We ensure that the application is not overloaded by sending the requests at a sufficiently low rate so that we are measuring per-request processing latency, rather than queuing effects. To minimize the variation in our results, we average the latency over five such workloads.

Overhead versus Policy. Our experiment involves extracting Envoy filters from the VPAs compiled in the previous experiment. Then for each policy, latency overhead is measured as the difference between the average request latency in the above applications when the policy checking enabled and when it is disabled. The policies in table 1 that are prefixed with "Hotel" were evaluated on the hotel reservation application, and the remainder were evaluated on the hospital application. The "Latency overhead" column in table 1 reports the average latency overhead in milliseconds.

Observe that all values are at most 2ms, which implies low latency overhead of running the monitor. We found these results to be stable across both of our benchmark applications, which is expected since the internal details of the services should not affect the latency overhead. ***To summarize, SafeTree monitor adds minimal latency overhead—on the order of a millisecond.***

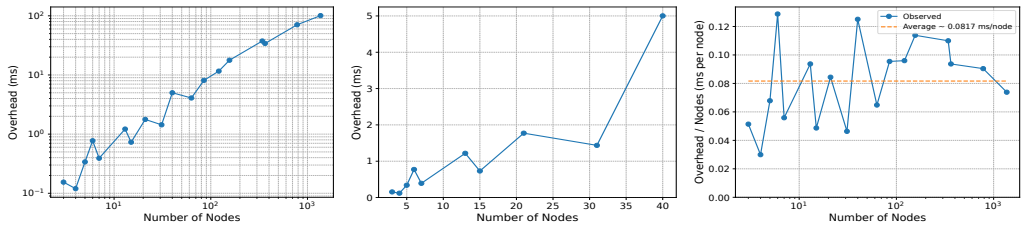
Checking multiple policies. To check multiple policies simultaneously, instead of unioning the VPAs of individual policies, we run each VPA independently. This prevents blowing up the VPA size, which is essential for maintaining low memory overheads. To understand the latency overhead of running multiple policies, we run multiple copies of the "Hotel Compliance" monitor from table 1. The column "# Policies" in table 2 describes the number of simultaneous policies being checked. We can conclude from the "Latency overhead" column that the latency overhead increases with the increase in the number of policies. **The results suggest we could feasibly monitor multiple policies with a reasonable amount of overhead.**

Table 2. Overhead for multiple policies.

# Policies	Latency overhead (ms)
1	0.278
2	0.510
3	0.610
4	0.676

Overhead versus topology scale. To evaluate SafeTree monitor's scalability, we measure the increase in latency overhead with the scale of the application's topology, which we measure as the number of nodes in the application's service tree. For this experiment, we synthetically generate applications with different topology shapes—all combinations of depths from 2 to 5 and fan-out from 1 to 4, resulting in the total number of nodes in the service trees ranging from 3 to 1365. We measure the latency overhead for the same ("Hotel Compliance") policy across different applications.

Fig. 9a shows that the latency overhead in milliseconds (on the y-axis) linearly increases with the increase in the number of nodes in the service tree (on the x-axis). Fig. 9a uses log scale on both the x-axis and y-axis. As a reference point, note that the data collected by Alibaba [25] showed that in their microservice deployment, the common case depth and fan-out per service is lower



(a) Overhead vs Nodes (b) Overhead vs Nodes (≤40) (c) Overhead to Nodes Ratio

Fig. 9. Latency overhead vs topology scale, measured as number of nodes in the application's service tree.

than the median depth of ~ 4 and the median fan-out of ~ 2. A binary tree with depth 4 will have maximum 31 nodes; the common case trees will have fewer nodes.

We zoom into the latency overhead for trees with nodes between 0–40 in fig. 9b, where the latency overhead in milliseconds is plotted on the y-axis, and the x-axis reports the number of nodes on a linear scale. Notice that the common case overhead is at most 5 ms and the overhead for a typical tree with ~ 31 nodes is at most 2 ms.

We plot the per-hop overhead (latency overhead divided by number of nodes) in milliseconds on fig. 9c's y-axis for trees with a range of nodes (plotted on the log scale on the x-axis). We can see that on an average 0.082 ms of latency overhead is incurred per hop in the service tree. ***In summary, SafeTree monitor's latency overhead is under*** ~ 5***ms for a typical topology. The overhead linearly increases with the tree's size, adding*** 0.082***ms of average overhead per hop.*** We expect that a production implementation of SafeTree could achieve significantly better performance, e.g., by writing filters using C++/WASM instead of Lua.

9 Related Work

Safety in microservice and cloud applications. Today's microservice systems support policies that control communication between pairs of microservices. Recent works have explored more general policies. For instance, Trapeze [2] is a system for dynamic information flow control [34] in serverless computing. Trapeze can precisely specify how data at different security levels flow around the application. In contrast, our system can specify properties about the structure of the API call tree.

Another interesting work in this area is Whip [36], a higher-order contract system for describing service-level specifications as contracts on blackbox services. Whip policies can describe the arguments and return values of microservices; however, these policies are focused on individual API calls, and require a custom network adapter for monitoring. In contrast, our approach can express policies about trees of API calls using a lightweight monitoring approach that can be implemented in existing service mesh frameworks.

Our prior work [17] proposed policies for microservices based on a linearization of the service calls; SafeTree is more general in supporting policies that describe the tree structure of the calls, which requires a richer automaton (VPA) for monitoring. The Copper [35] system also uses this idea of linearization to combine several single hop policies. Unlike SafeTree, Copper does not support tree policies, nor policies over sequences of API calls.

Execution correctness in serverless runtime. A recent line of work aims to make it easier to correctly execute microservice applications on serverless platforms. For example, serverless operational semantics were formalized in the λ_λ calculus [21]; language primitives were introduced in *μ2sls*

[22] to write microservice code with transaction and asynchrony abstractions without manually handling failures and execution nuances; durable functions [8] introduced a programming model for ensuring orchestration correctness of stateful workflows under retries and failures in otherwise stateless serverless applications. SafeTree assumes that microservices are executed correctly, but instead checks that the runtime behavior conforms to some desired specification.

Programmable runtime verification in networks. Our work can be seen as a programmable system for runtime verification in the service mesh layer. Runtime verification has been used at other layers of the network stack. For instance, DBVal [24] checks for packet forwarding correctness. Hydra [32] allows the network operator to enforce their desired policy and specify custom telemetry to attach to packets in a custom high-level language, and synthesizes monitors to enforce polices in the dataplane. For security policies, Poise [23] converts high-level security and access control policies into P4 code for enforcement. At a higher level, Aragog [37] is a scalable system for specifying and enforcing policies about high-level events in networked systems.

Distributed and decentralized runtime verification. Distributed and decentralized monitoring has been well-studied in the runtime verification community, but most works target distributed systems and are designed to contend with monitors that may see events out of order [6] or monitors that may see different views of the global system [27]. We do not face such difficulties in our setting: the state of our distributed monitors is carried with the request, and so each monitor has the full information required to enforce the policy. Distributed tracing frameworks [20, 30, 38] used for observability rely on a centralized collector for recording execution traces, which can then be analyzed later. In contrast, a key goal of our work is to check policies and detect violations efficiently at runtime, rather than after the fact.

Automata-theoretic models for hierarchical structure. Our policy specification and enforcement is based on nested word languages and visibly-pushdown automata [5]. The literature on nested words and VPAs is too large to survey here; the interested reader should consult Alur and Madhusudan [5]. While tree automata [11, 12] also work on hierarchical input, their inputs must be structured trees. SafeTree uses VPA because its input is a serialized tree of nodes being incrementally processed, not the entire service tree structure.

Monitoring context-free properties. SafeTree policies resemble to context-free properties, which can be captured in extensions of temporal logic. For instance, VLTL [7] supports a more complex policy language, although monitoring such policies requires Büchi and parity automata, which seem difficult to realize in a microservice setting. Other examples include CaRet [3] and PTCaRet [33], temporal logic extensions with call-return matching that can be interpreted using recursive state machines. Runtime monitoring algorithms that have been considered for context-free properties specified in temporal logic are include formula rewriting-based pushdown-automata for PtCaRet [33]; pushdown Mealy machine for CaReT with future fragment [10]; and an LR(1) parsing based algorithm for parametric properties [26].

In the realm of software verification, PAL [9] is a DSL for writing context-sensitive monitors for C programs, where the user directly encodes the low-level state transitions of the monitor, and the framework automatically instruments an existing C program. In contrast, the SafeTree policy language is high-level, and the monitoring automaton is generated automatically.

10 Conclusion

We have presented SafeTree, a policy language for specifying rich, tree-based safety properties for microservices. By compiling policies to VPA, we derive an efficient and performant distributed runtime monitor for enforcing our policies without invasive code changes to microservices.

We see several possibilities for future work. First, extending our work to the asynchronous setting, where API calls are processed concurrently, will expand the applications of our work. However, it is unclear how to specify safety policies where part of the service tree may not be completed yet, and asynchronous calls may return in a different order than they were originally issued, leading to service trees that may not be well-matched. Second, it can be useful to support policies that reason about arguments of API calls. It could also be interesting to support richer nested word languages—our design uses just call and return symbols, but internal symbols might be useful for modeling other aspects of microservice behavior. Finally, our monitoring strategy could be useful beyond microservices; for instance, for network control planes.

Acknowledgements

We thank the reviewers for their constructive feedback. This work is supported by NSF grants #2152831 and #2312714. This work benefited from discussions with Loris D'Antoni and the HTTP benchmark tool by Talha Waheed.

Data-Availability Statement

Our artifact [16] consists of the SafeTree policy compiler [15], which takes in a policy, and generates a VPA followed by emitting a distributed monitor. It also contains the source code of the policy compiler to generate a monitor; source for the benchmark applications; a list of test policies; and workload data to reproduce the experiments.

References

[1] Proton AG. 2024. Complete guide to GDPR compliance. https://gdpr.eu/. Accessed: 2024-11-04.

[2] 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 (2018), 118:1–118:26. doi:10.1145/3276488

[3] Rajeev Alur, Kousha Etessami, and P. Madhusudan. 2004. A Temporal Logic of Nested Calls and Returns. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Barcelona, Spain (Lecture Notes in Computer Science, Vol. 2988)*. Springer-Verlag, 467–481. doi:10.1007/978-3-540-24730-2_35

[4] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *ACM SIGACT Symposium on Theory of Computing (STOC), Chicago, Illinois*. ACM, 202–211. doi:10.1145/1007352.1007390

[5] Rajeev Alur and P. Madhusudan. 2006. Adding Nesting Structure to Words. In *International Conference on Developments in Language Theory (DLT), Santa Barbara, California (Lecture Notes in Computer Science, Vol. 4036)*. Springer-Verlag, 1–13. doi:10.1007/11779148_1

[6] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2020. Runtime Verification over Out-of-order Streams. *ACM Trans. Comput. Log.* 21, 1 (2020), 5:1–5:43. doi:10.1145/3355609

[7] Laura Bozzelli and César Sánchez. 2018. Visibly Linear Temporal Logic. *J. Autom. Reason.* 60, 2 (Feb. 2018), 177–220. doi:10.1007/s10817-017-9410-z

[8] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 133 (Oct. 2021). doi:10.1145/3485510

[9] Swarat Chaudhuri and Rajeev Alur. 2007. Instrumenting C programs with nested word monitors. In *International SPIN Conference on Model Checking Software, Berlin, Germany*. Springer-Verlag, 279–283.

[10] Normann Decker, Martin Leucker, and Daniel Thoma. 2013. Impartiality and Anticipation for Monitoring of Visibly Context-Free Properties. In *Runtime Verification (RV), Rennes, France*. Springer-Verlag, 183–200.

[11] John Doner. 1970. Tree acceptors and some of their applications. *J. Comput. Syst. Sci.* 4, 5 (Oct. 1970), 406–451. doi:10.1016/S0022-0000(70)80041-1

[12] Joost Engelfriet. 2015. Tree Automata and Tree Grammars. *CoRR* abs/1510.02036 (2015). arXiv:1510.02036 http://arxiv.org/abs/1510.02036

[13] Envoy. 2024. Envoy Proxy. https://www.envoyproxy.io/. Accessed: 2024-11-04.

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou.

2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS), Providence, Rhode Island*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3297858.3304013

[15] Karuna Grewal. 2025. SafeTree Compiler. https://github.com/aakp10/SafeTree-Compiler.

[16] Karuna Grewal. 2025. *SafeTree: Expressive Tree Policies for Microservcies*. doi:10.5281/zenodo.15751182

[17] Karuna Grewal, Philip Brighten Godfrey, and Justin Hsu. 2023. Expressive Policies For Microservice Networks. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Cambridge, Massachusetts*. ACM, 280–286. doi:10.1145/3626111.3628181

[18] Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. 2025. SafeTree: Expressive Tree Policies for Microservices. arXiv:2508.16746 [cs.PL] https://arxiv.org/abs/2508.16746

[19] Istio. 2024. Istio. https://istio.io/. Accessed: 2024-11-04.

[20] Jaeger. 2024. Jaeger. https://www.jaegertracing.io/. Accessed: 2024-11-04.

[21] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 149 (Oct. 2019). doi:10.1145/3360575

[22] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing Microservice Applications on Serverless, Correctly. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 13 (Jan. 2023). doi:10.1145/3571206

[23] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *USENIX Security Smposium (USENIX)*. USENIX Association, 595–612. https://www.usenix.org/conference/usenixsecurity20/presentation/kang

[24] K. Shiv Kumar, Ranjitha K., P. S. Prashanth, Mina Tahmasbi Arashloo, Venkanna U., and Praveen Tammana. 2021. DBVal: Validating P4 Data Plane Runtime Behavior. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. ACM, 122–134. doi:10.1145/3482898.3483352

[25] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC), Seattle, Washington*. Association for Computing Machinery, New York, NY, USA, 412–426. doi:10.1145/3472883.3487003

[26] P. O. Meredith, D. Jin, F. Chen, and G. Rosu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), L'Aquila, Italy*. IEEE Computer Society, USA, 148–157. doi:10.1109/ASE.2008.25

[27] Menna Mostafa and Borzoo Bonakdarpour. 2015. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *IEEE International Parallel and Distributed Processing Symposium, (IPDPS), Hyderabad, India*. IEEE Computer Society, 494–503. doi:10.1109/IPDPS.2015.95

[28] US Department of Health and Human Services. 2024. Summary of the HIPAA Privacy Rule. https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html. Accessed: 2024-11-04.

[29] US Department of Health and Human Services. 2024. The De-identification Standard. https://www.hhs.gov/hipaa/for-professionals/special-topics/de-identification/index.html.

[30] OpenTelemetry. 2024. OpenTelemetry. https://opentelemetry.io/. Accessed: 2024-11-04.

[31] M. O. Rabin and D. Scott. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.* 3, 2 (April 1959), 114–125. doi:10.1147/rd.32.0114

[32] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. 2023. Hydra: Effective Runtime Network Verification. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), New York, New York*. ACM, 182–194. doi:10.1145/3603269.3604856

[33] Grigore Rosu, Feng Chen, and Thomas Ball. 2008. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In *International Workshop on Runtime Verification (RV), Budapest, Hungary (Lecture Notes in Computer Science, Vol. 5289)*. Springer-Verlag, 51–68. doi:10.1007/978-3-540-89247-2_4

[34] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[35] Divyanshu Saxena, William Zhang, Shankara Pailoor, Isil Dillig, and Aditya Akella. 2025. Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies. In *International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS), Rotterdam, The Netherlands*. Association for Computing Machinery, New York, NY, USA, 233–248. doi:10.1145/3669940.3707257

[36] Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: higher-order contracts for modern services. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 36 (Aug. 2017). doi:10.1145/3110280

[37] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. 2020. Aragog: Scalable Runtime Verification of Shardable Networked Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual Event*. USENIX Association, 701–718. https://www.usenix.org/conference/osdi20/presentation/yaseen
[38] Zipkin. 2024. Zipkin. https://zipkin.io/. Accessed: 2024-11-04.

Received 2025-03-26; accepted 2025-08-12