Montag: Cloud Native Data Tainting and Policy Enforcement

Hantao Wang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-105
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-105.html

May 29, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank Michael Chang and Scott Shenker for their help and advice through this project and my time at Netsys.

Montag: Cloud Native Data Tainting and Policy Enforcement

by Hantao Wang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Scott Shenker

Professor Scott Shenker
Research Advisor

05 / 27 / 2020

(Date)

* * * * * * *

Professor John Kubiatowicz
Second Reader

05 / 29 / 2020

(Date)

Doc ID: 682ba1ba81debcd6570f346140f6b8a7cf66200d

Abstract

In the wake of the newly adopted privacy regulations (GDPR) and recent string of user data compromises (Equifax, etc), there is an urgent need for operators and regulatory experts to be able to deploy data related policies within their network and control the flow of sensitive data in a way beyond what can be afforded through database ACLs. This new system must be scalable, easily to thousands of services, fit into the current architecture and industry standards, and retain the same easy of development and deployment that comes with microservices.

Montag is a system built on top of the existing popular orchestrator and service mesh infrastructure, Kubernetes and Istio, that allows data taints to be forwarded along with the data itself from service to service. This should require minimal changes to the application logic of the services and no changes to the infrastructure code. Privacy experts and cluster operators should be able to declaratively define global and domain specific privacy policies that are automatically enforced at the data plane based on the tuple's taints and the source / destination service. In our experimentation, we find that Montag operates with just a 0.4% increase in end to end latency for our example microservice application when using Kubernetes and Istio, and a 2.4% increase compared to applications just using Kubernetes.

Contents

Contents **i**

List of Figures **ii**

1 Introduction **1**
1.1 Vocabulary 1
1.2 Motivation 1
1.3 Design Criteria 3

2 Design **7**
2.1 Tainting 7
2.2 Configuration 11
2.3 Application Edge Cases 12

3 Implementation **15**
3.1 Kubernetes and Istio Background 15
3.2 Montag Reverse Proxy 18
3.3 Montag Database Proxy 20
3.4 Deployment 21

4 Evaluation **23**
4.1 Test Application 23
4.2 Experimentation 25
4.3 Code Efficiency Comparison 27

5 Discussion **29**
5.1 Future Work 29
5.2 Related Work 31
5.3 Conclusion 33

Bibliography **35**

List of Figures

2.1 Tainting of parent and child requests 8
2.2 Demonstration of request tainting in the application 9

3.1 Architecture of the Envoy proxy 17
3.2 Structure of the DB Proxy sidecar 20

4.1 Architecture of the Bookinfo application 24
4.2 99th Percentile End-to-End Latency, in Seconds 26
4.3 99th Percentile through a Single Proxy, in Seconds 27

Chapter 1

Introduction

1.1 Vocabulary

We define a *data tuple* to be a single record or unit of data. In many cases, it will be something like a single row from a table or a single key-value pair. Tuples are able to move from *service* to service via HTTP, where each service is implemented as a set of containers. We define data flowing *downstream* to be data stored in the request and flowing from a client to a server or a caller or a callee. Data that flows *upstream* is stored in the response and goes in the opposite direction. This is an important note as the terminology is sometimes reversed in other works.

1.2 Motivation

The modular design of microservice applicants works by breaking a larger application into its individual functions and then packaging each function into separate containers [7] as independent services. Entire teams or even companies can be dedicated to building and maintaining a singular microservice, such as Redis, Elasticsearch, or Apache Spark [10]. These services, despite existing as independently managed code, are run together on the same

cluster by packaging them into containers and deploying them with container orchestrators such as Kubernetes [15]. Container orchestrators provide a rich set of abstractions for cluster state management and features such as service discovery, load balancing, and networking. The Kubernetes network can be even be transformed into a powerful ”service mesh,” which not only connects service to service but also adds the additional functionalities of traffic shaping, fine-grained load balancing, traffic monitoring, and security via traffic inspection.

Data in a distributed system, due to its widely complex and different nature, must then be treated differently from data in a monolithic system. It is crucial to manage data well, especially in the age in increasingly strict privacy and security regulations. However, managing data is not at all like managing code or infrastructure, and we lack the appropriate tools and abstractions to efficiently manage data in a way that follows current and future regulations [5, 16, 20]. The data, over its lifetime, is often accessed by the hundreds or thousands of services spread cross tens of thousands of servers and will be shifted from service to service, undergoing countless transformations. Existing role based access control (RBAC) policies solely at the database layer are not well-suited for a distributed application.

Consider the case where one privileged service with the correct permissions accesses a piece of sensitive data. It then can pass that data down to another service it trusts. We consider the passing of data from one service to another the flow of data through a distributed microservice application. As data moves around, it becomes increasingly more difficult to enforce permissions around accessing that data, as role-based access control only happens at the database. For a complex system with hundreds of complex independently-owned microservices, one cannot expect the service developer to know for every piece of data, what are the services that have the correct permissions to receive this data.

Another, perhaps more common, use case is when there is so much data stored in so many different databases in a distributed application that it is hard to keep track of what kind of security or privacy policies need to be applied. Data tuples can be retrieved, transformed, sent, and stored back into different tables or different databases. With a large and complex system, it can be extremely difficult for the service developers to be aware of all the policies

that apply to data they are using, especially if the data is constantly being transformed and moved.

To solve this problem, Montag applies the notion of information flow control (IFC) [8, 9, 12, 18, 19] in a novel way to provide data access control. Data tuples in a database are tainted with labels describing what kind of data it is. For example, medical records can be tainted with the labels `RAW-DATA` and `MEDICAL-DATA`. Anytime that data is retrieved and flows through an application, the requests that touch that data or are created on behalf of that data are tainted as well. This is done by a reverse proxy that sits in front of the containers of each microservice, which manage the taints for every request currently being handled by that service. At any of the proxies, checks can be performed to make sure the current service does indeed have the correct permissions to handle the target labels. If not, the HTTP message is dropped by Montag , and the service never receives that data.

1.3 Design Criteria

Much work has been done with similar motivations with information flow control (IFC) systems. However, IFC systems are based on adding and checking labels on OS level resources [8, 9, 12, 19]. This is cumbersome and difficult for microservice developers to reason about, and does not extend well to a system with multiple operating systems, resulting in very little deployment of IFC systems in actual applications. Some work has also been done on with policies on the tuple-level, although only with policies based on attributes [13].

Goals and Approach

The most important goal of Montag is to help reduce the accidental or malicious release of privileged information bypotentially compromised microservices. Our basic approach is to design a system for tainting data with labels in a cloud native application. It should be easily integratable with a microservice architecture, containerized and deployable with

standard container orchestration software, and be able to taint data across services and databases. To this end we apply the following approach:

1. We design our system to taint data as it moves around in a distributed system. If requests retrieve a data tuple from a database, then the request should be tainted with the labels of that tuple. If the handling of that request then results in a call to another service, then this outgoing child request should also be tainted.

2. It is very important that Montag is implemented with as few modifications to the microservices as possible. We want this to be implementation, framework, and language agnostic. In the most basic case, the service should not even know Montag is working in the background.

3. Services or boundaries should reject requests based on the existence or non existence or certain labels. This form of access control allows operators to program services to only accept a specific set of labels or to reject any requests with illegal labels, and each service or application boundary can have its own access controls.

These three goals come with some assumptions about the application and the data.

1. We assume that sensitive data in the system can be feasibly labelled. New data can be labelled during generation and then saved into a database with the label attached, or automatically labelled based on which table that data was saved into. Preexisting data in can be labelled correctly by conducting a one time system wide audit requiring operators to read the table schema of every table and determine what kind of data it contains.

2. We assume that the application is cloud-native and built with the latest industry standard distributed application standards like containerization, container orchestration, and a basic service mesh with distributed tracing. At the very least, the application must be modern enough so that adding these technologies is not an impossible feat.

3. We can trust the application to be well-behaved, that whatever must be implemented in the application is done so correctly and in good faith. We trust that none of the developers of the application are malicious. The next section details why we make this assumption.

Security

Data in distributed systems present a difficult challenge when it comes to security. Its large application surface area means companies typically focus on securing only the microservice perimeter, and not between microservices themselves. However this means that even a single compromised service can be sufficient for exploiting data through multiple services. The desired solution is to place strict ACLs between each microservice. This is not always feasible to do in practice because operators are often times not aware of the dependencies between services, a problem that is compounded by the constant patching and updating that microservices undergo with high regularity. The constantly changing nature of the distributed applications mean that it can be difficult to update data policies accordingly.

We consider the following **threat model** in Montag : attackers may compromise and run arbitrary code in the application container of any service (due to some flaw in the application-level code), but are unable to compromise Montag itself (under the assumption that the implementation of Montag is loaded into a sidecar container separate of the application container). We assume that the attacker's goal is to use the compromised service to access data that this service should never be able to access in a privilege escalation attack.

The assumption that attackers may compromise services creates the necessity that Montag not only be implemented with minimal modifications to the service code as stated in Goal 2, but also that access control is completely separate from and not the responsibility of the service. If fact we cannot trust anything in the container in which the service is running to perform the necessary checks. One reason for this is that we assume an attacker can gain shell privileges into the container, and thus can bypass the any checks that are happening in

the container. Another reason is that by definition, the application is not part of the trusted computing base for Montag .

However we can trust that the creator of the service to be well-behaved and that whatever must be implemented in the application is done so correctly and in good faith. For the small parts of logic where Montag does rely on service code, we trust that the developer has implemented that logic correctly. So in short we trust that once the service does gain access to the data after passing all necessary checks, services will correctly do its part in making sure the labels for that data are correctly attached as it flows to any downstream or upstream services.

Chapter 2

Design

This section describes the details of the system architecture and how it meets all of our goals and criteria. We discuss in detail what Montag does to each request at each service without explicitly detailing how it manages the taint for the parent and child requests.

2.1 Tainting

The idea for Montag is built on top of the model for distributed tracing. Distributed tracing works by attaching a `x-request-id` header to each request that flows in the system. As this request flows from microservice to microservice, the `request-id` is passed as well. This builds a *trace* for the request, similar to how a stack trace works for a single-service application. In Montag , each request is also associated with an `x-data` header, which contains the set of labels associated with the data in that message.

`x-data` works like a taint. Refer to Figure 2.1, which shows the timeline of a request handler on the vertical axis and communications with the upstream and downstream in the horizontal axis. Through the handing of a request, which we now call the *parent request*, the service can make calls to its downstream dependencies. We will call these the *child requests*. If the parent request is originally tainted with a labels L_0, it taints the child request with L_0

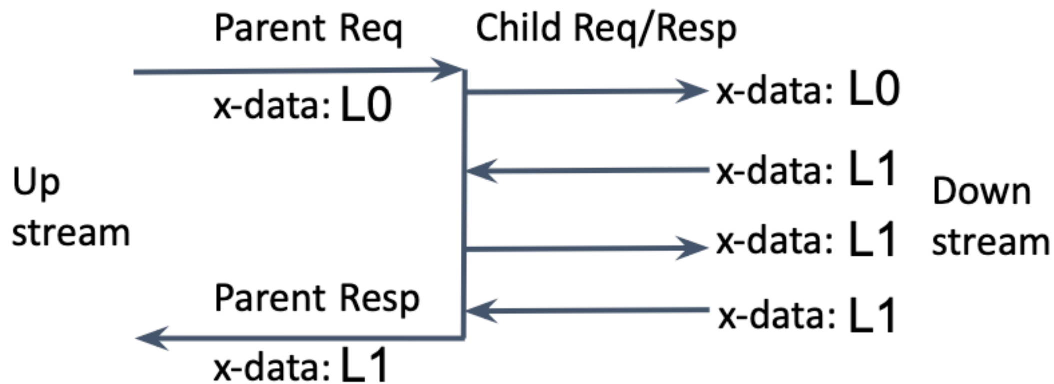

Figure 2.1: Tainting of parent and child requests

as well. If that child request returns a response with the taint L_1, then the original parent request now becomes tainted with the newer labels L_1. Now if that parent request makes a second child request, that second child request will have the newer taint L_1. If that child request does not change the taints, it should return L_1 in its response. Upon completion of the request handler, the parent request will generate a parent response which carries the taint level of the parent request at that point, L_1.

The end result is that the request always has the taints that describe what kind of that it is currently operating on or with. Even if the parent request does not store the data return by the downstream service, its possible that the parent request performs some action based on that data, which still taints the parent request.

However this protocol results in *overtainting*. Even if a parent request makes a request that operated on some privileged data, that does not mean that the response of that parent request will necessarily incorporate or even use that privileged data. Because the `x-data` label is encoded as an HTTP header, the application can actually eliminate overtainting itself and perform the management of the current request's taint by manually modifying or setting that header. If Montag detects that the header is not set when the request or

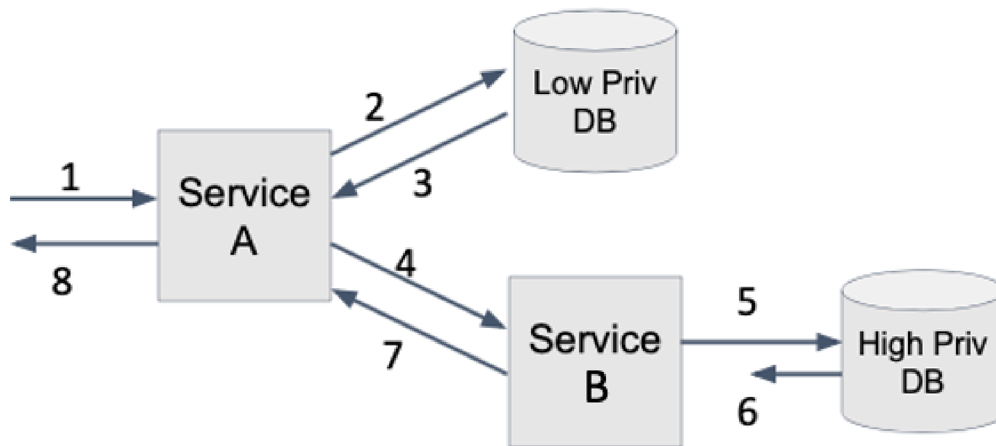

Figure 2.2: Demonstration of request tainting in the application

response leaves the service, it will automatically be set to the most recent labels that request (as identified by `x-request-id`) has. This is the part where we must trust that

1. the service correctly implements distributed tracing and forwards the `x-request-id` header anytime it makes a request.

2. the service acts in good faith when implementing its own taint override logic and applies the taint if the request does indeed use the privileged data.

The tainting happens at every service during the life of a request through the application. To illustrate this, we show an example of a request as it flows through a simple two service application in Figure 2.2. The the figure below, Service A receives a request from some client and then makes two sequential requests to a low privilege database and then Service B. Service B makes a request to high privilege database. The numbered arrows request the direction of the request and response, and the number is the order in which they occur.

1. Service A receives a request from an outside source. Here `x-data` is not set, so the incoming request has no associated data tuples of note.

2. Service A makes a request to the low privilege database. `x-data` is still not set.

3. This database only returns returns anonymized data tuples with the `x-data` header set to `ANON-USER-DATA`. This now sets the taints for the parent request at service A to be `ANON-USER-DATA`.

4. Service A makes a request to Service B, and `x-data=ANON-USER-DATA` is passed.

5. Service B makes a request to the database, and `x-data=ANON-USER-DATA` is passed.

6. The database retrieves raw user financial data. It returns a response that adds the label `RAW-FINANCIAL-DATA` to `x-data`. Thus the response from the database has the header set to `ANON-USER-DATA; RAW-FINANCIAL-DATA`. A check at Montag for Service B verifies if it can access `RAW-FINANCIAL-DATA`. For this demonstration, lets assume that it cannot and thus the response is dropped. Service B can either receive an HTTP 5XX error or the connection is simply severed, depending on the implementation.

7. Service B handles the connection issue and responds to Service A with an error message indicating the failure. The header in this response is just `x-data=ANON-USER-DATA` since Service B never ended up receiving the raw financial data.

This protocol works even in the face of a asynchronous type request. It is possible for step 4 occurred before the response from step 3 if Service A does not wait for the response from the Low Priv database before issuing the call to Service B. Then it should rightfully have `x-data` unset because that request would not have been made with any knowledge of the data returned from the low privilege database.

However an interesting problem occurs if any service is not stateless. That is, the service stores state or data from a request past the life of handling that request. For example, a service might return a list of the past ten actions performed, where each action is a result of a request with varying permissions. Then the list, which is stored in memory, should have privilege equal to at least the highest privilege of these ten most recent actions. In this case,

the application itself must be responsible for not only storing the data (here, the actions) but also the taints for that data.

2.2 Configuration

Developers and policy experts have the ability to configure Montag to perform actions at each service proxy that help maintain an accurate tainting and access control. These actions are `ADD`, `REMOVE`, `ENSURE_INCLUDE`, and `ENSURE_EXCLUDE`.

- `ADD`: adds a specific label to all HTTP messages either entering or leaving the service
- `REMOVE`: removes a specific label, if it exists in the `x-data`, from all HTTP messages either entering or leaving the service
- `ENSURE_INCLUDE`: for all HTTP messages either entering or leaving the service, ensure that a specific label taints that message. If not, then drop that message.
- `ENSURE_EXCLUDE`: for all HTTP messages either entering or leaving the service, ensure that a specific label does not taints that message. If it does, then drop that message.

The first two actions are useful to configure Montag to be responsible for adding or removing labels for services. For example, a service that anonymizes financial data may choose to remove the label `RAW-FINANCIAL-DATA`. This action is performed for all requests passing through the service (although in the future can be configured to target specific paths rather than all) and thus is not as useful for databases where different queries can return different labels. For the database, we use a different database specific reverse proxy that is discussed later.

The later two actions are used to configure access control at the service boundary. It can be used to ensure low privilege services do not ever receive high privilege data, or to ensure only data marked with an export OK label can egress from the application.

2.3 Application Edge Cases

The design for Montag unfortunately does not work as well for some specific edges cases without additional input from the service. Here we provide two examples of the more serious and likely common edge cases, and explain how we built in overrides mechanisms to solve these.

1. The first case occurs when the developer of the service has more context about what data is sent with each HTTP message than the proxy does. In a nutshell, Montag tries to attach labels to HTTP messages that correctly reflect what kind of data went into the construction of that message, and it does so using tainting which admittedly can easily lead to over tainting.

   Consider the case where a request handler, at the end of handling a request, simply emits a log message by making a call to a logging service. This message to the logging service will automatically have whatever labels are tainting the parent request, as it is technically a child request, but does not necessarily carry any sensitive data to warrant that.

2. The second case occurs for concurrent messages. One downside of Montag is that the proxy actually serializes all messages passing through it, since each parent request only has one set of labels. This is not usually a problem, as we should automatically taint child requests with the taints of the parent request at the time the child request is made. This makes sense because the child request is made from the same context as the parent, and so should get the same tags.

   However an issue arises when we consider that the taints are not actually applied when the child request is made, but rather when it passes through the proxy. It is technically possibly for two child requests which are created in the service code concurrently (and thus should have the same taints) to actually pass through the proxy at different times and end up with different taints, due to the unpredictable nature of scheduling and

executing concurrent code. One way this happens is if the taints for the proxy change between the time that concurrent child request 1 (C_1) and child request 2 (C_2) pass through the proxy, which can happen if the response to C_1 returns before C_2 is actually sent to the proxy.

Concurrent calls to downstream services are a common paradigm for asynchronous web servers. For example, the popular framework Express using Node.JS is built on running mostly asynchronous code while using promises and callbacks for some synchronous behavior. In this paradigm, the handler can perform other meaning full while it waits for the downstream service to reply, such as make calls to other services. Developers who run concurrent code now need some way to guarantee that this sort of unpredictable behavior does not happen.

Previously we already mentioned one such way a developer can break the abstraction of letting Montag handle all the tainting by manually setting the `x-data` header themselves. Developers can do this if they have a good understanding of the tainting system and feel confidently enough to handle managing all or some of the tainting themselves.

In response to edge case 1, if the developer knows that the log message does not contain any taint worthy data then they can set the empty header themselves to override any automatic tainting that Montag will perform. In response to edge case 2, a developer who executes concurrent requests can first find out the labels currently tainting the parent request and then manually override the `x-data` of both concurrent child requests with those labels. That way, both child requests are guaranteed to be sent with the same labels independent of their ordering by the scheduler. A developer will be able to find the labels currently tainting the request by either recording what the labels were from the previous HTTP child response, by looking at the labels from the parent request if no previous child requests have been made, or by pinging an echo server which should return a response with the labels in the header.

Another mechanism that gives the developer control over the tainting system is the

`x-data-override` header. Where as setting `x-data` allows a developer to completely override a header, setting `x-data-override` allows he or she to perform specific actions to the taints of an HTTP message. The header is set to a string of actions, formatted as shown below, and is removed once the message passes through the proxy. The header can be set on HTTP child requests, which means the actions are applied as the requests head downstream, or on HTTP parent responses which apply the actions as the response heads upstream.

An example of an `x-data-override` header. It can perform the same actions as can be configured into a proxy, but are only applied to specific HTTP messages passing through the proxy instead of all messages. These actions are performed after the actions for all messages in the proxy are applied.

```
ADD(LABEL1);REMOVE(LABEL2);CHECK_INCLUDE(LABEL3);CHECK_EXCLUDE(LABEL4)
```

The benefit of the `x-data-override` header is that it gives the developer more granular control over how to manipulate the tainting system. It is useful in the cases where the developer does not know or need to know all the taints of a message, but only care about remove or adding a specific label to a specific message. It does however require the developer to have more knowledge of what the labels mean than the coarse grained override mechanism.

Chapter 3

Implementation

3.1 Kubernetes and Istio Background

Taints are handled by a proxy that sits in front of each service. To give some context, in modern cloud-native applications, services are implemented with a set of logically identical containerized applications. Each application is a server that listens on a specific port for requests. Requests are made to the abstract name of the service and then routed or load-balanced to any one of possibly many containers serving that service. We built Montag on top of the container orchestrator Kubernetes, an industry standard tool responsible for deploying and managing containerized microservice based applications.

Normally requests made to a service hostname (i.e. `service-name.default.k8s-cluster`) will be automatically load-balanced to one of the many containers for that service. In Kubernetes, containers are not actually deployed individually but rather as *pods*, which is a small set of containers and the smallest unit of work. Oftentimes in simple applications, each pod is just a single container and so the two concepts are identical, but more advanced applications have a pod comprised of one or two main application containers and some number of *sidecar* containers [6]. The main containers include the application logic, such as web servers and workers, and the sidecar containers

include any application-specific or infrastructure tools that run alongside the main container. Examples of common sidecars are log forwarders (like Logstash or Fluentd), authentication servers, network proxies, or reload servers responsible for detecting changes in configuration and reloading the main application.

Kubernetes is able to load balance requests to the service abstract name to pods through the use of IP tables. After the abstract name is resolved to a service IP (which does not actually route anywhere), that IP is used for lookup in the IP table for routing. A Kubernetes minion, (called the Kubelet), which sits on every VM as a privileged container will write the IP table in a way such that the service IP will translate to the actual IP of one of the pods registered under that service. Then, the packet will be routed to that pod.

Montag hijacks this process with another industry-standard tool called Istio, which injects Envoy sidecars into each service [1,2]. Istio installs a service mesh on top of the microservice application deployed with Kubernetes. From RedHat [4], one of the pioneers of the modern service mesh,

> A service mesh is a way to control how different parts of an application share data with one another. Unlike other systems for managing this communication, a service mesh is a dedicated infrastructure layer built right into an app. This visible infrastructure layer can document how well (or not) different parts of an app interact, so it becomes easier to optimize communication and avoid downtime as an app grows.

The service mesh is, at its core, an essential network tool that allows the infrastructure operators to introduce simple application-level software designed networking functionality into a microservice application. It offers programmable routing rules, traffic shaping, fault injection, logging, and policy based firewalls – all through an out-of-process architecture that does not involve any additional work from the developer. Istio does this by installing Envoy, the L4 and L7 proxy, as a sidecar to every pod. This proxy intercepts all packets entering and leaving the pod and is able to modify or drop those packets. As a stateful reverse proxy,

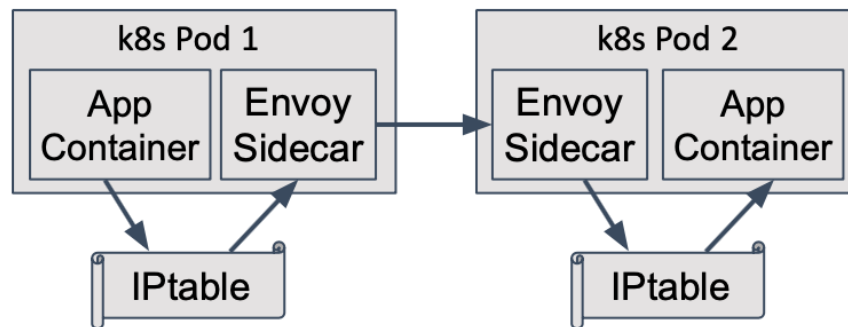

Figure 3.1: Architecture of the Envoy proxy

it is also able to intercept TCP streams and reconstruct HTTP connections, so the proxy can implement TCP and HTTP specific functionality.

Envoy performs this interception by hijacking the IP table so that all packets get redirected to the proxy. It then rewrites the packet destination and forwards the packet to the Envoy proxy of the downstream pod. That proxy will again rewrite the packet destination and forward it to the target container. This is shown in Figure 3.1.

We originally had concerns about adding two extra hops per transmission, which comes with an extra 4 controls transfers between user and kernel space per packet. All of this is not even built on kernel bypass or zero copy networking, and so each control transfer incurs expensive copying of the packet data. However, upon investigating this issue, we have found that the industry largely believes that the performance overhead of the service mesh is acceptable given the functionality that it provides. It is interesting to see this as a direct application of the end-to-end principle, with the argument that it provides much needed functionality that is not feasible to do at the end hosts.

Envoy implements its functionality as a series of filters. Each packet is sent through filters that can modify or drop that packet. The filters can also be optionally stateful, so the TCP Proxy filter and HTTP Connection Manager filter keep the necessary state required in order to reconstruct the stream and connection, respectively.

3.2 Montag Reverse Proxy

Montag adds an additional filter to Envoy that builds on the HTTP Connection Manager, which is considered a subfilter, that processes the HTTP request and response. It is responsible for keeping track of the latest `x-data` for a given `x-request-id` and for connecting any outbound child requests to their parent based on what `x-request-id` the child is carrying. So, as long as the service correctly implements distributed tracing and passes the `x-request-id` from parent to child, Montag will automatically also attach the lastest `x-data` so far. It will also update the `x-data` for the request id on response from the child, if necessary.

Shown below is an annotated and greatly simplified Python version of how the proxy logic works. The actual filter is written in C++ and contains optimizations that reduce copying and the synchronization required for concurrent threads to execute the same filter in addition to logic related to performing the add, remove, and check actions. It does not include all logic, like overrides, the details around garbage collection and applying policies, and some edge cases.

```
data = {}
conns = {}
parents = Set()

# function called on inbound or outbound request
def onRequest(req):
    # save a mapping from conn ID to request ID
    conns[req.conn.id] = req.id
    if req.id not in parents:
        # new incoming parent request
        parents.add(conn.id)
        if req.data:
```

```
            # save data for new parent request
            data[req.id] = apply_policies(req.data)
    else:
        # new out going child request
        if not req.data and req.id in data:
            # load current data if not overridden
            req.data = apply_policies(data[req.id])


# function called on inbound or outbound response
def onResponse(resp):
    # load request ID for the connection because
    # x-request-id does not have to be in the response
    reqID = conns[resp.conn.id]
    if resp.conn.id in parents:
        # outgoing response to a parent request
        if not resp.data and resp.id in data:
            # load current data if not overridden
            resp.data = apply_policies(data[resp.id])
    else:
        # incoming response to a child request
        if resp.data:
            # save data in response, if necessary
            data[resp.id] = apply_policies(resp.data)
```

The function `apply_policies` takes in a set of labels and apply the relevant actions specified by the user's configuration. It is also aware of it the function was called from an ingress or egress context. In the case the action is `CHECK_INCLUDE` or `CHECK_EXCLUDE`, the function is able to sever the current connection.

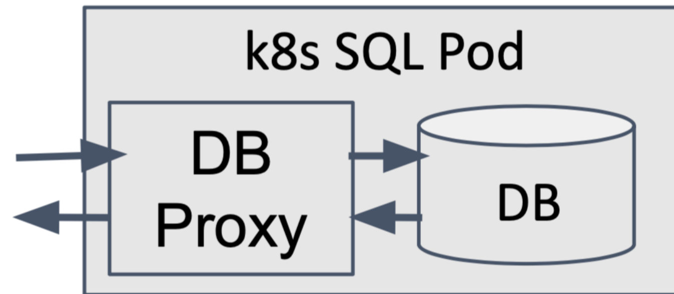

Figure 3.2: Structure of the DB Proxy sidecar

3.3 Montag Database Proxy

Our second component is responsible for initially loading the `x-data` when data is retrieved from the database. To do this task, we developed our own database layer which uses a proxy (Figure 3.2) that actually performs the query execution. The proxy works by accepting the query to execute over HTTP, along with the authentication required to execute this query. It modifies the query to also select all the labels of all the rows that the query would touch. Then, if finds the superset of labels used in the execution.

For example, if the following query was received

```
SELECT users.name, records.email
    FROM users, records
    WHERE users.id = records.id
```

it would be rewritten to

```
SELECT users.name, records.email, users.labels, records.labels
    FROM users, records
    WHERE users.id = records.id
```

Then the database proxy finds the superset of all labels from all rows of both labels columns, and fills the `x-data` header of the response with that superset.

Currently our proxy logic only works for relational databases, but can be extended easily to NoSQL as well. For example, labels can be associated with every key or set of keys in a key value store, or associated with a specifics files or directory in a filesystem like blob storage service.

3.4 Deployment

Because all of Montag is deployed with Kuberentes, the configuration and deployment is done through Kuberentes API objects defined by Custom Resource Definitions (CRDs). These are custom resources that allow you to define, configure, and deploy your own Kubernetes objects. Istio exposes a set of new custom resources, of which Montag uses the EnvoyFilter object.

The first step to deployment is to simply change the Envoy proxy that is deployed with Istio. Depending on the method you are using to deploy Istio, this can simply be a search and replace of the container image to our custom Montag Envoy fork. Istio works by using a Kubernetes admission control service that automatically injects the necessary Envoy proxy as a sidecar into every pod when the pod is created. We can turn on this admission control by labelling the namespace we are using with `istio-injection="enabled"`.

Once Envoy is deployed with every pod, we then can configure Envoy to turn on our Montag filter with the EnvoyFilter object.

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: service-A-filter
spec:
```

```
workloadSelector:
  labels:
    app: reviews
configPatches:
# The patch adds the custom filter to the listener/http connection manager
- applyTo: HTTP_FILTER
  ... some configuration omitted for brevity
  patch:
    operation: INSERT_BEFORE
    value:
     name: netsys.data_tracing
     config:
      actions:
       - operation: ADD
         member: test1
         when: EGRESS
       - operation: CHECK_INCLUDE
         member: test2
         when: INGRESS
```

Part of the configuration allows the developers to specify what actions the filter should perform when a message is received or sent. This filter adds the label `test1` to every HTTP response leaving the service and checks for the existence for the label `test2` on every HTTP request received.

The configuration can also be selectively applied to only specific services with the workload selector. This filter only applies to services that have the "`app: reviews`" Kubernetes label set.

Chapter 4

Evaluation

In order to evaluate the success of this project, we tested its functionality extensively against our test application, which was built off the Istio book reviews application. We will first discussed the modifications we made to the application so that it fit the requirements to properly test our system. Namely, that it contains a variety of different services that accesses multiple data sources and generates data, which may or may not be anonymized by the services.

Our system should be able to effectively enforce declarative global data policies with few changes to the application and no changes to the service infrastructure. Quantitatively, the addition of the data taints should not significantly affect the performance of the system that is already using Istio. We measured the impact of our service by determining the latency between services and comparing to the latency of the system without using our service.

4.1 Test Application

The application that we are using to test our system's functionality is based off the Istio book reviews application. The application has four separate microservices: the product page, details, reviews, and ratings services. The product page microservice calls the details

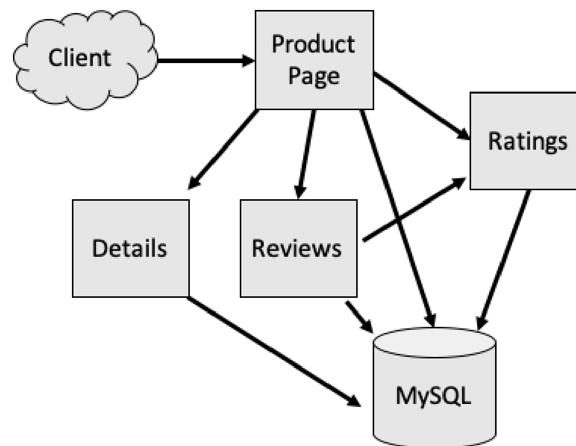

Figure 4.1: Architecture of the Bookinfo application

and ratings services to populate the page that is being served. The details service produces book information, like its author, title, ISBN number, etc. The reviews service retrieves the reviews associated with a product and can call the ratings service to receive the star rating out of five for the product. The dependency graph for this application is shown in Figure 4.1.

We modified the system to be database-backed for all data retrieval. In particular, we have added a database table to store reviews, which was previously hard-coded. In order to support our service that enforces data regulation, we had to modify the table schemas such that each row was tainted with a set of labels, which indicated what kind of data the row stored.

The second major change we introduced into the Bookinfo application is modifying the database client to be HTTP-based. Rather than building a custom MySQL client to use the MySQL protocol, we opted instead to send database queries as POST requests.

4.2 Experimentation

To test the impact of Montag on the performance of an application, we run two experiments to measure the additional latency that Montag adds. The first experiment was to measure the end of end latency of our modified Bookinfo under three conditions:

1. Bookinfo alone
2. Bookinfo with Istio and Envoy
3. Bookinfo with Istio, Envoy, and Montag

We were most interested in the 99th percentile latency, because we found that 50th percentile latency had no significant difference. Our experiment consisted of executing 1000 requests for the endpoint at a concurrency of 50 requests. We used to tool `hey` [14], which is an HTTP load generator and measurement command line tool meant to replace the Apache tool `ab` for HTTP2. We choose the endpoint because that is the endpoint exposed to the user (as opposed to the other endpoints which are API endpoints) and it invokes all of the API endpoints anyways to generate the result.

Our results in Figure 4.2 show that while adding Istio and Envoy produce a small amount of overhead (2%), additionally adding Montag has a near zero additional overhead at just 0.4%. However this result was not conclusive in telling us the actual overhead of Montag . The overhead incurred in an end to end system is much too dependent on the workload. We could have easily constructed an application with hundreds of serialized services required to completed the request and very little work at each service, and that would incur a much more significant overhead as using each service would require passing the proxy twice. We could have also constructed an compute intensive application where the time spent at each service is an order of magnitude larger, in which case the overhead of Montag would be dwarfed by that of the compute time.

Thus, our next experiment tries to test the overhead of just a single proxy itself, there by eliminating as many workload dependent variables as possible. We set up an HTTP web

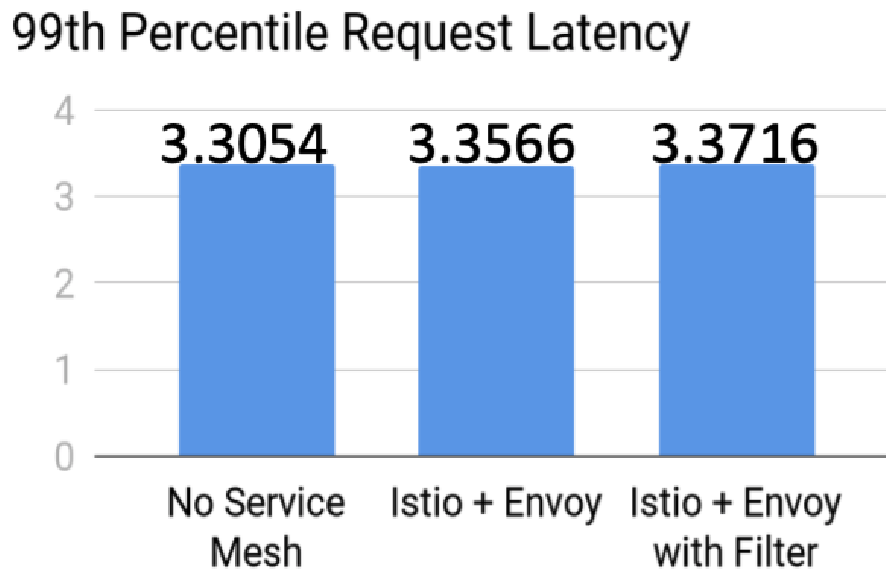

Figure 4.2: 99th Percentile End-to-End Latency, in Seconds

server, and sent it HTTP requests with no body, getting back HTTP responses with no body. Because our filter only looks at the header, the goal is to isolate as best as possible the overhead of just Envoy. The HTTP web server we used is a standardized test server called HTTPBin [11]. In order to test this in a realistic environment, we set up a pod with HTTPBin served by Envoy. Then we set up another pod with a shell but without Envoy, and we make requests from the shell to HTTPBin with `hey`. While this measurement also includes the round trip latency from the shell to HTTPBin as well as the service time at HTTPBin, we think that it is the most realistic testing environment. Experiments where Envoy is tested in isolation without Istio, where the measurement is only how long it takes Envoy to process a packet, do not consider factors like the switch between user and kernel space as well as the copying of the packet in memory. Our requests in Figure 4.3 show that adding Istio and Envoy incurs expensive overheads of more than 165% on top of just the application itself. However the overhead of adding Montag once you already have Istio and Envoy is rather small, at only 26.8% before optimizations. At first these numbers may seem

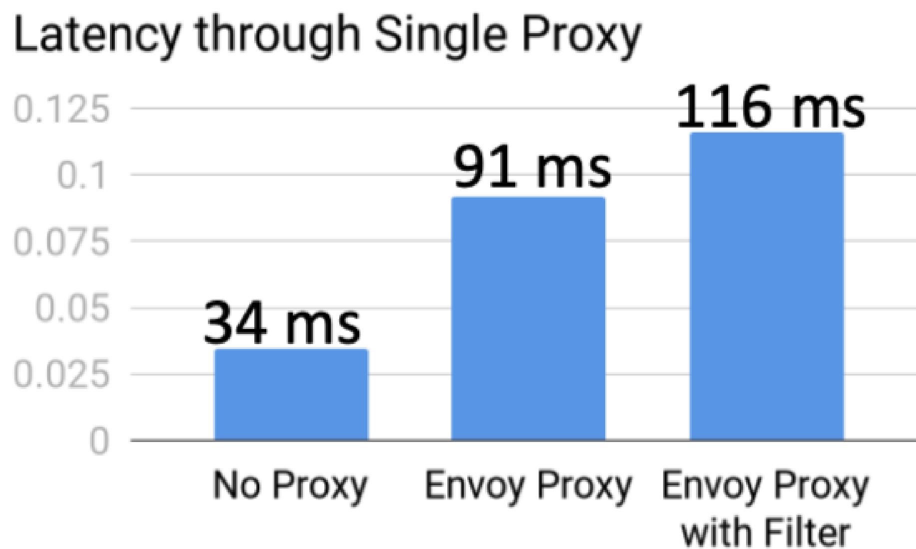

Figure 4.3: 99th Percentile through a Single Proxy, in Seconds

scary, since the overhead is quite significant. However the results from both experiments are in line with previous well known open source experimentation done on Istio and Envoy [3]. The trade off is end to end - an operator must decide whether the performance overhead is worth functionality that Montag , and to an extend the service mesh as a whole, will bring.

4.3 Code Efficiency Comparison

One of the key goals of Montag is to allow for simple integration with any existing application. In order to test our proficiency in meeting this goal, we have recreated the same data privacy enforcement functionality within the application itself. Relative to the size of the application, we can then compare the number of additional lines necessary to use Montag versus implementing the functionality internally.

The modified Bookinfo application, which contains four microservices, is composed of roughly 750 lines of logic. This will serve as a frame of reference for the number of modified

lines of code. In order to support Montag , we require only that the developer specify the privacy level of each table in the database, which amounted to a total of 5 lines of code for Bookinfo. In contrast, in order to support the same logic by implementing it within the application itself, it required roughly 100 lines of additional code, which is 13% of the size of the entire application.

In addition to the increased number of lines of code, implementing the data privacy regulation logic within the application can lead to both maintenance hassle and many potential sources of error. In order to maintain this logic, each service must be aware of the current architecture being used to enforce data regulations because the service must determine the appropriate privacy label. In the case that the architecture changes, this change needs to be reflected in every service. This leads to the second major downside: it is prone to error. Although we assumed that the services are well-behaved, it is difficult and possibly impossible to enforce that each service has implemented the data regulation properly. Thus, services can misbehave and break data regulation policies without ill-intent.

The most important downside is probably that by implementing in the application you break the abstraction of out of process policy enforcement, since data needs to be received by the application before it the access control can be enforced. This is dangerous not only because of least privilege, but also because than any attacker with shell access to the container will be able to bypass the permission check. Recall that from Section 1.3, we have a thread model where the attacker has shell access to an unprivileged container and wishes to access privileged information.

The performance upside of implementing in the application is that you can skip the overhead of needing to run a service mesh and achieve performance similar to that of the non mesh case in Figure 4.2. However we strongly believe that this is not a good enough reason in order to implement enforcement in process, and that the benefits on an easy to manage, hands off, and secure Montag is more valuable.

Chapter 5

Discussion

5.1 Future Work

Other Protocols

Montag currently only works over HTTP, as it is the easiest to implement distributed tracing with. However distributed tracing can be implemented over gRPC connections and even raw TCP connections. In order to make Montag a practical system that can be deployed over any microservice based applications with minimal changes, it must also support at the very least gRPC, Thrift, and TCP based connections natively.

Control Plane

One original goal for Montag was to make it easy to declarative implement with a global control plane, where operators simply specify the access rights for each service and the service is automatically updated with those rights. In this, it would be easy to implemented role based access control (RBAC) at the service level. The control plane would be made of a Kubernetes controller. Users would interact with this control by uploading *Custom Resources*. Custom Resources and Custom Resource Definitions are a Kubernetes functionality that allows an

operator to define their own resource type and create controllers which respond to updates to those resources.

Montag would have a `DataAccessRole` object which defines the roles, each of which have a unique access level. It would also have a `DataAccessRoleBinding` object responsible for binding the selected role to a service or a group of services, based on some selector. This is analogous to how role based access control work in Kubernetes today. A controller will watch for new role bindings, and then update the roles of the services that has changed. These updates will be pushed to the data plane (the Envoy proxies), which makes management of the policies at each service much easier.

By using both a control plane and a data plane, we take a lesson from software design networks and can create a massively scalable system. The control plane is very light and only needs to work when changes are applied, and the data plane handles all the policy decisions and thus scales well with the number of pods running.

General Improvements

There are a variety of improvements that can be made on the existing Envoy and Database proxies. Envoy needs to be configured to return a standardized HTTP 5XX message instead of simply severing the connection, which will help the upstream service respond to the permission failure by differentiating permission failure from a network failure. There are also general optimizations that can be made, such as improving the global mappings by removing the global lock and decreasing the amount of string copying that needs to be done.

On the database side, we plan to rewrite the entire database proxy in Golang. We also want to create language specific clients with the same interface as the SQL clients for that language. Because we are currently restricted to HTTP only, we must write these clients to allow applications to communicate with the database layer over HTTP. If we are able to get tainting to work over TCP, then we will be able to use the standard TCP based SQL clients.

5.2 Related Work

There have been numerous systems and strategies to enforce data privacy regulations. Below, we will discuss in further detail a few of the representative and most well-known options. However, as we will see, each of these tackles a different problem or is not feasible for implementation in a real-world distributed system. Thus, Montag provides a unique value not seen in these systems.

Qapla

Currently, the most widespread method of enforcing privacy regulations is through database ACLs. A recent example of a system that follows the same structure as database ACLs is Qapla [13]. Qapla is a layer over the database that rewrites queries in order to enforce policies on data access. These policies are specific to which row or column was accessed. However, this method falls short when considering the flow of data through multiple services. Database ACLs only limit the direct readers and writers for a set of data, and there is significant application logic necessary to maintain the privacy rules as the data flows through multiple services.

Thoth

Thoth [9] is a system with similar ideas as Montag , but is implemented as part of the operating system. Thoth defines a conduit boundary around processes of a service where data can only pass through this boundary via conduits. These conduits are files, network connections, and key value stores. Syscalls to access these conduits, either to egress or ingress data, are intercepted and policies around ingress or egress are applied. These policies can be specific to the exact network connection (as identified by the 5-tuple), file path, or key accessed.

As an OS module, Thoth is difficult for developers of microservices to deploy. Although the Thoth model allows for distributed processes on multiple nodes, it is still quite difficult to

install custom OS modules on all operating systems across all nodes. Additionally, with the advent of containers, using keys, 5-tuples, and file path as keys can be problematic. Finally, Thoth policies are possibly not expressive enough as it is applied based on how services get data (or from what source), rather than what kind of data is passed.

Riverbed

Riverbed is a framework to enforce privacy policies in web services [17]. Both solutions are designed with modern, cloud-native applications in mind and aim to reduce the load on developers. However, the two systems differ in the types of policies that they enforce and who bears the load of specifying the policies. Riverbed provides users with the ability to define privacy controls about their data. They allow broad controls per user to control their data, such as whether a user's data can be stored persistently or sent over the network. This is user-facing and requires any users who would like to set privacy controls to do so themselves. In contrast, Montag gives developers the ability to implement data privacy policies regarding the contents of the data itself. It implies that the application has provided the user with a privacy policy in accordance to relevant regulations, and Montag aids in the actual enforcement of the promised privacy policies. We believe this to be a more realistic approach rather than giving users the ability to choose their privacy and then requiring applications to perform remote attestation.

Asbestos & HiStar

There are a few systems that follow the same structure as Montag . Namely, Asbestos and HiStar use data tainting in order to enforce privacy rules. An important fact to note is that both of these systems differ from Montag because they do not assume well-intention from the developer.

Asbestos is an operating system that tackles the same type of problem of data security [8]. In particular, Asbestos focuses on the issue of data leakage from inter-process communication

and system-wide information flow. In order to ensure that data is not leaked across users, Asbestos uses a labeling mechanism that designates the sensitivity of data. In order to protect information flow, the operating system is given the responsibility of tracking the labels associated with each piece of data. Thus, only the kernel is required to properly read and pass along the labels. By using kernel-enforced labels, Asbestos prevents against faulty or malicious logic in applications, servers, or any other point of failure. It can also simplify the logic necessary to enforce data policies.

Another alternative is proposed by HiStar [18], which is built on top of Asbestos labels and can be considered a successor to Asbestos. Using this formulation, data is labeled according to various taint levels. HiStar also has no superuser with unique untainting abilities, read access, or write access within the system.

However, both HiStar and Asbestos are OS kernels, and their uses differ from our goals. We would like to implement a system that can be used to enforce data regulation policies within existing applications with a targeted industry audience. The requirement to replace all or part of the kernel that industry applications run on is a huge barrier to the adoption of operating system level IFC. We also assume then that the problem is less focused on malicious / incorrect behavior and more targeted towards reducing the effort necessary to enforce regulation policies. Thus, we would like to impose as little impact on the user as possible.

5.3 Conclusion

Montag is a system for globally deploying and enforcing data access control at the microservice level. Montag uses taint labels to store the privacy status of each piece of data and is built on industry-standard infrastructure, Kubernetes and Istio. Unlike existing solutions, our system provides service-to-service data access enforcement that requires very few modifications to the existing system. Additionally, we have seen that application performance is not significantly impacted by adding Montag on top of the overhead that already exists

from running a service mesh.

Montag stems from a line of work on general policy enforcement in web applications and making guarantees about how data is being used. We believe that data is a fundamental building block of distributed systems with special handling required in the infrastructure rather than just in the application. As industry moves towards a smarter, more programmatic standard for infrastructure, we expect that this idea will continue to be explored in greater depth.

Bibliography

[1] Envoy. https://www.envoyproxy.io retrieved 05/09/2019.

[2] Istio. https://istio.io/ retrieved 05/09/2019.

[3] Istio Performance and Scalability. https://istio.io/docs/ops/deployment/performance-and-scalability/ retrieved 12/17/2019.

[4] Microservices: What's a service mesh? https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh retrieved 12/17/2019.

[5] A. Booth. Charities hit with fines for sharing donors' data without consent, december 7, 2016. https: //nakedsecurity.sophos.com/2016/12/07/charities-hit-with-fines-for-sharing-donors-data -without-consent/ retrieved 05/09/2019.

[6] B. Burns. *Designing Distributed Systems: The Sidecar Pattern.* O'Reilly.

[7] Docker Inc. What is a Container? https://www.docker.com/what-container retrieved 05/09/2019.

[8] P. Efstathopoulos, M. N. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, M. F. Kaashoek, and R. T. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 17–30, 2005.

[9] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Con-*

ference on Security Symposium, SEC'16, pages 637–654, Berkeley, CA, USA, 2016. USENIX Association.

[10] A. Gupta. Microservice Design Patterns. http://blog.arungupta.me/microservice-design-patterns/, retrieved 05/09/2019.

[11] kennethreitz. httpbin: A simple http request response service. https://httpbin.org retrieved 12/17/2019.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[13] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy Compliance for Database-backed Systems. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 1463–1479, Berkeley, CA, USA, 2017. USENIX Association.

[14] rakyll. hey: Http load generator. https://github.com/rakyll/hey retrieved 12/17/2019.

[15] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly Media, 2015.

[16] P. Sayer. German Consumer Groups Sue What- sApp Over Privacy Policy Changes, January 30. http://www.pcworld.com/article/ 3163027/private-cloud/german-consumer-groups-sue- whatsapp
-over-privacy-policy-changes.html retrieved 05/09/2019.

[17] F. Wang, R. Ko, and J. Mickens. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 615–629, Berkeley, CA, USA, 2019. USENIX Association.

[18] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. *Commun. ACM*, 54(11):93–101, 2011.

[19] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[20] K. Zetter. Hackers Finally Post Stolen Ashley Madison Data, August 18, 2015. https://www.wired.com/2015/08/happened-hackers-posted-stolen-ashley-madison-data/ retrieved 05/09/2019.