HyShield: Enforcing Service Level Data Access Control with Data Tainting

Rachel Li[†] Will Wang[†]

[†]UC Berkeley

ABSTRACT

Distributed and containerized systems have become incredibly popular in recent years and its easy to understand why. One of the major benefits it brings is the ability for teams to work independently on individual microservices. This is often hand in hand with work flows like agile development [5] and mantras like "move fast and break things," but in the face of strict security and privacy regulations such as GDPR [15], it's those exact benefits that make managing data in distributed systems tremendously difficult. In particular, independent teams have no clear visibility into how other downstream and upstream services handle data and the fast pace of development often prioritizes policy compliance in favor of feature development.

In this paper, we present HyShield, a system for globally deploying and enforcing data access control at the microservice level without intrusive application code level modifications. HyShield uses taint labels to store the privacy status of each piece of data and is built on top of the existing popular orchestrator and service mesh infrastructure, Kubernetes and Istio. Our system differs from existing solutions and database ACLs that it deals with policies after the data leaves the database and can be enforced at every service-to-service hop. Our system is unique in that it is designed with modern cloud-native applications in mind. We hope that it can be used by compliance experts with minimal software engineering experience to configure the correct level of data access control for each microservice, without having to think about how the services are connected.

The rest of our paper is structured as follows: first, we discuss the problem at hand and the system we have built, HyShield. Then, we discuss the test application that we've adapted and an evaluation of its ability to meet our success criteria. Finally, we finish by discussing future improvements to HyShield and related work.

1. INTRODUCTION

The modular design of microservice applicants works by breaking a larger application into its individual functions and then packaging each function into separate containers [8] 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 [11]. 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 [17]. 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 functionality 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 than 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 manage data in a way that follows current and future regulations [6, 18, 22]. 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 thousands of complex independently-owned microservices, one cannot expect the service developer to know

for every piece of data, which are the services that have the correct permissions to receive this data.

HyShield takes a novel approach to data access control management. It taints each piece of data with a label that describes the permissions require to access that data. Then, at each service, a reverse proxy intercepts all HTTP requests and responses heading to that service. This proxy checks the permissions required to access any data that has come through request or response and then verifies that the service has the correct permissions in order to access that data. If not, the HTTP message is dropped by HyShield, and the service never receives that data.

1.1 Motivations

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 HyShield: 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 HyShield itself (under the assumption that the implementation of HyShield is loaded into a sidecar container separate of the application container). We believe this is a reasonable model, as HyShield code is not changing from day to day, and also has a significantly smaller attack surface than the microservice applications running atop HyShield.

Much work has been done with similar motivations with information flow control (IFC) systems. However, IFC systems are based on adding and checking labels, often on OS level resources [10, 13, 21]. This is cumbersome and difficult for 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 [19]. Some work has also been done on with policies on the tuple-level, although only with policies based on attributes [14].

1.2 Criteria

Goals

1. The first and most important goal of HyShield is to enforce data access policies at each microservice. For every piece of data that arrives at the service, no matter the source of that data, we want to control if a particular service is allowed to access this it.
2. It is very important that HyShield is implemented with as few modifications to the service as possible. We want this to be implementation independent and not rely on trusting the application to perform its own access control.
3. A third and slightly less important goal is to also enforce data egress policies when data leaves the service, either en route to the next service or to a destination outside the cluster. This goal is less important because as long as (1) all services use HyShield and (2) network level policies are enforced on any communication with endpoints outside the private cluster, then this goal is automatically achieved.

Constraints

1. It is very important that HyShield is implemented with as few modifications to the service as possible. We want this to be implementation independent and not rely on trusting the application to perform its own access control.
2. We can trust the application to be well-behaved and that whatever must be implemented in the application is done so correctly and in good faith.

This may at first seem confusing because the two seem like the same constraint, but there is actually a significant difference. We cannot trust the application itself, or, to an extent, 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 HyShield. However, we do trust that once the service does gain access to the data after passing all necessary checks, it will correctly do its part in making sure the labels for that data is correctly attached as it flows to any downstream services.

Assumptions

1. We assume that every piece of data in the microservice application can be labeled based on which table or source it was retrieved from. That is, data tuples of differing privileges or access rights are not mixed in the same table. At the very least, it must be feasible, given a query in the database for some data, to be able to tell what privilege is required. This labelling can either be constructed when the data is created (i.e. any data of

this type has these labels) or pre-determined ahead of time (i.e. any data in this schema has this label).

2. We assume that the application is cloud-native and built with the latest industry standard distributed application technologies 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.

2. HyShield

This section describes the design goals of HyShield, along with the constraints and assumptions. We then detail the system architecture and how it meets all of our criteria. Lastly, we discuss our design limitations.

2.1 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. This tuple fits within an HTTP request or response. 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.

2.2 Tainting

The idea for HyShield 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 HyShield, each request is also associated with `x-data` header, which denotes the highest privilege level required to access any data associated with this request. If a service receives a request (or response) with an `x-data` higher than what that service is allowed to access, then the service must drop that message.

`x-data` works like a taint. Refer to Figure 1, which shows the timeline of a request handler on the vertical axis and communications with the upstream and downstream in the horizontal axis. Anytime a request receives data tainted with a higher `x-data`, then that request is automatically labeled

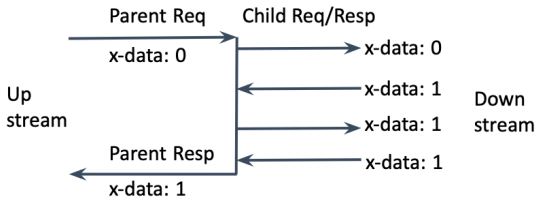

Figure 1: Tainting of parent and child requests

with that higher taint. Through the handing of this request, which we now call the *parent request*, the service can make calls to its dependencies. We will call these the *child requests*, which are requests made to another service during the handling of a parent request. If the parent request with original taint P_0 becomes tainted with a higher privilege P_1, any child requests it makes will automatically have the higher privilege taint P_1. If that child request returns a child response with an even higher taint level P_2, then the parent request is tainted P_2 upon receipt of that response. Thus if the parent request makes another child request, the second child request will now have the taint P_2. 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.

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 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 setting that header. If HyShield detects that the header is not set when the request or response leaves the service, it will automatically be set to the highest privilege that request (as identified by `x-request-id`) has seen so far. 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 tainting 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 3 service application in Figure 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.

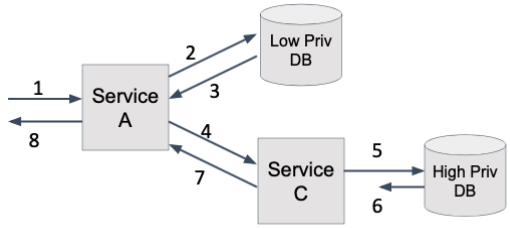

Figure 2: Demonstration of request tainting in the application

Here `x-data` is not set, so its assumed to be at the lowest privilege.

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

3. On response from the database, `x-data` is set to 1, indicating low privilege permissions required. Service A has these permissions.

4. Service A makes a request to Service B, and `x-data=1` is passed.

5. Service B makes a request to the database, and `x-data=1` is passed.

6. The database returns a response with `x-data` set to 2, indicating that high privilege permissions are required. Service B does not have this, 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 `x-data=1`.

This protocol works even in the face of concurrent requests. If step 4 occurred before the response from step 3, then it should rightfully have `x-data=0` because that request would not have been made with any knowledge of the data 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 10 actions performed, where each action is a result of a request with varying permissions. Then, the list should have privilege equal to at least the highest privilege of these 10 most request actions. In this case, the application itself must be responsible for not only storing the state and data (here, the actions) but also the taints for that data. This is a downside to data tainting through requests, as it violates our goal of being as application independent as possible, but so far we have no discovered a way to taint data in stateful services without explicit knowledge of the data that is stored.

2.3 Implementation

So far we have discussed in detail what HyShield does to each request at each service without explaining how it can manage the taint for the parent and child requests. In this section, we will explain the different components of HyShield along with how they perform the tainting and access control at scale. We will also discuss future implementation specific improvements that we had envisioned but did not have a change to implement.

2.3.1 Background

Taints are handled essentially by a proxy that sits in front of each service. To be specific, 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 HyShield 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 [7]. 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.

HyShield 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 microser-

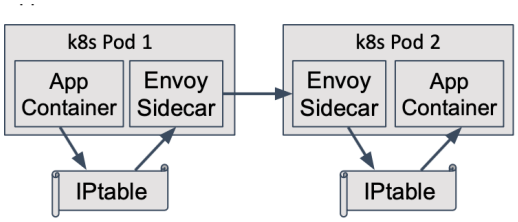

Figure 3: Architecture of the Envoy proxy

vice 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, 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.

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.

2.3.2 *HyShield Reverse Proxy*

HyShield 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 highest `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, HyShield will automatically also attach the highest `x-data` so far. It will also update the `x-data` for the request id on response from the child, if necessary.

Show 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.

```
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 parent request
        parents.add(conn.id)
        if req.data:
            # save data for new parent request
            data[req.id] = req.data
    else:
        # new child request
        if not req.data and req.id in data:
            # load current data if not overridden
            req.data = 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:
        # response to a parent request
```

```
        if not resp.data and resp.id in data:
            # load current data if not overridden
            resp.data = data[resp.id]
        # garbage collect
        del data[resp.id]
        del conns[resp.conn.id]
        parents.remove(conn.id)
    else:
        # response to a child request
        if resp.data:
            # save data in response, if necessary
            currData = data.get(resp.id, 0)
            data[resp.id] = max(resp.data, currData)
```

Additionally, this proxy is also responsible for performing the privilege checks in order to determine if the service at which it was installed is allowed to access a given tuple. Currently, this logic is implemented in the same filter as data tainting and simply drops the request if the permissions are not met. In the future, we plan to massively update this part of the design. A major goal we had in mind was to build a control plane from which operators can easily and decoratively install which services have the correct permissions. Currently the per-service permissions are specified by installing a different filter, each with a hard-coded permission level, at each service. First, we want to separate out the enforcement filter into a separate filter. Then we want to add a control plane component to either automate the process of installing versioned enforcement filters, which will greatly simplify the management of service level permissions.

2.3.3 *HyShield 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 4) 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 also parses the query and looks up the labels for the sources that the query touches. In our implementation, we targeted SQL databases and parsed our query for the tables that the query would touch. Then, we used a separate table to store the mappings from each table to its privilege level. This information could optionally have also been stored in a separate database, like a key-value store. When the proxy executes the query, it will then also look up the maximum privilege for any of the sources that the query touches and then return the response with the correct `x-data`.

3. 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.

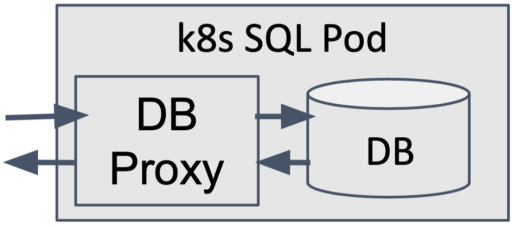

Figure 4: Structure of the DB Proxy sidecar

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.

3.1 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 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 5.

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 code such that each table was labeled with a taint, which indicated the level of anonymity of the data stored. For reviews, this was not anonymized data since it included the reviewer. Both the star rating and the details of the book are considered anonymized data.

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.

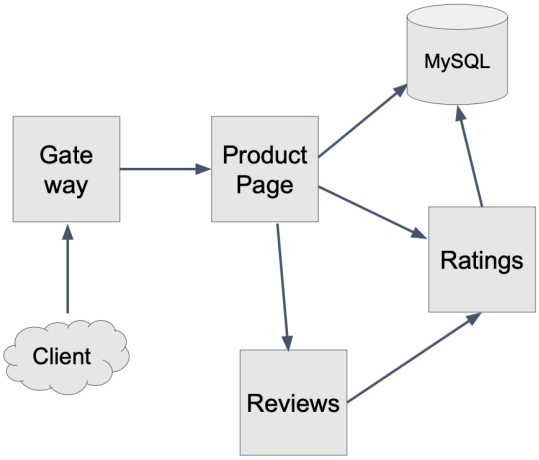

Figure 5: Architecture of the Bookinfo application

3.2 Experimentation

To test the impact of HyShield on the performance of an application, we run two experiments to measure the additional latency that HyShield 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 HyShield

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` [16], 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 6 show that while adding Istio and Envoy produce a small amount of overhead (2%), additionally adding HyShield has a near zero additional overhead at just 0.4%. However this result was not conclusive in telling us the actual overhead of HyShield. 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

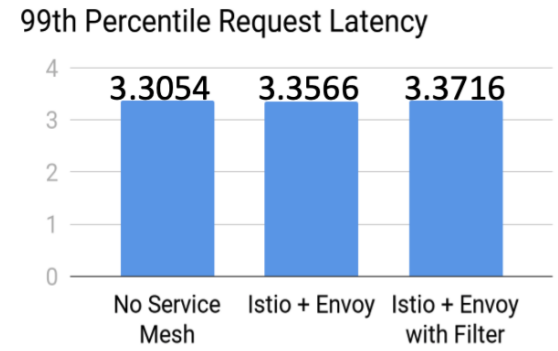

Figure 6: 99th Percentile End-to-End Latency, in Seconds

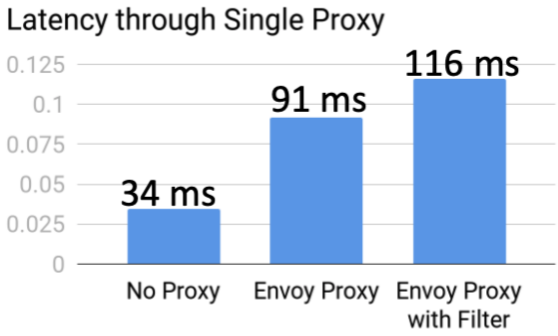

Figure 7: 99th Percentile through a Single Proxy, in Seconds

the overhead of HyShield 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 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 [12]. 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 7 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 HyShield once you already have Istio and Envoy is rather small, at only 26.8% before optimizations. At first these numbers may seem 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 HyShield, and to an extend the service mesh as a whole, will bring.

3.3 Code Efficiency Comparison

One of the key goals of HyShield 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 HyShield 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 HyShield, 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. Given this application, we see a 20-factor increase in the number of lines required to implement the same logic.

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 1.2, 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 6. 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 HyShield is more valuable.

4. DISCUSSION

4.1 Future Work

4.1.1 Open Policy Agent

Our biggest future work on this project is to remove enforcement from Envoy altogether. Instead, we have plans to build a filter which contacts a second per-pod sidecar responsible solely for the enforcement. We have already chosen to use the industry standard Open Policy Agent, an open source project whose sole purpose is to allow operators to easily write and manage rule based policies. OPA already has proven tight interactions with Istio and Envoy, and and will provide additional flexibility in not only what policies we can create. More importantly, OPA is set up so that management of the set of polices at each service can be done easily and centrally through a single source of truth.

A feature that OPA will allow us to provide that we highly anticipate is more granular labels. We can taint data with different categories of labels, such as `Anonymized`, `UserData`, `MedicalData`, `FinancialData`, etc., instead of simply using permission levels. This will provide incredibly fine-grained role-based access control at every single service.

4.1.2 Other Protocols

HyShield 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 HyShield 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.

4.1.3 Integrations with IFC

There are clear parallels with this work and other IFC work in recent years. Riverbed [19] is a related work 4.2 that uses remote attestation to allow clients to ensure that their data is not being used in ways that is inconsistent with the client's policies. A similar line of thought can further be applied to HyShield. At the very basis, HyShield is about tainting requests with labels for the data that the request is using or has used. These labels are be also thought of as the policies that need to be applied, and a Riverbed like IFC system can

then be used to make sure that the policies defined in the labels are followed by the service.

4.1.4 Control Plane

One original goal for HyShield was to make it easy to declarative implement with a global control plane, where operators simply specify the access rights for each service 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.

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's open policy agent through OPA's bundle API, and then OPA will be able to immediately execute the new policies.

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.

4.1.5 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.

4.2 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, HyShield provides a unique value not seen in these systems.

4.2.1 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 [14]. Qapla is a layer over the database that rewrites queries in order to enforce policies on data access. 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.

4.2.2 Thoth

One way to enforce privacy rules as data flows through services is to require that each service have knowledge about which services it can interact with. Along the same lines, an external system can have knowledge about each service's access level. An existing system that follows this type of privacy enforcement is Thoth [10]. Thoth is a kernel-level policy compliance layer that helps data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve. However, Thoth is an operating system component that intercepts interrupts involving data and thus is not easily deployed. Our system aims to enforce privacy rules at every hop while requiring minimal overhead to add to existing systems. In order to do this, HyShield passes along the access restrictions with the data itself and requires each service to check that it's able to access the data. By abstracting the common infrastructure into a separate system, developers will need to make minimal changes to their applications to meet privacy regulations. Rather than intercepting requests, HyShield only requires passing along an additional request header in order to enforce privacy regulations.

4.2.3 Asbestos & HiStar

There are a few systems that follow the same structure as HyShield. 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 HyShield 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 [9]. 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. Using this formulation, Asbestos gives unprivileged users fine-grained access to the data they request, which prevents data leakage across users. The main difference between our system HyShield and Asbestos is in the use-case. Asbestos focuses on allowing processes to serve multiple users without data leakage across users, which means that processes are labeled according to their access rights and the sensitivity of data already received. In contrast, HyShield only checks the incoming data sensitivity and does not require services to track what sensitivity of data they have received so far. We do not worry about services leaking data to each other as we assume the services are part of a trusted environment.

Another alternative is proposed by HiStar, which is built on top of Asbestos labels described previously [20]. Using this formulation, data is labeled according to various taint levels, which restrict read and write accesses. In particular, HiStar improves upon Asbestos by improving further the operating system design on top of the explicit information flow concept and removing the need for a superuser from an operating system. This means that there is no user with unique untainting abilities, read access, or write access within the system.

However, both HiStar and Asbestos are operating systems, 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. We assume then that the problem is less focused on malicious 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.

4.2.4 Riverbed

The most similarly designed solution to HyShield is Riverbed, a framework to enforce privacy policies in web services [19]. 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 all of 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, HyShield gives developers the ability to implement data privacy policies regarding the contents of the data itself. This is a more fundamental issue regarding the access level of data rather than user preferences. HyShield enforces necessary privacy regulation requirements at the application and does not require any input from the end user. We aim to restrict access to different privacy levels of data depending on the service, which is an internal decision not supported by Riverbed.

5. CONCLUSION

HyShield is a system for globally deploying and enforcing data access control at the microservice level. HyShield 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 HyShield on top of the overhead that already exists from running a service mesh.

HyShield stems from a line of work on general policy enforcement in microservice applications and making guarantees about how data is being used. There is a lot to be done with controlling data usage from an infrastructure level. 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 be explored in greater depth by industry leaders.

6. REFERENCES

[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] AgileAlliance. Agile 101. https://www.agilealliance.org/agile101/ retrieved 05/09/2019.

[6] 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.

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

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

[9] 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.

[10] 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 Conference on Security Symposium*, SEC'16, pages 637–654, Berkeley, CA, USA, 2016. USENIX Association.

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

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

[13] 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.

[14] 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.

[15] E. Parliament. Gdpr portal. http://www.eugdpr.org/eugdpr.org.html retrieved 05/09/2019.

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.