Mazu: A Zero Trust Architecture for Service Mesh Control Planes

Aashutosh Poudel
William & Mary
Virginia, USA
apoudel01@wm.edu

Pankaj Niroula
William & Mary
Virginia, USA
pniroula@wm.edu

Collin MacDonald
William & Mary
Virginia, USA
cmacdonald01@wm.edu

Lily Gloudemans
William & Mary
Virginia, USA
algloudemans@wm.edu

Stephen Herwig
William & Mary
Virginia, USA
smherwig@wm.edu

Abstract

Microservices are a dominant cloud computing architecture because they enable applications to be built as collections of loosely coupled services. To provide greater observability and control into the resultant distributed system, microservices often use an overlay proxy network called a service mesh. A key advantage of service meshes is their ability to implement zero trust networking by encrypting microservice traffic with mutually authenticated TLS. However, the service mesh control plane—particularly its local certificate authority—becomes a critical point of trust. If compromised, an attacker can issue unauthorized certificates and redirect traffic to impersonating services.

In this paper, we introduce our initial work in Mazu, a system designed to eliminate trust in the service mesh control plane by replacing its certificate authority with an unprivileged principal. Mazu leverages recent advances in registration-based encryption and integrates seamlessly with Istio, a widely used service mesh. Our preliminary evaluation, using Fortio macro-benchmarks and Prometheus-assisted micro-benchmarks, shows that Mazu significantly reduces the service mesh's attack surface while adding just 0.17 ms to request latency compared to mTLS-enabled Istio.

CCS Concepts

• **Security and privacy** → **Key management**; *Security protocols*; **Distributed systems security**.

Keywords

Cloud Computing, Microservice Security, Service Mesh, Registration-Based Encryption

ACM Reference Format:
Aashutosh Poudel, Pankaj Niroula, Collin MacDonald, Lily Gloudemans, and Stephen Herwig. 2025. Mazu: A Zero Trust Architecture for Service Mesh Control Planes. In *The 18th European Workshop on Systems Security (EuroSec'25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3722041.3723100

EuroSec'25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1563-1/2025/03
https://doi.org/10.1145/3722041.3723100

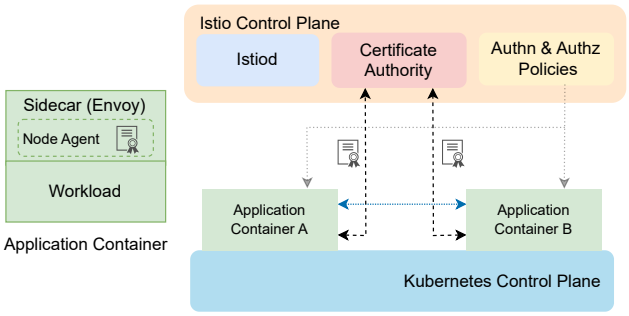

Figure 1: Architecture of the current Istio service mesh.

1 Introduction

In cloud computing, a common software architecture is *microservices*: rather than deploy a large, monolithic application, the software developers decompose the application into a distributed system of small, loosely-coupled services, that communicate via well-defined interfaces. The primary benefits of microservices are twofold: *elasticity* (the cloud can scale each component independently), and *isolation* (a failed component does not, by itself, cause other components to fail).

The microservice architecture, being inherently distributed, presents unique challenges compared to monolithic applications in terms of reliability, observability, and security. A *service mesh* is middleware that addresses these challenges, allowing each microservice to focus exclusively on its application logic. In most service meshes, each microservice runs in an application container (see Figure 1) and is paired with a proxy container, known as a *sidecar*. The sidecar modifies the host's network routing so that all traffic to and from the application container flows through it. This design enables the sidecar to manage microservice traffic—providing features such as authentication, authorization, and logging—without requiring any modifications to the application itself.

Problem. A key feature of service meshes is *zero trust networking*, where the sidecars tunnel communication between microservices using mutually authenticated TLS (mTLS). To support this, the service mesh control plane acts as a certificate authority (CA), issuing certificates to the sidecar proxies. Additionally, this control plane also provisions sidecars with authorization rules that define which peer sidecars are allowed to connect. Although zero trust networking stymies an attacker's ability to laterally move in the

victim's network, if an attacker compromises the service mesh's control plane, *the attacker can issue rogue certificates, impersonate applications, and redirect traffic to malicious endpoints—effectively subverting the entire system.*

In this paper, we ask the following research question:

Is it possible to deprivilege the service mesh's local CA while maintaining microservice compatibility and performance?

One of the earliest attempts to eliminate CAs—and the public key infrastructure altogether—is *Identity-Based Encryption* (IBE) [30], where a user's public key is a meaningful identifier, such as their email or IP address. Unfortunately, IBE replaces the CA with an even more powerful *key authority* that generates all private keys, and hence constitutes a *key escrow*. Related approaches, such as self-certified keys [17, 28] and certificateless public-key cryptography [1], remove the key-escrow problem while retaining IBE's ability to *encrypt to an identity*, but still place CA-like trust assumptions on an authority. Recently, some research systems [2, 11, 35] run critical control plane components, such as the CA, in trusted execution environments (TEEs), notably Intel SGX [26]. While TEEs protect the CA from certain classes of attack, the CA remains unchanged and thus highly privileged.

We present Mazu,[1] our early-work extension to existing service meshes that removes the need for a trusted CA. Mazu leverages recent advances in *Registration-Based Encryption (RBE)* [19] to enable microservices to generate their own keys locally while registering a global binding between their public key and microservice instance. RBE addresses IBE's key escrow problem by replacing the key authority with a less powerful entity called the *key curator (KC)*. Unlike a key authority, the KC does not possess secret keys; instead, it manages the system's public parameters, including cryptographic commitments that bind public keys to identities. While RBE's obvious use case is encrypted messaging, we instead use it as a cryptographically verifiable registry to enhance service authentication.

Although the KC functions as a "drop-in replacement" for a CA in terms of architecture, its use in a service mesh introduces several additional security challenges. First, in traditional service meshes, services periodically rotate their TLS certificates to mitigate the impact of key compromise. However, achieving similar post-compromise security guarantees with RBE is not straightforward given the binding commitment of an ID to a public key. Second, RBE assumes that the KC somehow authenticates an ID's initial registration, and disallows unauthorized unregistering or re-registering of that ID. In contrast, Mazu's threat model allows the KC to engage in Byzantine behaviors, including malicious re-registration attacks. Mazu addresses these concerns by amending RBE with protocol enhancements that make both the registration process and interactions with the KC publicly auditable.

Contributions. We make the following contributions:

- We introduce Mazu, a service mesh that enhances zero trust networking for microservices by eliminating the local CA, a critical weak point. Mazu leverages recent advances in registration-based encryption to replace the CA with a less privileged, untrusted entity.
- We implement an initial Mazu prototype using the widely adopted Istio service mesh [6] and Envoy sidecar proxy [5], ensuring compatibility with existing microservice architectures. Our implementation requires no modifications to applications or the Kubernetes cluster manager.
- We conduct a preliminary evaluation of Mazu using the Fortio load tester and micro-benchmarks for profiling the node agent. Our preliminary results show that Mazu adds a modest increase of 0.17 ms in request latency compared to Istio with mTLS enabled.

Outline. This paper is organized as follows. In §2 we specify our threat model, goals, and assumptions. We provide an overview of RBE in §3, and then present the design of Mazu in §4. In §5 we describe our early-work implementation of Mazu using the Istio service mesh in a Kubernetes cluster, and in §6 evaluate the performance of Mazu against traditional service mesh configurations. We then discuss future work to reduce the trust assumptions on Kubernetes in §7, and highlight related work in §8. Finally, we conclude in §9.

2 Goals & Assumptions

We assume a service mesh architecture that reflects popular implementations like Istio [6], Linkerd [8], and Consul [4]. Specifically, the service mesh logically comprises a *control plane*—which handles traffic rules, logging, authorization, and certificate issuance—, and a *data plane*—namely, the sidecar proxies. We assume the sidecar acts as a layer-7 (HTTP or gRPC) proxy. Within the Kubernetes cluster [7], each node (a physical or virtual machine) runs a mesh *node agent*, which serves as an interface between the service mesh's control plane and data plane.

2.1 Threat Model

We assume an attacker has remote code execution on the cluster and can exploit the service mesh's control plane. In particular, the attacker can issue rogue certificates, spawn microservices to impersonate legitimate applications, and configure routing rules that redirect traffic to these malicious microservices.

We assume an attacker can temporarily compromise a node agent or sidecar, and can thus leak sensitive data, such as keys and credentials. The attacker may also compromise a microservice itself, but cannot escape its container to fully take over or replace its associated sidecar.

We consider availability attacks as out-of-scope, as an attacker who breaches the service mesh's control plane can trivially deny service.

Kubernetes assumptions. We trust Kubernetes to manage and provision the cluster and assume the attacker cannot compromise Kubernetes' control plane, including its API server. A key aspect relevant to Mazu is Kubernetes' default role in issuing a signed JWT *admin token* to each microservice during initialization. This token contains the service's internal URL, along with IDs for its cloud service account, node, and pod. It is primarily used to bootstrap the

[1] Mazu is a Chinese sea goddess and the deity of seafarers.

service by granting access to additional resources. For example, in a traditional service mesh, the node agent presents this token to the CA to obtain a certificate.

While Mazu assumes Kubernetes properly manages (e.g., grants and revokes) these tokens, we assume that an attacker may steal them from a compromised node. We further make the assumption that Kubernetes does not use the same token for multiple services, and assigns services distinct IP addresses (both of which are true in a conventional deployment where a pod hosts a single service). In §7 we describe our future work to reduce trust in Kubernetes.

2.2 Goals

Our primary security goal when designing Mazu is:

S1 *Untrusted Service Mesh Control Plane*: An attacker that breaches the service mesh's control plane must not be able to undermine the confidentiality or integrity of the microservice application's network communications.

Additionally, we have the following functional goals:

F1 *Application transparency*: Mazu should preserve the core property of service meshes, which is that the application itself remains unmodified.

F2 *Compatibility with existing service meshes*: Mazu should extend current service mesh software (we use Istio).

F3 *Low performance overheads*: Mazu should impose little performance overhead on the microservice applications, both in terms of client latency and resource usage.

3 Registration-Based Encryption Overview

3.1 Overview

At a high-level, Mazu achieves its goals by replacing the CA with a decentralized protocol based on the recently introduced concept of Registration-Based Encryption (RBE) [16, 19]. RBE is an alternative to IBE that replaces IBE's key authority with a weaker principal called the *key curator* (KC) that does not have knowledge of any secret key (or any secret information). A user in an RBE system locally generates their keys and then publicly registers their identity and corresponding public key with the KC. In response, the KC updates the public parameters of the system and returns to the user some supplementary, non-secret, information called the user's *opening*, which is necessary for the user to decrypt ciphertexts. As new users register and the public parameters change, existing users need to contact the KC to fetch their updated opening. Encryption and decryption work analogously as in IBE: given the public parameters and Bob's identity, Alice can non-interactively encrypt a message for Bob. Bob, given his secret key and his opening, can decrypt Alice's message.

3.2 Parameters & Algorithms

Mazu uses the RBE implementation from Glaeser et al. [19], which is the first (and to our knowledge, only) efficient implementation of RBE. To make our presentation of Mazu self-contained, we list the RBE parameters in Table 1, and briefly present the cryptosystem's high-level interface, prefacing each algorithm with the principal that invokes it (or using the notation $[A \to B]$ in the case of a network request from A to B). The RBE system consists of the following algorithms:

Table 1: RBE Parameters. The gray cells are the public parameters pp. The KC maintains C and Λ, as these change upon each new registration.

	Parameter	Description
Public	N	Max number of registered users
	bg	Bilinear group with generator g of order p and pairing $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$
	crs	Common reference string
	C	Aggregation of public keys ($\prod_{i\in[N]} \mathsf{pk}_i$)
	Λ	User openings ($\{\Lambda_i : i \in [N]\}$)
User$_i$	id_i	Registered identity
	pk_i	Public key
	sk_i	Private key
	ξ_i	Helping information
	Λ_i	Opening

Setup(N, bg) → pp
We assume the Kubernetes admin runs the Setup to initialize the system's public parameters pp. The initialization of the common reference string crs requires the admin to generate a secret number, which it can then purge after computing the crs.

U$_i$.KeyGen(pp) → (pk_i, sk_i, ξ_i)
A user U$_i$ that wants to register first executes the KeyGen algorithm to generate their keypair (pk_i, sk_i). The algorithm also uses the crs to output the *helping information* ξ_i, which is U$_i$'s contribution to the openings of the other users.

[U$_i$ → KC].Reg(pp, id_i, pk_i, ξ_i)
To register an identity id_i, user U$_i$ sends their public key and helping information to the KC. The KC then updates the commitment of public keys as $C' \leftarrow C \cdot \mathsf{pk}_i$, and likewise the openings of the other users: $\forall_{j\in[N\setminus i]} \Lambda_j \leftarrow \Lambda_j \cdot \xi_i[j]$.

[U$_i$ → KC].Upd(pp, id_i) → (C', Λ'_i)
A user U$_i$ periodically makes an Upd request to the KC to fetch their updated opening Λ'_i, as well as the updated aggregation of all public keys: C'.

U$_i$.Enc(pp, id_j, msg) → ct
User U$_i$ encrypts a message msg $\in \mathbb{G}_T$ to a user with ID id_j using the Enc algorithm, which also takes as input the public parameters pp.

U$_i$.Dec(pp, sk_i, Λ_i, ct) → msg User U$_i$ decrypts a message encrypted to their ID id_i using the Dec algorithm, which takes as input the public parameters pp, the user's private key sk_i, the user's opening Λ_i, and the ciphertext ct.

KC.MProve(pp, id_j, pk_j) → π_j Any user (though typically the KC) produces a proof of membership π that shows that id_j is registered under public key pk_j using the MProve algorithm.

U$_i$.MVerify(pp, id_j, pk_j, π_j) → bool Any user U$_i$ can validate a proof of membership π_j certifying that id_j is bound to public key pk_j using the MVerify algorithm.

4 Design

We first present an initial, insecure version of Mazu, and then gradually improve the design to counter distinct attacks. Figure 2 shows our final design.

4.1 An Initial Approach

ID registration. By default, Kubernetes provisions each microservice instance with a Kubernetes-signed *admin token* containing the instance's internal URL, as well as the IDs for its cloud service account, node, and pod. The pod's node agent uses the token to register the ID:

$$\mathsf{id}_i = H(\mathsf{token}_i),$$

for a co-resident service i, where H is a cryptographic hash function. The node agent then generates a self-signed TLS certificate for the service that embeds the admin token.

TLS handshake. During the TLS handshake between two sidecars, the client sidecar verifies that the server's certificate includes a signed admin token for the intended destination URL. Additionally, the client sidecar derives the server's ID id_s as a hash of the received token. To verify that the server registered id_s, the client encrypts a challenge nonce to id_s, and passes the challenge as an extension to the initial `ClientHello` message of the TLS handshake. If the server can decrypt and return the nonce in the subsequent `ServerHello` TLS message, validation succeeds and the client continues with the connection. The server's validation of the client works in an analogous fashion. The remaining steps of the TLS protocol are unchanged.

Issues. The initial approach verifies that an endpoint both possesses a valid token for a given service (URL) and knows the corresponding private key. However, this method remains vulnerable if an attacker compromises the endpoint's token and its private key sk_i. (In practice, the system must treat the token as public since any client can initiate a TLS handshake and retrieve it.)

Additionally, beyond the risk of RBE key leakage, this approach is also susceptible to a re-registration attack, where an adversary unregisters an endpoint's ID and re-registers it with a keypair of their choice. Indeed, unregistering id_i simply implies removing the public key from the global aggregation of public key commitments:

$$C' \leftarrow C \cdot \mathsf{pk}_i^{-1}$$

and likewise removing the relevant index of ξ_i from each user's opening. Note that the KC need not expose a separate Unreg API: if an attacker $\mathcal{A}$ wants to unregister id_i and re-register it as their own, they need only register with a "public key" of

$$\mathsf{pk_i}^{-1} \cdot \mathsf{pk}_{\mathcal{A}}$$

and similarly for ξ_i.

4.2 Removing the Threat of Key Compromise

The key to mitigating the risk of an RBE key compromise is recognizing that Mazu only requires RBE's registration properties, not its *encryption with respect to identity* functionality, and thus the choice of the private key is critical only during registration.

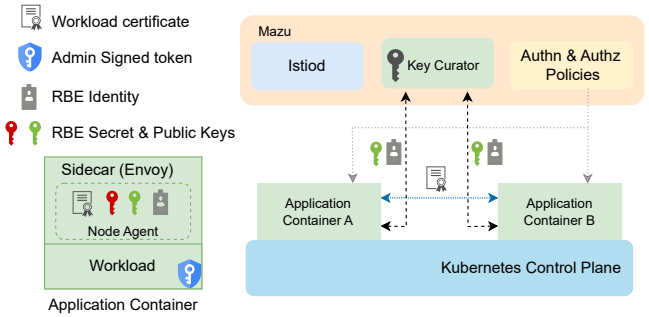

Figure 2: Mazu architecture

Specifically, when registering id_i, the node agent locally generates a keypair, choosing as its private key:

$$\mathsf{sk}_i = H(\mathsf{token} \parallel \mathsf{IP}),$$

where IP is the service's IP address.

As before, during TLS certificate validation, the client extracts the server's token from its self-signed certificate, and computes the server's ID as:

$$\mathsf{id}_s = H(\mathsf{token}_s),$$

As before, the client encrypts a challenge nonce to id_s, but now retrieves the server's opening Λ_s from the KC. The client then derives the server's expected "secret" key $\widetilde{sk_s}$ as:

$$\widetilde{sk_s} = H(\mathsf{token}_s \parallel \mathsf{IP}_s)$$

and locally checks:

$$\mathsf{Dec}(\mathsf{pp}, \widetilde{sk_s}, \Lambda_s, \mathsf{Enc}(\mathsf{pp}, \mathsf{id}_s, \mathsf{nonce})) \stackrel{?}{=} \mathsf{nonce}$$

If the client can decrypt the nonce, validation succeeds and the client continues with the connection. The server's validation of the client proceeds in a likewise fashion, and Mazu leaves the rest of the TLS handshake unchanged.

Stolen Tokens. Suppose an attacker $\mathcal{A}$ leaks B's token and tries to impersonate B. For now, we assume $\mathcal{A}$ cannot acquire the same IP as B, but that $\mathcal{A}$ can compromise the service mesh control plane to route services to itself. During the TLS handshake, the peer service will derive an incorrect candidate secret key due to the mismatch in the server's IP, fail to decrypt the nonce, and thus abort the connection.

IP reuse. Suppose now that Kubernetes tears down B, allowing $\mathcal{A}$ to reuse B's IP to launch a malicious service. If a peer service connects to this malicious service, the peer will derive the exact same RBE private key as for the retired B, and Mazu's custom TLS validation checks will pass. To counter this threat, we note that Kubernetes automatically invalidates a service's token when tearing down that service. Thus, we amend the sidecar's certificate validation to also query Kubernetes for the validity of a token.

4.3 Mitigating Re-registration Attacks

While Mazu does not prevent re-registration attacks, it does make such an attack detectable. For each registration, the KC updates the user openings Λ; the node agents periodically poll the KC for the updated openings for their resident services. We modify this

operation so that the KC returns to the node agent a history of updates, where each history entry specifies an individual update as the tuple:

$$(\text{token}_i, \text{id}_i, \text{pk}_i, \pi_i)$$

In this way, ID re-registration is auditable: any node agent can verify the proof of registration π_i and further detect instances of token reuse.

5 Implementation

We port Glaeser et al.'s [19, 29] Python-based implementation of RBE (which depends on the `petrelic` [18] Python wrapper to the RELIC cryptographic library [3]) to Go. Our implementation uses Cloudflare's `Circl` cryptographic library [14], and specifically its implementation of the BLS12-381 elliptic curve. We additionally implement the cryptosystem's proof of registration algorithm, and provide a Google Protocol Buffer's serialization of parameters and messages. In total, our Go implementation is 1050 LoC.

We introduce the following high-level changes to *Istio*:

- *Control Plane*: We replace *Istiod's* CA functionalities with a *Key Curator*.
- *Data Plane*: We modify *Envoy*'s certificate validation to use RBE's registration properties.

Istiod. Istiod acts as a CA, issuing certificates to enable secure mTLS communication between services. The CA is implemented as a gRPC service that receives certificate signing requests (CSRs) from node agents, validates their credentials, and generates workload certificates [10]. In Mazu, we likewise add the KC as a gRPC service within Istiod, providing methods to register a service's RBE ID and fetch updated public parameters from the system.

Node agent. Mazu treats each service as a user in the RBE scheme and uses the node agent to generate an ID and keying material $(\text{pk}, \text{sk}, \xi)$ for each service on that node. Instead of contacting the CA to sign the CSR, the node agent registers the RBE ID with the KC. Once registration is complete, it creates a self-signed certificate that embeds the token as a certificate extension. The node agent then provisions the Envoy proxy with the certificate using Istio's standard xDS service discovery protocol, which uses a UNIX domain socket connection between the agent and the proxy [32].

The node agent periodically polls the KC for updates to the system's public parameters and retrieves the history of all registered services. Using this information, it precomputes the results of the trial encryption-decryption (see §4.3) for all registered services (non-interactively). This precomputation is an optimization, as it obviates the need for the Envoy proxy to perform additional network requests and cryptographic operations within the critical path of the TLS handshake. The node agent periodically sends the Envoy proxy updates to its opening, the RBE public parameters, and the precomputed challenge-response results using the same local xDS service.

Envoy proxy. The Envoy proxy acts as the data plane component, managing traffic between all services within the service mesh. We implement a custom certificate validation extension in Envoy. Specifically, during the mTLS authentication, our extension extracts the admin token from the peer p's certificate and validates it using the Kubernetes `TokenReview` API. If the admin token is valid,

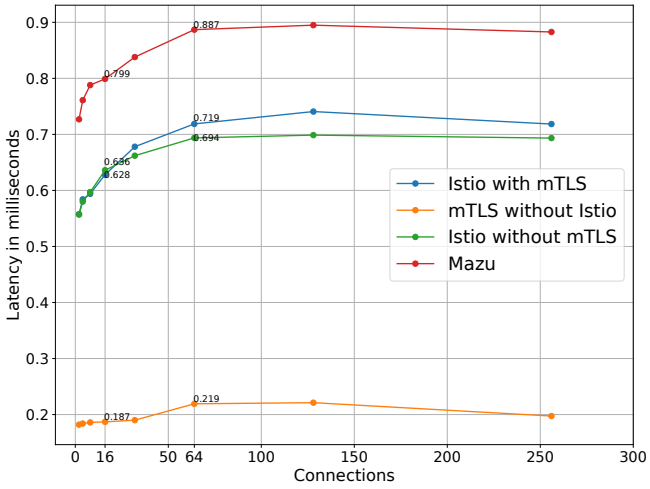

Figure 3: Latency versus connections at the 90th percentile (p90), with a request rate of 1000 RPS over 120 seconds.

Envoy then retrieves the p's IP address from the TLS connection stream, and checks the validity of the $(\text{token}_p, \text{IP}_p)$ tuple against the pre-computed challenge-response results it received from the node agent.

6 Evaluation

Experimental setup. We perform our experiments using a local minikube Kubernetes cluster on an on-premise server. We configure the clusters with 4 CPUs and 16 GB of memory. Our server has a 4^{th} generation AMD EPYC chip (7354P) with 64 logical cores and 786 GB of RAM. Mazu extends Istio version 1.24 and Envoy Proxy version 1.32.

Macro-benchmark. We evaluate Mazu's end-to-end performance using a "Two pods benchmark test," which measures data path performance through latency measurements across different proxy configurations. The benchmark uses two Fortio pods (a Go-based load testing tool) in a client-server setup, with the client sending parameterized echo requests to the server. Key parameters include client connections, mutual TLS status, payload size, request rate, protocol type, sidecar configuration, and request distribution. We compared four configurations: baseline (using mTLS and no Istio), Istio with sidecars using mTLS, Istio with sidecars using plaintext (no mTLS), and Mazu.

Our tests use the HTTP/1.1 protocol with 1 KB payloads, a request rate 1000 requests per second (RPS), and concurrent client connections ranging from 2 to 256. As shown in Figure 3, Mazu introduces minimal latency overhead, adding 0.17 ms latency at both 16 and 64 concurrent connections compared to Istio with mTLS enabled.

Micro-benchmark. We conduct micro-benchmark experiments to evaluate the performance impact of retrieving key updates, from both latency and size perspectives. We log the latencies and update sizes that a node agent observes using Prometheus, and then export this data for further analysis. As Figure 4 shows, as more

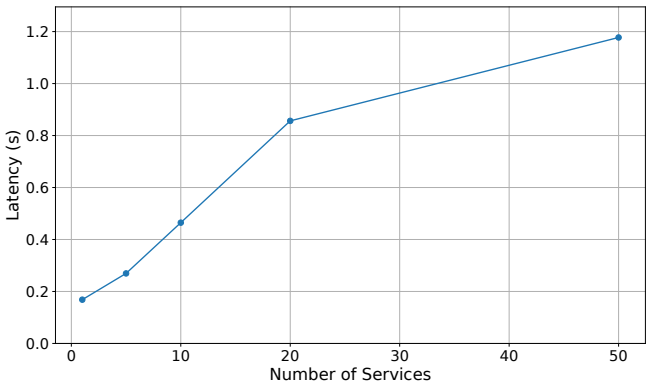

Figure 4: Key Update Latency Micro-benchmark Results. We measure key update latencies across minikube clusters with 1 to 50 active services.

services register with the KC, the latency of an update request grows at an approximately linear rate. We note that the response sizes themselves grow at a sub-linear rate.

7 Limitations & Future Work

7.1 Byzantine Key Curator

A remaining threat in Mazu is a Byzantine KC that provides microservices with inconsistent registration views. For instance, a malicious KC could divide the service instances into two sets, and facilitate an attack where an adversary steals a token from one registration set, and registers that token in the other. This is an example of fork linearizability [15], which typically requires communication between the two sets to detect and resolve the forked views. Given that the service mesh control plane exerts control over routing, achieving communication between the two sets may be challenging.

As future work, we plan to explore solutions like TrInc [23], which leverage basic trusted hardware (e.g., TPMs) to integrate monotonic counters into network protocols. Applied to Mazu, this approach would require the KC to attach an attested counter to each update, making equivocation and network partitioning attacks detectable.

7.2 Reducing Trust in Kubernetes

Mazu relies on Kubernetes in two key ways: *(1)* as a trusted issuer of tokens when initializing a service, and *(2)* as an oracle during the TLS handshake to verify whether a token is valid or invalid. While the former likely requires the integration of a hardware root of trust, the latter is within the purview of RBE, as we describe next.

Token invalidation. As described in §4.3, during the TLS handshake, the sidecar extracts the token from the peer's certificate and queries Kubernetes to check its validity. In this setup, the proxy depends on Kubernetes to correctly revoke a service's token when the service terminates and to provide up-to-date, correct revocation status.

To eliminate the need for Kubernetes to handle these trusted operations, Mazu could be extended to use a separate RBE ID-space with a logically separate key curator KC_{rev} for registering token revocations. Specifically, when a node agent detects that a service on its node has exited, it registers an ID with the value $H(\text{token})$ with KC_{rev}. After this registration, the node agent can safely discard the secret key, as Mazu relies on the registration solely for revocation tracking, not for encrypting messages.

During certificate validation, the proxy must check whether the peer's token is registered with KC_{rev}, and abort the TLS handshake if it is. Similar to the normal KC, the node agent can periodically poll KC_{rev} for a history of revocation updates, and verify the membership proofs that accompany the updates. Alternatively, the proxy could query KC_{rev} for a specific proof of membership or non-membership at the time of a TLS handshake. Future research should examine the trade-offs between TLS handshake latency and overall bandwidth usage in these two approaches.

8 Related Works

PKI alternatives. Attempting to remove the complexities of the PKI and the need for certificates altogether, Shamir [30] proposed an identity-based encryption system in which a user's public key could be an arbitrary string, such as their e-mail address. Later, Boneh & Franklin [9] developed the first practical construction of IBE using bilinear groups. A fundamental challenge in IBE is that a trusted key authority creates all private keys, effectively acting as a key escrow. To mitigate this, Boneh and Franklin [9] originally suggested a key threshold scheme to distribute key generation across multiple authorities. Kate and Goldberg [21] later implemented and evaluated this approach, though the design necessarily adds communication and infrastructure costs over standard IBE.

Alternative approaches, such as self-certified public keys [17, 28] and certificateless public-key cryptography [1], occupy a middle ground between traditional PKI and IBE, but still place trust assumptions on a key authority regarding forgery. Mazu similarly occupies an intermediate design space, but unlike these prior systems, counters authority misbehavior by requiring the KC to generate publicly-verifiable, cryptographic proofs of membership. Additionally, Mazu is fundamentally an authentication system, rather than an encryption system.

Serverless with trusted execution environments. Several research efforts [2, 11, 34, 35] aim to protect microservice applications from untrusted cloud providers or cloud-based attackers by running services within trusted execution environments, particularly Intel SGX enclaves [26]. In addition to hosting serverless workloads in enclaves, these systems offer protected control-plane services such as key distribution enclaves, software TPMs, and resource accounting. Later research [22, 24, 37] focuses on optimizing these initial designs for better performance, or develops mutual attestation schemes that do not rely on a trusted (enclaved) authority, though with some constraints on the application design [12, 36]. Mazu is orthogonal to these efforts, and their composition represents an interesting design in which TEEs provide confidentiality and integrity guarantees for the services, and an untrusted control plane facilitates mutual attestation.

Hardening microservices. Finally, many efforts focus on hardening serverless runtimes. Sun et al. [31], for example, enhance the Linux kernel with *security namespaces*, enabling containers to define and enforce security policies independently. Google's gVisor [33] application kernel provides a userspace alternative to mediate application interactions, reducing the risk of container escapes. In turn, BASTION [27] improves network security by limiting container visibility through per-container network stacks with fine-grained security policies. Other research [13, 20, 25] focuses on enforcing information flow control between services. Mazu also provides defense-in-depth for microservices, but specifically protects the control plane rather than the data plane.

9 Conclusion

Service meshes are essential for providing zero trust networking to microservices, but the service mesh's local CA represents a security weak point. In this paper, we presented our initial work on Mazu, a service mesh extension that replaces the CA with a secure, unprivileged public key registry using registration-based encryption. Our preliminary implementation and evaluation show Mazu's potential, and future work will address even stronger threats.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CNS-2348130.

References

[1] Sattam S. Al-Riyami and Kenneth G. Paterson. 2003. Certificateless Public Key Cryptography.
[2] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *ACM Workshop on Cloud Computing Security (CCSW)*.
[3] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. [n. d.]. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.
[4] Consul Authors. 2025. Consul by HashiCorp. https://www.consul.io/
[5] Envoy Proxy Authors. 2025. Envoy proxy - Home. https://www.envoyproxy.io/
[6] Istio Authors. 2025. The Istio service mesh. https://istio.io/latest/about/service-mesh/
[7] Kubernetes Authors. 2025. Production-Grade Container Orchestration. https://kubernetes.io/
[8] Linkerd Authors. 2025. The world's most advanced service mesh. https://linkerd.io/
[9] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *International Cryptology Conference (CRYPTO)*.
[10] Craig Box (Google). 2020. Introducing istiod: simplifying the control plane. https://istio.io/latest/blog/2020/istiod/
[11] Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In *ACM International Systems and Storage Conference (SYSTOR)*.
[12] Guoxing Chen and Yinqian Zhang. 2022. MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties. In *USENIX Security Symposium*.
[13] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *The Web Conference (WWW)*.
[14] Armando Faz-Hernandez and Kris Kwiatkowski. 2019. *Introducing CIRCL: An Advanced Cryptographic Library.* Cloudflare. Available at https://github.com/cloudflare/circl. v1.6.0 Accessed Jan, 2025.
[15] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. 2010. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Symposium on Operating Systems Design and Implementation (OSDI)*.
[16] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. 2018. Registration-Based Encryption: Removing Private-Key Generator from IBE. In *Theory of Cryptography Conference (TCC)*.
[17] Marc Girault. 1991. Self-Certified Public Keys. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.
[18] Laurent Girod and Wouter Lueks. 2022. petrilic is a Python wrapper around RELIC. https://github.com/spring-epfl/petrelic.
[19] Noemi Glaeser, Dimitris Kolonelos, Giulio Malavolta, and Ahmadreza Rahimi. 2023. Efficient Registration-Based Encryption. In *ACM Conference on Computer and Communications Security (CCS)*.
[20] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. 2023. Guarding Serverless Applications with Kalium. In *USENIX Security Symposium*.
[21] Aniket Kate and Ian Goldberg. 2010. Distributed Private-Key Generators for Identity-Based Cryptography. In *Security and Cryptography for Networks*.
[22] Seong-Joong Kim, Myoungsung You, Byung Joon Kim, and Seungwon Shin. 2023. Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves. In *ACM Symposium on Cloud Computing (SOCC)*.
[23] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*.
[24] Mingyu Li, Yubin Xia, and Haibo Chen. 2021. Confidential Serverless Made Efficient with Plug-In Enclaves. In *International Symposium on Computer Architecture (ISCA)*.
[25] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. 2021. Automatic Policy Generation for Inter-Service Access Control of Microservices. In *USENIX Security Symposium*.
[26] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.
[27] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *USENIX Annual Technical Conference (ATC)*.
[28] Holger Petersen and Patrick Horster. 1997. Self-certified Keys—Concepts and Applications. In *Conference on Communications and Multimedia Security (CMS)*.
[29] Ahmadreza Rahimi and Noemi Glaeser. [n. d.]. efficientRBE. https://github.com/ahmadrezarahimi/efficientRBE.
[30] Adi Shamir. 1984. Identity-based Cryptosystems and Signature Schemes. In *International Cryptology Conference (CRYPTO)*.
[31] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security Symposium*.
[32] Lei Tang (Google). 2020. Remove cross-pod unix domain sockets. https://istio.io/v1.12/blog/2020/istio-agent/
[33] The gVisor Authors. [n. d.]. gVisor. https://gvisor.dev.
[34] Dave (Jing) Tian, Joseph I. Choi, Grant Hernandez, Patrick Traynor, and Kevin R. B. Butler. 2019. A Practical Intel SGX Setting for Linux Containers in the Cloud. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.
[35] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. 2019. Clemmys: Towards Secure Remote Execution in FaaS. In *ACM International Systems and Storage Conference (SYSTOR)*.
[36] Furkan Turan and Ingrid Verbauwhede. 2019. Propagating Trusted Execution through Mutual Attestation. In *Workshop on System Software for Trusted Execution (SysTEX)*.
[37] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable Enclaves for Confidential Serverless Computing. In *USENIX Security Symposium*.