Calypso: Fine-Grained Access Control for Zero-Trust Cloud Service Discovery

Pankaj Niroula
pniroula@wm.edu
William & Mary
Williamsburg, Virginia, United States

Peyton Boggs
prboggs@wm.edu
William & Mary
Williamsburg, Virginia, United States

Aashutosh Poudel
apoudel01@wm.edu
William & Mary
Williamsburg, Virginia, United States

Stephen Herwig
smherwig@wm.edu
William & Mary
Williamsburg, Virginia, United States

Abstract

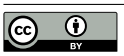
Modern clouds have embraced zero-trust at the application layer via service meshes, yet the cluster-wide discovery plane—DNS servers and their key-value backends—remains implicitly trusted. As a result, service names and addresses are exposed to any workload, enabling unauthorized enumeration and undetectable tampering if the directory is compromised. We present Calypso, a DNS-compatible service-discovery system that extends zero-trust data-plane protections to cloud-based service discovery by hiding both record contents and service names. Calypso treats the directory as an untrusted conduit: it stores only ciphertexts and RSA-based salted hashes, and performs equality matching over obfuscated names. Our design composes hierarchical identity-based encryption with an obfuscated-name matching scheme so that *(i)* only authorized clients learn record plaintexts and the existence of matching names, and *(ii)* unauthorized parties—including a fully compromised directory—cannot enumerate names or undetectably modify records. We implement Calypso in CoreDNS with an etcd backend, preserving existing operational workflows and client APIs. Our prototype demonstrates that Calypso is deployable with modest lookup-latency overheads and negligible server resource impact, while providing authenticity, end-to-end confidentiality, tamper evidence, and name hiding not available in today's discovery stacks.

CCS Concepts

• **Networks** → **Naming and addressing**; • **Security and privacy** → **Access control**; **Key management**; *Distributed systems security*.

Keywords

Domain Name System; Service Discovery; Access Control; Hierarchical Identity-Based Encryption; Cloud Security



SACMAT '26, Waterloo, ON, Canada

ACM ISBN 979-8-4007-2107-6/2026/07
https://doi.org/10.1145/3750555.3811897

ACM Reference Format:
Pankaj Niroula, Peyton Boggs, Aashutosh Poudel, and Stephen Herwig. 2026. Calypso: Fine-Grained Access Control for Zero-Trust Cloud Service Discovery. In *Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT '26), July 8–10, 2026, Waterloo, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3750555.3811897

1 Introduction

Zero trust [53] embodies the principle of "never trust, always verify." It rejects the assumption that any component within a network boundary is inherently trustworthy: every request must be authenticated and authorized, access must follow least privilege, data must remain encrypted in transit and at rest, and all activity must be monitored. Cloud platforms have largely adopted this model at the application layer—often through service meshes such as Istio [41] and Linkerd [50]—which provide transparent mutual authentication and fine-grained access controls between microservices.

However, zero-trust principles have not been extended to the cluster-wide metadata plane that underpins cloud and microservice systems. Core infrastructure services—such as key-value stores and service directories—remain implicitly trusted. For instance, a cluster's local DNS service typically allows any workload to query and resolve any other service, exposing the entire topology to all tenants. While mechanisms such as DNS-SD and etcd-based directories are essential for dynamic service discovery, they introduce two systemic vulnerabilities: *(i)* adversaries can enumerate services to map cluster structure, identify targets, and expand the attack surface; and *(ii)* compromise or misconfiguration of the directory can enable tampering and traffic redirection.

These weaknesses are not merely theoretical. The etcd key-value store—the default in Kubernetes—has repeatedly suffered from vulnerabilities [18, 19, 21, 22, 26, 27], including improper client TLS validation that allowed attackers to exfiltrate the entire key-value store. Similarly, CoreDNS [6]—the default cluster DNS service—has been vulnerable to cache-poisoning [25] and redirection attacks [24], while common misconfigurations [20, 23] have enabled privilege escalation into etcd. Collectively, these issues highlight that service discovery in contemporary cloud environments remains outside the zero-trust boundary.

In this paper, we ask the following research question:

Can service discovery be zero-trust, so that it is enumeration-resistant and tamper-evident—even if the directory is compromised—while preserving efficient lookups?

Our approach. We present Calypso, a DNS-compatible service-discovery system that protects both *record contents* and *service names*. Calypso treats the directory as an untrusted conduit: it stores and matches only ciphertexts and salted hashes. Concretely, Calypso composes hierarchical identity-based encryption (HIBE) with obfuscated name matching so that *(1)* only authorized readers learn record plaintexts *and* the existence of matching names, and *(2)* unauthorized parties—even a fully compromised directory—cannot enumerate names or undetectably alter records. Calypso integrates with CoreDNS and an etcd backend with only minor changes to the DNS naming model and client API.

Why this is hard. As we elaborate in §2, prior work secures messages end-to-end but leave identifying metadata (e.g., service names) visible for efficient lookup and filtering operations. As a result, enumeration remains possible even when the payloads are encrypted. Calypso instead encrypts *both* the values and securely obfuscates their lookup names, enabling fast, private equality tests at the directory while keeping keys at the endpoints.

Contributions. We make the following contributions:

- **Name-hiding service discovery.** We introduce Calypso, an extension to DNS-based service discovery in the cloud that conceals both service records and service names, thus preventing enumeration and tampering, and extending zero-trust principles to service discovery itself. Calypso combines prior work in hierarchical identity-based encryption with a novel design for obfuscated search.
- **Practical integration.** We implement a prototype of Calypso in the default Kubernetes DNS server of CoreDNS, with an etcd backend that preserves existing operational workflows, such as registration and updates.
- **Evaluation.** In our performance evaluation, Calypso shows only modest overhead in client latency and negligible effects on server-resource usage.

2 Background and Motivation

2.1 Service Discovery

Service discovery enables clients to locate available service instances and resolve each instance to its network address.

DNS-based approaches. DNS supports several service discovery conventions. When replicas share the same port and priority, they can use a common domain name: the client issues an `A` query to obtain all IP addresses and selects one (e.g., via round-robin). When replicas differ in hostname, port, or priority, deployments follow RFC 2782 [33]: the client sends an `SRV` query for the service name (e.g., `_postgresql._tcp.default.svc.cluster.local`) to obtain replicas and their parameters, selects one based on priority, and resolves its hostname with an `A` query. RFC 6763 ("DNS-Based Service Discovery") [15] defines a more flexible but less common cloud model in which a client first issues a `PTR` query to enumerate service instances, then resolves each using `SRV`, `A`, and `TXT` records. Unlike replica-based discovery, this approach treats instances as distinct and is common in zero-configuration environments such as printer discovery.

Cloud DNS stack. In a Kubernetes cluster, a local authoritative name server—typically CoreDNS, a graduated CNCF project and Kubernetes' default DNS service—handles name resolution for in-cluster services. Kubernetes stores DNS records in the etcd key-value store (another CNCF-graduated project), in JSON format. CoreDNS usually subscribes to the Kubernetes API server to receive service registration updates, caching the results locally. Optionally, it can use etcd directly as its DNS record backend.

The etcd service organizes its keys in a flat namespace where slashes act as pseudo-directory separators. For DNS records, the key corresponds to the domain name in reverse order, with slashes replacing dots (e.g., `www.example.com` → `/com/example/www`). When a domain has multiple values for a record type (e.g., multiple `A` records), CoreDNS stores each value under distinct subkeys such as `/com/example/www/1` and `/com/example/www/2`. The JSON value, not the key name, specifies the DNS record type and may include multiple types (e.g., `A` and `AAAA`).

2.2 How Fast Is Brute-Force Enumeration?

DNS-based service enumeration is a well-known attack in Kubernetes. A 2019 CNCF-funded Trail of Bits assessment [58] showed that a pod can enumerate cluster services via DNS alone, without Kubernetes API access. Commercial platforms such as Datadog include dedicated detection rules for this technique, mapping it to MITRE ATT&CK Network Service Discovery [28].

Consider an attacker who exploits a vulnerability in a web-facing pod. Although Kubernetes RBAC restricts API access, the pod can freely query CoreDNS. By issuing queries for common service names across namespaces (e.g., `db.prod.svc.cluster.local`, `redis.cache.svc.cluster.local`), the attacker maps the cluster's internal topology—identifying databases, caches, and authentication services—without privileged access. This reconnaissance enables targeted lateral movement toward high-value services.

To quantify the feasibility of such enumeration, we estimate how long a brute-force attack would take. We assume $\mathcal{A}$ knows a corpus of N possible domain names (as based on prior cloud configurations), of which the cluster actually uses $m < N$. For simplicity, we assume the domain names are uniformly random. We seek to determine how many DNS queries $\mathcal{A}$ must issue to *(i)* discover the first existing name, *(ii)* find all m names, and *(iii)* reach intermediate milestones, such as discovering half of them ($m/2$).

This problem is represented by the *negative hypergeometric* distribution. Table 7 in Appendix A illustrates the expected number of queries and corresponding total time to brute-force a cluster with $m = 1{,}000$ domain names drawn from a corpus of $N = 1{,}000{,}000$ possible names, where each query takes 0.90 ms (measured in our evaluation; see §8). Under these parameters, $\mathcal{A}$ could find the first valid name in under 2 seconds, enumerate 10% of the names in about 90 seconds, and 50% in roughly 7.5 minutes. Although these times are already short, they represent an upper bound—in practice, an attacker could reduce them substantially through parallel queries or by exploiting dependencies among names.

2.3 Off-the-shelf Defenses

Existing access controls. Existing infrastructure services such as etcd and CoreDNS lack mechanisms for enforcing fine-grained, identity-aware access control. etcd offers only coarse-grained role-based permissions—read, write, or read/write over keys or key ranges in a flat namespace. In Kubernetes deployments, these controls are typically bypassed altogether: the API server mediates all access to etcd and enforces RBAC solely at the API layer. As a result, although workloads cannot directly access etcd, authorization remains coarse, applying only to high-level resource abstractions rather than individual data objects.

CoreDNS exhibits similar limitations. Its `acl` plugin provides only IP-based filtering, allowing or denying entire subnets without regard to client identity or queried resource. The `kubernetes` plugin can verify that queries originate from valid pods by checking source IPs, but this mechanism relies on network-layer trust rather than cryptographic authentication. Although Kubernetes network policies and service accounts can further constrain which pods or namespaces may issue DNS queries, these controls remain coarse and largely orthogonal to the semantics of service discovery.

External policy engines. CoreDNS also supports integration with external policy engines [17] such as Open Policy Agent (OPA) [16] and Infoblox's Themis [39]. These engines evaluate each DNS query against fine-grained authorization rules, enabling policies far more expressive than CoreDNS's built-in controls. However, since the policy engine must observe plaintext queries to make authorization decisions, it necessarily learns which names each client requests. If the engine or DNS server is compromised, the attacker gains full visibility into query patterns and can enumerate all services.

Extensions for verifiable credentials. An alternative approach is for the cluster administrator to provision each workload with a *verifiable credential*, such as a JSON Web Token (JWT) [42]—attesting to the workload's identity—which the workload attaches to its DNS request. The token may also specify which record names the service is authorized to publish or query; alternatively, the DNS server can store these authorization rules itself. This mechanism limits the impact of a compromised workload, preventing it from enumerating unauthorized domains. However, it does not protect against compromise of the DNS server or its key-value backend, since an attacker with such access can read and modify all DNS records.

Trusted execution environments. One way to secure the DNS server is to run it inside a *trusted execution environment (TEE)*, such as an Intel SGX enclave [40, 51], potentially combined with verifiable credentials. SGX enables user-space applications to execute in isolation from untrusted system software (e.g., the OS or hypervisor) and to resist certain physical attacks. An enclaved DNS server that uses a key-value backend could also encrypt records before storing them in a non-enclaved service, extending protection to persisted data. However, TEEs are not a complete solution: side-channel attacks [13, 14, 47–49, 59, 60] can undermine their guarantees, and they do not prevent vulnerabilities in the application code itself.

2.4 Prior Solutions

Prior work on private service discovery (summarized in Table 1) primarily targets confidentiality and authenticity in broadcast-oriented settings such as IoT and mobile computing, where privacy leaks have been well documented [7, 34, 38]. At a high level, these efforts balance two competing goals: *(i)* hiding resource names to prevent enumeration attacks and *(ii)* supporting flexible access policies, where a single key can authorize access to multiple resources.

Early privacy-preserving service discovery systems [43] require static, out-of-band key exchanges between each client and service instance. While these schemes hide resource names (e.g., by replacing them with key-derived hashes), they do not scale due to the overhead of managing per-pair keys.

Later works investigate constructions based on *identity-based encryption (IBE)* [55], a public-key cryptosystem that removes the need for a certificate authority by deriving a user's public key from a meaningful string identifier, such as their email or URL. Wu et al. [61] ensure that only authorized clients can discover services using prefix encryption [46]—a generalization of IBE in which a key for identity id can decrypt ciphertexts encrypted under any identity with prefix id. However, their design exposes the service's authorization policy (the prefix under which the record is encrypted), leaving it susceptible to enumeration attacks. Tryst [52] takes a similar approach but uses *anonymous identity-based encryption (AIBE)* [1], which hides the identity under which a service advertises its announcement, at the cost of less flexible authorization policies. JEDI [45] instead uses a flexible, anonymous *hierarchical identity-based encryption (HIBE)* cryptographic scheme [2, 10] for end-to-end authenticated and encrypted messages in publish-subscribe systems, but does not conceal the metadata (e.g., the URLs on which messages are published). Calypso builds upon JEDI and extends it for both the DNS setting and to conceal the DNS query names (the analog to PubSub URLs); we review JEDI further in §4.

Like Wu et al. [61], PriSrv [62] handles the dual problem of private service discovery and mutual private authentication. PriSrv uses anonymous credentials with *matchmaking encryption* [4, 5] (a generalization of IBE), which allows not only the service to specify an authorization policy on the client, but also the client on the service. Nevertheless, to enable clients to filter unauthorized services without heavy computation, the policies (which include the domain names) are public. PriSrv+ [63] extends PriSrv with more expressive policies by supporting an unbounded attribute universe and optimizing its cryptographic operations. However, it still publicly exposes the attribute names in policies.

3 Goals, Assumptions, and Threat Model

System model. We consider a cluster deployment that coordinates multiple application services ("applications") in a microservice setting. We focus on two control-plane components: *(i)* a *DNS service* used by applications for service discovery and name resolution, and *(ii)* a backend *key-value store (KVS)* that the DNS service uses to store DNS records. Applications run on an orchestrator (e.g., Kubernetes) that includes a trusted *authority* to provision cryptographic keys for access control and record protection. As this authority resides in the cluster control plane—which is already trusted for workload scheduling and secrets management—Calypso does not

Table 1: Comparison of service discovery protocols.

	Approach	Cite	Privacy			Flexible Policies	Design
			Authn.	Data	Metadata		
Off-the-shelf	DNS-SD	[15]	✗	✗	✗	✗	Default DNS mechanism for service discovery
	Policy Engines	[17]	✗	✗	✗	✓	External engine (e.g., OPA, Themis) evaluates queries; requires trusted DNS server
	JWT Access Control		✓	✗	✗	✓	Client sends JWT with each request; server validates it
	Intel SGX Enclave		✓	✗	✗	✓	DNS server runs in an Intel SGX enclave; client also sends JWT
Academic	Tryst	[52]	✓	✓	✓	✗	Anonymous identity-based encryption (AIBE)
	KW14	[43]	✗	✓	✓	✗	Symmetric-key pairing between client and service
	WTSB16	[61]	✓	✓	✗	✓	Prefix encryption
	JEDI	[45]	✓	✓	✗	✓	Hierarchical identity-based encryption (HIBE)
	PriSrv	[62]	✓	✓	✗	✓	Anonymous credential-based matchmaking encryption (ACME)
	PriSrv+	[63]	✓	✓	✗	✓	Fast and expressive matchmaking encryption (FEME)
	Calypso		✓	✓	✓	✓	HIBE with concealed names

enlarge the existing trust boundary. Key provisioning follows static namespace-level policies; we leave contextual or dynamic trust evaluation (e.g., adapting key release to the security posture of a DNS server or domain) to future work.

Security goals. Calypso protects DNS service discovery even if the DNS service or KVS is compromised. Specifically, it guarantees:

S1 *End-to-end confidentiality and integrity:* Only endpoints holding the appropriate keys can read or publish valid DNS records; the DNS service and KVS handle only ciphertexts. An adversary lacking the decryption/signing keys cannot learn plaintexts or undetectably modify records.

S2 *Enumeration resistance:* An adversary can learn (enumerate) domain names only for records it is authorized to access (e.g., for which it holds the necessary keys); the existence of other names remains hidden.

Adversary model. The adversary may fully compromise and control the DNS service and the KVS (observe, insert, modify, delete). The adversary may also compromise *some* application instances and exfiltrate their keys. Continuous compromise of *every* application instance is out of scope. In this work we specifically focus on attackers that compromise an endpoint (an application, the DNS service, or the KVS); standard encrypted-DNS techniques, such as DNS-over-TLS [37] or DNS-over-HTTPS [35], ensure DNS privacy against network-level eavesdropper.

Out of scope. We do not address: *(i)* freshness or rollback protection against a malicious key-value store; *(ii)* leakage from access patterns or traffic analysis; *(iii)* immediate key revocation, which would require a trusted broadcast channel or broadcast encryption [30]; *(iv)* fine-grained accountability (e.g., attributing a published record to a specific application); or *(v)* availability and DoS attacks, since an adversary controlling the DNS server or KVS can trivially deny service. Calypso addresses trust minimization for the DNS data plane but does not implement the dynamic policy evaluation and continuous verification prescribed by the full NIST ZTA framework [53]; these remain future work.

4 Preliminaries

Notation.

- $\{0,1\}^n$ is the set of bit strings of length n.
- $a \parallel b$ denotes concatenation for strings and byte arrays.
- $[n]$ is the set of integers $\{1,\ldots,n\}$.
- $x \xleftarrow{\$} S$ denotes sampling a uniformly random element $x \in S$. $x \xleftarrow{\$} \mathsf{A}(\cdot)$ denotes that x is the return value of a randomized algorithm A.

Bilinear maps. Since the HIBE schemes that we review (and that underlie Calypso) use bilinear maps, we briefly recall this concept.

Let G_1, G_2, and G_T be cyclic groups of prime order p where g_1 is a generator for G_1 and g_2 is a generator for G_2. A bilinear map is a map $e : G_1 \times G_2 \to G_T$ with the following properties:

(1) **Bilinearity:** For all $u \in G_1, v \in G_2$ and $a, b \in \mathbb{Z}_p^*$, $e(u^a, v^b) = e(u,v)^{ab}$.
(2) **Non-degeneracy:** $e(g_1, g_2) \neq 1$
(3) **Computability:** The map e is efficiently computable.

4.1 WKD-IBE

WKD-IBE [2] is a HIBE variant; like many HIBE schemes, WKD-IBE uses bilinear maps but offers greater flexibility in key derivation. In standard HIBE [31, 36], user identities are represented as lists of strings: the root authority issues private keys to first-level users, and a user at level i can derive keys for its descendants at level $i+1$, enabling hierarchical delegation. WKD-IBE generalizes this model by allowing delegation based on pattern matching rather than strict level-by-level hierarchy, supporting more expressive and flexible key relationships. For instance, a motivating use case for WKD-IBE is to derive secret keys for all departmental sysadmin email addresses `sysadmin@*.univ.edu`, where `*` is a wildcard that can be replaced with any string.

Patterns. In WKD-IBE, messages are encrypted with *patterns*, and keys correspond to patterns. A pattern consists of a list of slots: $P = (\mathbb{Z}_p^* \cup \{\perp\})^\ell$, where $\perp$ denotes that the slot lacks a value. If

$P(i)$ denotes the i^{th} slot of P (1-indexed), then we say a pattern P_1 *matches* P_2 (written $P_1 \sim P_2$) if, for all $i \in [\ell]$, either $P_1(i) = \perp$ or $P_1(i) = P_2(i)$. In other words, if P_1 specifies a value for slot i, then P_2 must have the same value at i.

Encryption. A key for pattern P_1 can decrypt a message encrypted with pattern P_2 if $P_1 = P_2$. Furthermore, a key for pattern P_1 can derive a key for pattern P_2 if and only if $P_1 \sim P_2$.

The algorithms that comprise WKD-IBE are:

WKD.Setup$(1^\lambda, \ell) \xrightarrow{\$}$ pp, msk: Given the security parameter 1^λ and the number of slots ℓ in a pattern, return the public parameters pp and the master secret key msk. We assume that pp is an implicit parameter to the remaining algorithms.

WKD.KeyGen$(msk, P) \xrightarrow{\$} sk_P$: Given the master secret key msk and a pattern P, return a secret key that can decrypt ciphertexts encrypted to that pattern.

WKD.KeyDer$(sk_{P_1}, P_2) \xrightarrow{\$} sk_{P_2}$: Given a secret key sk_{P_1} for pattern P_1, and a pattern P_2 such that $P_1 \sim P_2$, derive a secret key sk_{P_2} that can decrypt ciphertexts encrypted to P_2.

WKD.Encrypt$(P, m) \xrightarrow{\$} ct_P$: Given a pattern P and message $m \in \mathbb{G}_T$, compute the ciphertext ct_P.

WKD.Decrypt$(sk_P, ct_P) \to m$: Given a secret key sk_P for pattern P and a ciphertext ct_P that is encrypted to pattern P, decrypt the ciphertext and return the message m.

Signatures. As Abdalla et al. [2] note, WKD-IBE admits a signature scheme in the same fashion as IBE [12] and HIBE [31]. Namely, to sign a message $m \in \mathbb{Z}_p^*$ with a key sk_P for pattern P, the signer uses WKD.KeyDer to fill a dedicated "signature" slot with m, and then presents the decryption key as the signature. The verifier checks the signature by encrypting a random message under the message-augmented pattern and then attempting to decrypt it using the key acting as a signature.

The algorithms that comprise WKD-IBE signatures are:

WKD.Sign$(sk_P, m) \xrightarrow{\$} \sigma_{m,P}$: Given a secret key sk_P for pattern P and a message $m \in \mathbb{Z}_p^*$, compute a signature $\sigma_{m,p}$ over m.

WKD.Verify$(P', \sigma_{m,p}, m') \to$ {true, false}: Given a pattern P', a signature $\sigma_{m,p}$, and a message $m' \in \mathbb{Z}_p^*$, return true if $P = P' \wedge m = m'$; otherwise, return false.

4.2 JEDI

JEDI (Joining Encryption and Delegation for IoT) [45] builds on and extends WKD-IBE to provide an end-to-end encryption protocol for the many-to-many, decoupled communication model common in IoT systems, such as publish-subscribe architectures in smart buildings. In these systems, senders publish messages about resources, while receivers subscribe to those resources through an intermediary router.

JEDI extends WKD-IBE in two key ways: *(i)* it introduces a more efficient and provably anonymous signature verification algorithm; and *(ii)* it supports encryption to patterns containing $\perp$ slots, allowing partial pattern specification. (We elided this detail when presenting WKD.Encrypt in §4.1, as Calypso uses JEDI's more flexible version of this algorithm.) Conceptually, JEDI models resources hierarchically as URLs—publishers encrypt to a resource's URL rather than to specific recipients, and only authorized subscribers possess the corresponding decryption keys. Through hierarchical delegation, principals can derive and delegate keys restricted to subtrees of the resource hierarchy, enabling scalable, fine-grained access control. While JEDI provides strong confidentiality, authenticity, and delegation, it leaves resource URLs visible, a limitation that motivates Calypso's design.

5 Design

Consider three approaches to resolving d = postgres.prod.svc.cluster.local. In *standard DNS*, the client queries d directly; CoreDNS looks up /local/.../postgres in etcd and returns the plaintext record. An attacker who compromises the directory observes both names and records. In *JEDI* (§4.2), records are encrypted end-to-end, but the domain name—the etcd lookup key—remains in plaintext: /local/.../postgres ↦ ciphertext. An attacker cannot read records but can still enumerate names, learning that a `postgres` service exists.

Calypso inherits JEDI's encryption, signatures, and hierarchical delegation, but conceals *both* the record and the domain name. Let $\bar{d}$ denote the *obfuscated* form of d. An authorized publisher computes $\bar{d}$ from secret cryptographic salts issued by the authority, encrypts the record, and stores $\bar{d} \mapsto$ ciphertext. A querier with matching authorization independently derives the same $\bar{d}$ and issues a direct key lookup—preserving $O(1)$ performance. Without the salts, an attacker sees only opaque $\bar{d}$ values and ciphertexts. The core design challenge (S2, §3) is to ensure that $\bar{d}$ is both *stable* (all authorized parties derive the same $\bar{d}$) and *unguessable* (infeasible to derive without authorization).

Building blocks. Let $\mathbb{P}$ be the set of prime numbers, and define $\mathbb{P}_{<p} = \{q \in \mathbb{P} : 2 < q < p\}$ as the set of odd primes less than p. We assume the following collision-resistant hash functions:

- $H_p : \{0,1\}^* \to \mathbb{P}_{<p}$ maps a bit string to an odd prime number.
- $H_{\mathbb{Z}} : \{0,1\}^* \to \mathbb{Z}_p^*$ maps a bit string to a scalar.
- $H : \{0,1\}^* \to \{0,1\}^\ell$ maps a bit string to an output of length ℓ.

We also assume the following *key derivation function (KDF)*:

- $\mathsf{KDF} : \mathbb{G}_T \to \{0,1\}^\ell$ takes an element of the target group of a bilinear map and outputs a bit string of length ℓ.

Finally, we assume a symmetric encryption scheme (e.g., AES) with algorithms SE.KeyGen, SE.Encrypt, and SE.Decrypt.

5.1 Encoding Domain Names to Patterns

Inspired by JEDI, Calypso derives an encryption pattern from a domain's label hierarchy. For example, for d = `b.a.com`, Calypso reverses the labels to `com.a.b` and appends a terminator symbol `"$"`, forming `("com", "a", "b", "$")`. As in JEDI, Calypso computes the corresponding WKD-IBE pattern as $P(1) = H_{\mathbb{Z}}(\texttt{"com"})$, $P(2) = H_{\mathbb{Z}}(\texttt{"a"})$, $P(3) = H_{\mathbb{Z}}(\texttt{"b"})$, and $P(4) = H_{\mathbb{Z}}(\texttt{"\$"})$.

5.2 RSA-based Accumulator Salt Extension

A naïve approach is to hash the pattern directly: $\bar{d} \leftarrow H(P)$. However, the pattern is deterministic—anyone who knows the domain name can compute P—so an attacker can still perform the brute-force search from §2.2 to test candidate names without holding a decryption key.

Table 2: Example of a writing key with a wildcard deriving a new key.

		1	2	3	4	5	6
Domain Name	`www.*.com`						
	Reverse Labels	`"com"`	`"*"`	`"www"`	`"$"`		
	Slots	$H_{\mathbb{Z}}(\texttt{"com"})$	$H_{\mathbb{Z}}(\texttt{"*"})$	$H_{\mathbb{Z}}(\texttt{"www"})$	$H_{\mathbb{Z}}(\texttt{"\$"})$	$\perp$	$\perp$
	Salts	A_1	A_2	A_3	A_4		
Derived Name	`www.a.com`						
	Slots	$H_{\mathbb{Z}}(\texttt{"com"})$	$H_{\mathbb{Z}}(\texttt{"a"})$	$H_{\mathbb{Z}}(\texttt{"www"})$	$H_{\mathbb{Z}}(\texttt{"\$"})$	$\perp$	$\perp$
	Salts	A_1	$A_2' \leftarrow A_2^{H_p(2\|\|\texttt{"a"})} \bmod N_2$	$A_3' \leftarrow A_3^{H_p(2\|\|\texttt{"a"})} \bmod N_3$	$A_4' \leftarrow A_4^{H_p(2\|\|\texttt{"a"})} \bmod N_4$		
	Obfuscated Name			$H(A_1 \| A_2' \| A_3' \| A_4')$			

Instead, Calypso extends WKD.Setup and WKD.KeyGen to generate a cryptographic accumulator for each slot. A *cryptographic accumulator* [8, 9] is a compact, fixed-length value that represents a set of elements while supporting efficient membership proofs. Calypso uses RSA-based accumulators—not for the purpose of proving membership—but rather to generate unforgeable and unguessable values (which we term *salts*) that bind each domain label to the domain prefix. Specifically, we amend the authority as follows:

Setup. In addition to generating the master secret key msk, the authority performs the following for each pattern slot i:
(1) Generates an RSA modulus $N_i = p_i q_i$
(2) Samples a random base element $b_i \in \mathbb{Z}_{N_i}^*$
(3) Initializes an accumulator $A_i = b_i$

Key Generation. For a reversed domain name $d = (d_1, \ldots, d_k)$, the authority adds to each slot's accumulator the index-label pairs up to and including that slot:

$$A_i = b_i^{\prod_{t=1}^{i} H_p(t \| d_t)} \bmod N_i$$

Each A_i compactly represents the prefix set $\{d_1, \ldots, d_i\}$. Including the label index in the hash prevents label swapping. The authority issues the salts $\{A_i\}$ and moduli $\{N_i\}$ to the application along with its private key, but only for slots corresponding to actual DNS labels (including wildcards), not empty positions. The random base elements b_i remain secret.

Under the Strong RSA assumption, the accumulator provides collision-resistant encoding of sets in groups of unknown order, but requires that the accumulated values be primes (hence H_p) for correctness.

5.3 Issuing Keys

When the authority issues a key for a domain name d, it must compute the corresponding pattern P_d and return the private key sk_{P_d}. Calypso introduces additional challenges beyond JEDI—specifically, determining how to compute salts and patterns when domain names include wildcard labels (`"*"`) and how to distinguish between reading keys, which permit only record queries, and writing keys, which allow signing and record publication. Table 2 summarizes the relationship between key issuance, slots, and salts; we describe these mechanisms in detail below.

Computing slot values. For a given slot i, if all slots up to and including i contain non-wildcard labels, the authority can compute its final accumulator value A_i. If d_i is a wildcard `"*"`, the authority cannot compute the final salt for i or for any subsequent slot $j > i$. For example, for the reversed domain name `com.a.*.c`, the authority lacks sufficient information to compute the salts corresponding to `"*"`, `"c"`, and the terminator `"$"`.

When issuing a key, the authority constructs the pattern P using only the slots it can compute. It returns to the client the secret key sk_P for this pattern, along with the pairs (A_i, N_i) for all slots that correspond to a provided label, a wildcard, and the terminator label.

Reading vs. writing keys. As in JEDI, Calypso reserves the final slot for a signature. Unlike JEDI, however, Calypso uses this slot to distinguish between *writing keys*, which can generate signatures and thus publish DNS records, and *reading keys*, which cannot and are limited to querying and decrypting existing records. For writing keys, the authority treats the final slot as unfilled ($\perp$). For reading keys, it assigns the special label `"&"` and computes the slot value as usual ($H_{\mathbb{Z}}(\texttt{"\&"})$). As this final slot is used as a *mode* and not as part of the domain name, it does not have a corresponding salt.

5.4 Publishing and Querying Records

Let $\mathsf{Pattern}(sk_P) \rightarrow P$ be a function that, given a secret key sk_P, returns the pattern bound to that key. We also overload this function so that, on input a domain name, it returns the pattern for a reading key for that domain; the context should make the distinction clear.

To publish a DNS record for domain d_A, a party proceeds as follows:

(1) **Obtain a writing key.** The party must possess a writing key k for a pattern corresponding to a domain d such that $d \sim d_A$. If $d \neq d_A$, the party derives a signing key:

$$k_s \xleftarrow{\$} \mathsf{WKD.KeyDer}(k, d_A).$$

Otherwise, $k_s \leftarrow k$.

(2) **Derive the reader (decryption) pattern, P_e.** Using the signing key k_s, the party derives $P_e = \mathsf{Pattern}(d)$. Deriving P_e from $\mathsf{Pattern}(k_s)$ simply means converting the last slot from $\perp$ to $H_{\mathbb{Z}}(\texttt{"\&"})$.

(3) **Sign the record.** Given the DNS record r, the party computes a signature:

$$\sigma \xleftarrow{\$} \mathsf{WKD.Sign}(k_s, H_{\mathbb{Z}}(r))$$

(4) **Encrypt the record (KEM-DEM construction).** As a *key encapsulation mechanism (KEM)*, the party samples a random element $g \in \mathbb{G}_T$ and encrypts it using

$$c_g \xleftarrow{\$} \mathsf{WKD.Encrypt}(P_e, g).$$

As a *data encapsulation mechanism (DEM)*, the party derives a symmetric key (e.g., an AES key) as $k_{\mathsf{sym}} \leftarrow \mathsf{KDF}(g)$, samples a random initialization vector (IV), and encrypts the record

$$c_r \leftarrow \mathsf{SE.Encrypt}(k_{\mathsf{sym}}, \mathsf{iv}, r).$$

(5) **Output the published record.** The published authenticated ciphertext consists of the tuple $(c_g, \mathsf{iv}, c_r, \sigma)$.

Obfuscating names. To create the obfuscated name $\bar{d}$ for a domain d, the publisher computes a hash over all salts in the pattern associated with the encryption pattern P_e, up to and including the salt for the terminator label (`"$"`). Let j denote the index of the terminator label; the publisher computes:

$$\bar{d} \leftarrow H(A_1 \| \ldots \| A_j)$$

The publisher then stores the encrypted record under name $\bar{d}$.

Querying. To query a DNS record for domain d_A, a party proceeds as follows:

(1) **Obtain a reading key.** The party must possess either a writing or reading key k for a pattern corresponding to a domain d such that $d \sim d_A$. If $d \neq d_A$, the party derives a reading key:

$$k_d \xleftarrow{\$} \mathsf{WKD.KeyDer}(k, d_A)$$

Otherwise, $k_d \leftarrow k$. If k was a writing key, then k_d must be derived as a reading key by converting k's last slot from $\perp$ to $H_{\mathbb{Z}}(\texttt{"\&"})$.

(2) **Decrypt the record.** Decrypting the ciphertext tuple $(c_g, \mathsf{iv}, c_r, \sigma)$ simply involves unwrapping the KEM-DEM construction to compute the plaintext record r:

$$\begin{aligned} g &\leftarrow \mathsf{WKD.Decrypt}(k_d, c_g) \\ k_{\mathsf{sym}} &\leftarrow \mathsf{KDF}(g) \\ r &\leftarrow \mathsf{SE.Decrypt}(k_{\mathsf{sym}}, \mathsf{iv}, c_r) \end{aligned}$$

(3) **Verify the signature.** The querier verifies the record's signature σ by first creating the verification pattern. This entails setting the dedicated signing/reader slot for k_d back to the unfilled value $\perp$, as for a signer:

$$\begin{aligned} P_{\mathsf{ver}} &\leftarrow \mathsf{Pattern}(k_d) \\ P_{\mathsf{ver}}(\ell) &\leftarrow \perp \end{aligned}$$

The querier can then verify the signature on the hash of the plaintext record:

$$\mathsf{isValid} \leftarrow \mathsf{WKD.Verify}(P_{\mathsf{ver}}, \sigma, H_{\mathbb{Z}}(r))$$

Administration and debugging. Name obfuscation prevents administrators from browsing etcd to inspect records, since stored keys are opaque hashes. However, an administrator provisioned with a wildcard reader key (e.g., for `*.svc.cluster.local`) can derive the obfuscated name and decryption key for any service matching the wildcard, enabling targeted troubleshooting: given a known service name, the administrator computes $\bar{d}$, retrieves the record from etcd, and decrypts it. Bulk discovery of unknown services via directory browsing is not supported—this is an inherent consequence of enumeration resistance.

6 Security Analysis

We sketch why an unauthorized adversary cannot derive the obfuscated name $\bar{d}$ for a domain d, and thus cannot enumerate names.

Adversary model and security goals. We assume an adversary $\mathcal{A}$ that fully controls the directory plane (DNS/KVS), can observe/modify traffic, and may compromise a subset of endpoints (including transient exfiltration of their state). $\mathcal{A}$ does not obtain authorization for an uncompromised domain d and cannot compromise all endpoints continuously. Our subgoals of S2 in §3 are: S2.a (name hiding): the directory and unauthorized parties learn no information about domain names beyond what is revealed by their own authorizations; S2.b (authorization-bound derivation): only parties authorized for d can compute the obfuscated name $\bar{d}$, which is required to locate its record. We rely on the Strong RSA assumption, collision resistance of H_p, $H_{\mathbb{Z}}$, and H, and the standard security of HKDF and the symmetric cipher.

Intuition. Informally, $\bar{d}$ acts like a keyed hash of the domain labels, where the keys are the per-slot salts A_i. Without those salts, $\mathcal{A}$ can at best guess $\bar{d}$ at random, or attempt to reconstruct A_i values. However, reconstructing the exact A_i for an unauthorized prefix would allow $\mathcal{A}$ to "rewind" the accumulator without the authority, which we reduce to breaking Strong RSA.

Lemma 6.1 (Salt unforgeability under Strong RSA)**.** *Suppose that two domains d and d' differ at label i. Given the A_i accumulator for d, $\mathcal{A}$ cannot compute $A_i' = b_i^{\prod_{t=1}^{i} H_p(t \| d_t')} \bmod N_i$ with non-negligible probability.*

Sketch. Consider any such i, and suppose $\mathcal{A}$ outputs A_i'. We embed a Strong RSA challenge $y \in \mathbb{Z}_{N_i}^*$ into the base by choosing $b_i := y$ and programming the hash-to-prime oracle so that the unknown prefix product is the exponent $e > 1$ in the challenge instance. Any algorithm that recovers b_i (or that can transform valid accumulators across prefixes without the authority) yields an e-th root of y, violating Strong RSA. The index binding $H_p(t \| d_t)$ prevents exponent collisions across different label positions, so the reduction holds for each slot independently. □

Lemma 6.2 (Obfuscated name indistinguishability)**.** *If the salts $\{A_i\}$ for d are hidden, then $\bar{d} = H(A_1 \| \cdots \| A_j)$ is computationally indistinguishable from a random ℓ-bit string to any party lacking those salts.*

Sketch. Conditioned on Lemma 6.1 and the collision resistance of H, the concatenation $B = A_1 \| \cdots \| A_j$ is unpredictable to the adversary; hence $H(B)$ is pseudorandom in the random-oracle

heuristic or preimage-resistant in the standard model. Thus, without salts, random guessing is optimal, with success probability $2^{-\ell}$. □

Theorem 6.3 (Enumeration resistance)**.** *Let $\mathcal{A}$ be any probabilistic polynomial-time (PPT) adversary that does not hold authorization for d. The probability that $\mathcal{A}$ outputs d (and thereby detects the existence of d in the directory) with more than negligible advantage over random guessing is negligible under Strong RSA and the hash/KDF assumptions above.*

Sketch. Consider the following game. The challenger sets up the system, withholds all salts for d, and publishes the directory state (all ciphertexts and obfuscated names). $\mathcal{A}$ may adaptively corrupt some principals, query the directory, and interact with the KVS, but never obtains salts for d. If $\mathcal{A}$ outputs d, then either *(i)* it computed the hidden salts (contradicting Lemma 6.1/Strong RSA) or *(ii)* it found a preimage/collision for H on an unseen input (breaking collision/preimage resistance), or *(iii)* it guessed d at random (probability $2^{-\ell}$). Therefore $\Pr[\mathsf{win}] \le \mathsf{negl}(\lambda) + 2^{-\ell}$. □

Post-quantum considerations. Calypso's symmetric primitives (SHA-256, AES-256-CTR, HKDF) are quantum-resistant; however, the WKD-IBE layer (BLS12-381 pairings) and the RSA-based accumulator are not. Replacing them with lattice-based HIBE schemes [3] and lattice-based accumulators would preserve the design while providing post-quantum security. Evaluating the performance impact of such a transition remains future work.

7 Implementation

Core packages. We implement Calypso in Go by first developing a package that implements WKD-IBE based on JEDI's construction of this cryptosystem. Our implementation uses Cloudflare's Circl cryptographic library [29]—specifically, its BLS12-381 elliptic curve. Building on WKD-IBE, we then implement a `calypso` Go package that provides DNS-based semantics, reader/writer access controls, and the obfuscated-name generation algorithm.

Cipher suite selection. Calypso instantiates both H and $H_{\mathbb{Z}}$ with *SHA-256*. The hash function H_p also uses SHA-256 and follows the construction of Boneh et al. [11]. For key derivation, we use HKDF [44] parameterized with SHA-256. SE.Encrypt employs AES-CTR with a 256-bit key.

CoreDNS plugin. CoreDNS features a flexible plugin architecture, within which we implement the `calypso-dns` plugin. Calypso uses the DNS `OPT` record type to *(i)* indicate that a query is a Calypso request—causing the normal query name to be ignored—and *(ii)* carry the obfuscated name used by the DNS server to retrieve the encrypted record. The plugin recognizes this `OPT` record, queries the etcd backend for the corresponding value, and, if found, returns it to the client encapsulated within an `OPT` record in the response.

Support for domains with multiple records. In §2.1, we described DNS conventions for service discovery (e.g., `SRV` records) that return multiple DNS records in a single response. Although we presented Calypso's design in terms of individual records, it remains compatible with these conventions. As in standard CoreDNS behavior—where a lookup for a key in etcd may also return its subkeys—Calypso obfuscates only the domain name and encrypts each subkey's value under that name. The subkeys themselves may use arbitrary, non-semantic identifiers (e.g., random numbers).

Table 3: Comparison of the DNS request and response sizes for different access control approaches.

Approach	DNS Request		DNS Response	
	Size	Expansion	Size	Expansion
Normal	45B	baseline	88B	baseline
JWT	374B	+329B (8.3×)	99B	+11B (1.1×)
JEDI	60B	+15B (1.3×)	1733B	+1645B (19.7×)
Calypso	124B	+79B (2.8×)	1817B	+1729B (20.6×)

Measured for an `A` query: `a.default.svc.cluster.local`.

8 Evaluation

We conduct experiments on an Azure DC4s v2 virtual machine with 4 vCPUs and 16 GB of RAM, running Ubuntu 24.04.3 LTS. The system supports SGX, which we use in our end-to-end comparisons.

We use CoreDNS (1.12.4) and etcd (v3.7.0) as the backend KVS. For client latency measurements, we extend **q** [54], a Go-based DNS tool built on the popular miekg/dns package [32], to handle the different approaches and craft DNS requests with OPT records. For running CoreDNS with SGX support, we use the open-source **ego** framework [56] to build and run CoreDNS within an SGX enclave.

To populate the DNS with records, we implement a simple registration tool, **etcd-client**, that uses the etcdv3 API to register DNS records on behalf of services. Based on the approach, the tool performs the necessary cryptographic operations (e.g., encryption, signing) before storing the records in etcd. Services follow the `<service>.<namespace>.svc.cluster.local` Kubernetes naming convention. For simplicity, key provisioning and management are also done via this tool.

8.1 Microbenchmarks

Packet expansion. We measure packet sizes at the DNS message layer (packed DNS protocol format) using an instrumented **q** client, capturing overhead from EDNS options, encrypted payloads, and authentication tokens before transport encapsulation.

The results in Table 3 reveal two key insights. First, authentication approaches differ in where they place overhead. JWT shifts most overhead to requests (+329B token), while IBE-based schemes (JEDI and Calypso) shift it to responses (∼20× expansion from encryption), keeping requests lightweight (+15B for JEDI, +79B for Calypso via EDNS options). Second, Calypso incurs an additional +64B in requests (the SHA-256 obfuscated name in EDNS0) and +84B in responses (obfuscated name in encrypted message) compared to JEDI.

Scalability with domain depth. Figure 1 demonstrates how Calypso's cryptographic operations scale with domain name depth. Client-side operations exhibit favorable scaling: decryption remains constant at ∼3.1 ms regardless of label count, while verification

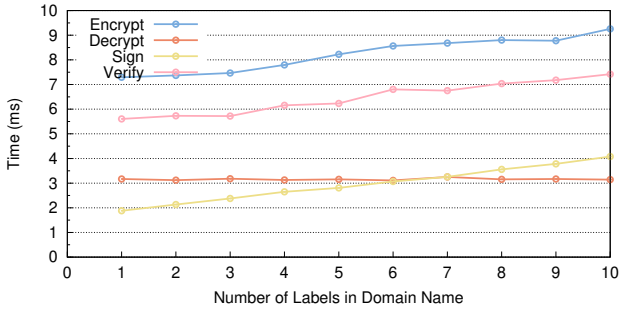


Figure 1: Latency of Calypso's cryptographic operations vs. number of DNS labels.

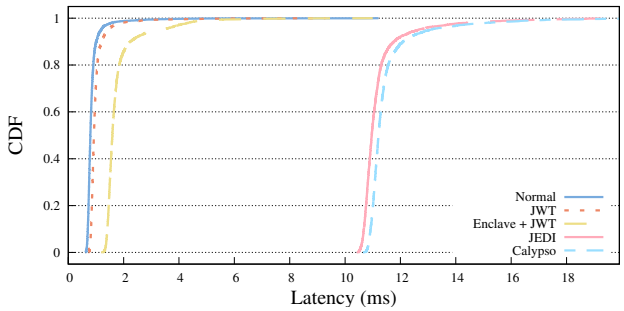


Figure 2: CDF of end-to-end DNS lookup latencies over UDP. Each curve represents the median distribution across 10 independent trials. Inter-trial variability is minimal.

Table 4: Comparison of Calypso's operation latencies (μs) to RSA (3072-bit modulus).

Operation	Calypso		RSA	Overhead
Labels	*1*	*10*		
IssueKey/KeyGen	10,074	72,355	289,003	0.035–0.25×
Encrypt	7,294	9,256	166	43.94–55.75×
Decrypt	3,169	†N/A	2,817	1.12×
Sign	1,879	4,078	2,952	0.64–1.38×
Verify	5,604	7,418	170	32.85–43.49×

† Calypso decryption is $O(1)$ with respect to the number of labels.

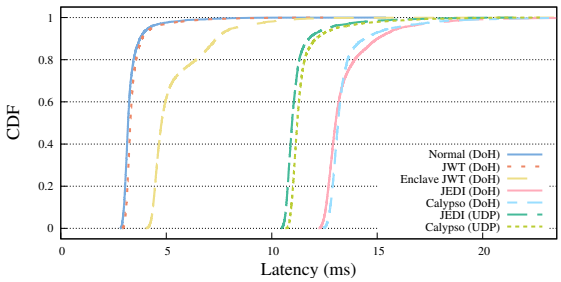


Figure 3: CDF of end-to-end DNS lookup latencies using DoH (cf. Figure 2 for UDP baseline). UDP versions of JEDI and Calypso included as reference baselines.

grows modestly from 5.6 ms to 7.4 ms (32% increase from 1 to 10 labels). Since these operations occur per-lookup, their limited growth is critical for scalability. Publisher-side operations show greater sensitivity—encryption increases 27% (7.3 ms→9.3 ms) and signing more than doubles (1.9 ms→4.1 ms) across the same range—but these are one-time registration costs. For typical Kubernetes service names with 2–4 labels, client-side costs remain under 9 ms total (3.1 ms decrypt + 6 ms verify), confirming Calypso's practicality.

Cryptographic operations. Table 4 compares Calypso's operation latencies against 3072-bit RSA as a computational baseline. Publisher-side operations—encryption and signing—incur 44–56× and 0.64–1.38× overheads due to HIBE pairing operations, though these costs are one-time registration expenses. Client-side operations show asymmetry: decryption remains comparable to RSA (1.12×), while signature verification imposes 33–43× overhead, becoming the dominant factor in end-to-end lookup latency (§8.2).

8.2 End-to-End Performance

To evaluate end-to-end performance, we compare Calypso against:

(1) a standard DNS setup with no access control
(2) a JWT-based approach where the DNS server verifies client credentials to authorize access to records
(3) an enclave-based approach using Intel SGX to isolate the DNS server from potentially malicious privileged software, with JWT tokens for client authentication
(4) a standard JEDI approach, which provides encrypted records and authentication but lacks enumeration resistance as it does not conceal the domain names in DNS queries

We configure each approach to use the same end-to-end path (**q** client → CoreDNS → etcd backend → CoreDNS → **q** client) and measure the latency of DNS lookups from a client perspective. All end-to-end measurements use UDP as the transport protocol to ensure fair comparison across approaches. For JEDI and Calypso, we pre-generate and provision the necessary keys to the client via the registration tool, etcd-client. We populate the DNS server with 1,000 service records distributed across 10 namespaces, modeling a large-scale, multi-tenant Kubernetes cluster deployment. This configuration is consistent with production deployments where CoreDNS has been benchmarked to handle 820–8,200 services [57]. It also reflects enterprise environments where organizations deploy 5–10 namespaces for multi-tenant isolation. For latency evaluation, we perform 4,000 DNS queries uniformly sampled across the registered services. We repeat the entire experiment 10 times and report percentile latencies (P50, P90, P95, P99) with 95% confidence intervals across trials. For cumulative distributions (CDFs), we report median distributions to characterize typical behavior while accounting for trial-to-trial variance. Both representations demonstrate consistent performance with coefficient of variation (CV) below 1% for most approaches, with plain UDP showing 3.2%.

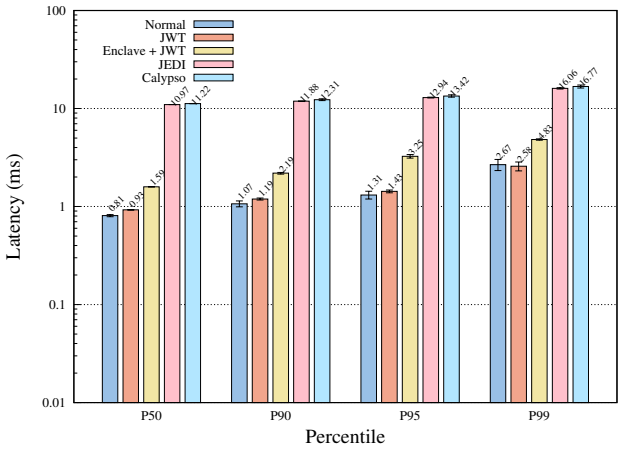


Figure 4: End-to-end DNS lookup latencies (P50, P90, P95, P99). Bars show mean latency across 10 independent trials; error bars indicate 95% confidence intervals. Non-overlapping error bars indicate statistically significant differences.

End-to-end latency. Figure 2 shows the median cumulative latency distributions across 10 trials, revealing three distinct performance tiers. The tight curves indicate highly consistent performance across trials. Normal DNS and JWT both cluster tightly around ∼1 ms median latency over UDP, establishing the baseline cost of unencrypted name resolution with optional authentication. The enclave-based approach introduces moderate overhead at ∼2 ms (2×), reflecting SGX context-switching costs for trusted execution. Calypso and JEDI occupy a third tier at ∼12 ms median over UDP, attributable to client-side HIBE decryption and verification operations. While Calypso incurs 12× overhead compared to unencrypted DNS, this drops to only 3–4× overhead when compared against standard encrypted DNS-over-HTTPS (DoH).

Figure 3 presents end-to-end request latencies for DoH, with UDP versions of JEDI and Calypso as baselines. While DoH protects against network-level eavesdropping, Calypso's threat model focuses exclusively on endpoint security. Since Calypso provides cryptographic end-to-end protection, the choice between UDP and DoH affects only performance, not security properties. DoH introduces modest latency overhead over UDP approaches due to TLS handshake and HTTPS encapsulation costs.

Figure 4 shows how latency overhead scales across percentiles. Error bars represent 95% confidence intervals computed across 10 independent trials, demonstrating consistent performance (all CV <4%). Compared to Normal, JEDI and Calypso exhibit ∼14× overhead at P50 (11.22 ms) that narrows to ∼6× at P99 (16.77 ms), indicating that baseline DNS variability dominates tail behavior. Calypso adds negligible overhead over JEDI (<0.8 ms across all percentiles) despite enumeration resistance, with client-side HIBE operations accounting for the bulk of latency in both schemes.

Latency breakdown. To understand where this overhead originates, Table 5 breaks down end-to-end latency into constituent phases: request preparation, DNS round-trip, and client-side cryptographic processing. Normal and JWT approaches spend nearly

Table 5: Latency breakdown of DNS operations by approach.

Approach	Prep (μs)	DNS (ms)	Decrypt + Verify (ms)	Total (ms)
Normal	2.16 ± 0.04	0.94 ± 0.00	—	0.94 ± 0.00
JWT	2.94 ± 0.08	1.07 ± 0.00	—	1.08 ± 0.00
JEDI	2.98 ± 0.14	1.39 ± 0.01	9.91 ± 0.01	11.30 ± 0.01
Calypso	3.37 ± 0.03	1.40 ± 0.01	10.13 ± 0.01	11.53 ± 0.01

Table 6: Server-side CPU overhead per query: Baseline CoreDNS vs. Calypso

Protocol	Normal (ms)	Calypso (ms)
DoH	2.673 ± 0.008	2.654 ± 0.011 (−0.7%)
UDP	0.635 ± 0.002	0.984 ± 0.001 (+54.8%)***

Values shown as mean ± standard error. Overhead percentages shown in parentheses.
*** $p < 0.001$ (two-tailed t-test); DoH difference not significant ($p = 0.100$).

all time (∼1 ms) on DNS lookups, while JEDI and Calypso shift the bottleneck to client-side decryption and verification (∼10 ms), which dominates their total latency of ∼11–12 ms. Despite Calypso's larger response payloads (Table 3), its enumeration resistance adds only 0.23 ms (2.0%) over JEDI, demonstrating that the overhead from obfuscating the name is minimal.

Server-side CPU overhead. To assess Calypso's impact on server-side CPU utilization, we measure the average CPU time per query on the CoreDNS server for both Calypso and a normal baseline CoreDNS setup without access control. Table 6 summarizes the results over 10 trials of 4,000 queries each, using both DoH and UDP transports. For DoH, Calypso shows no significant overhead ($p = 0.100$), as the HTTPS cost (∼2.04 ms) dominates the ∼0.35 ms server-side processing difference. For UDP, Calypso incurs a small 0.35 ms increase per query (+54.8%) from parsing the EDNS0 obfuscated name and processing the encrypted etcd entry, compared to normal DNS, which only fetches and returns. These results demonstrate minimal server-side overhead, especially under transport encryption, where Calypso's processing cost becomes negligible.

Scalability. Calypso's per-query overhead is constant and independent of the number of registered services. Server-side lookup is $O(1)$: the obfuscated name serves as a direct etcd key, requiring no enumeration or scanning of the keyspace. Client-side costs likewise depend only on domain label depth (§8.1), not on the number of registered services. Consequently, the server-side and client-side overheads reported above remain fixed whether the cluster hosts 1,000 or 100,000 services. Scaling beyond our testbed would therefore exercise etcd's well-characterized key-value scalability rather than Calypso's cryptographic layer.

9 Conclusion

Calypso strengthens cloud security by protecting service discovery systems, which are essential to microservice orchestration yet

expose sensitive reconnaissance data. Calypso applies zero-trust principles to the discovery data plane by embedding cryptographic mechanisms into the DNS stack, allowing these components to handle data without joining the trusted computing base. Dynamic policy evaluation, continuous verification, and post-quantum migration remain future work. To support further research, we have open-sourced Calypso at https://github.com/etclab/calypso-artifact.

References

[1] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. 2005. Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions. In *International Cryptology Conference (CRYPTO)*.

[2] Michel Abdalla, Eike Kiltz, and Gregory Neven. 2007. Generalized Key Delegation for Hierarchical Identity-Based Encryption. In *European Symposium on Research in Computer Security (ESORICS)*.

[3] Shweta Agrawal, Dan Boneh, and Xavier Boyen. 2010. Efficient Lattice (H)IBE in the Standard Model. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[4] Giuseppe Ateniese, Danilo Francati, David Nuñez, and Daniele Venturi. 2019. Match Me if You Can: Matchmaking Encryption and Its Applications. In *International Cryptology Conference (CRYPTO)*.

[5] Giuseppe Ateniese, Danilo Francati, David Nuñez, and Daniele Venturi. 2021. Match Me if You Can: Matchmaking Encryption and Its Applications. *Journal of Cryptology* 34, 3 (July 2021).

[6] The CoreDNS Authors. 2025. CoreDNS: DNS and Service Discovery. https://coredns.io. Accessed: 2025-07-25.

[7] Xiaolong Bai, Luyi Xing, Nan Zhang, XiaoFeng Wang, Xiaojing Liao, Tongxin Li, and Shi-Min Hu. 2016. Staying Secure and Unprepared: Understanding and Mitigating the Security Risks of Apple ZeroConf. In *IEEE Symposium on Security and Privacy*.

[8] Niko Bari and Birgit Pfitzmann. 1997. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[9] Josh Benaloh and Michael de Mare. 1993. One-way Accumulators: A Decentralized Alternative to Digital Signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[10] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. 2005. Hierarchical Identity Based Encryption with Constant Size Ciphertext. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[11] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *International Cryptology Conference (CRYPTO)*.

[12] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *International Cryptology Conference (CRYPTO)*.

[13] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security Symposium*.

[14] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM Conference on Computer and Communications Security (CCS)*.

[15] Stuart Cheshire and Marc Krochmal. 2013. DNS-Based Service Discovery. RFC 6763. https://www.rfc-editor.org/info/rfc6763

[16] Cloud Native Computing Foundation. 2024. Open Policy Agent (OPA). CNCF Graduated Project. https://www.openpolicyagent.org/ Accessed: 2025-02-09.

[17] CoreDNS. 2023. CoreDNS Policy Plugin. GitHub. https://github.com/coredns/policy Accessed: 2025-02-09.

[18] cve-2018-1085 2018. CVE-2018-1085. https://www.cve.org/CVERecord?id=CVE-2018-1085. Common Vulnerabilities and Exposures.

[19] cve-2018-16886 2019. CVE-2018-16886. https://www.cve.org/CVERecord?id=CVE-2018-16886. Common Vulnerabilities and Exposures.

[20] cve-2019-3779 2019. CVE-2019-3779: Cloud Foundry Container Runtime Allows A User To Bypass Security Policy When Talking To ETCD. https://www.cve.org/CVERecord?id=CVE-2019-3779. Common Vulnerabilities and Exposures.

[21] cve-2020-15115 2020. CVE-2020-15115: No Minimum Password Length In Etcd. https://www.cve.org/CVERecord?id=CVE-2020-15115. Common Vulnerabilities and Exposures.

[22] cve-2020-15136 2020. CVE-2020-15136: Improper Authentication In Etcd. https://www.cve.org/CVERecord?id=CVE-2020-15136. Common Vulnerabilities and Exposures.

[23] cve-2021-28235 2023. CVE-2021-28235. https://www.cve.org/CVERecord?id=CVE-2021-28235. Common Vulnerabilities and Exposures.

[24] cve-2022-2837 2023. CVE-2022-2837. https://www.cve.org/CVERecord?id=CVE-2022-2837. Common Vulnerabilities and Exposures.

[25] cve-2023-30464 2024. CVE-2023-30464. https://www.cve.org/CVERecord?id=CVE-2023-30464. Common Vulnerabilities and Exposures.

[26] cve-2023-32082 2023. CVE-2023-32082: Etcd Key Name Can Be Accessed Via LeaseTimeToLive API. https://www.cve.org/CVERecord?id=CVE-2023-32082. Common Vulnerabilities and Exposures.

[27] cve-2024-42480 2024. CVE-2024-42480: Kamaji's RBAC Roles For etcd Are Not Disjunct. https://www.cve.org/CVERecord?id=CVE-2024-42480. Common Vulnerabilities and Exposures.

[28] Datadog. 2023. Kubernetes DNS Enumeration Detection Rule. Datadog Cloud Security Management. https://docs.datadoghq.com/security/default_rules/kubernetes_dns_enumeration/ Accessed: 2025-02-09.

[29] 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.

[30] Amos Fiat and Moni Naor. 1993. Broadcast Encryption. In *International Cryptology Conference (CRYPTO)*.

[31] Craig Gentry and Alice Silverberg. 2002. Hierarchical ID-Based Cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

[32] Miek Gieben. 2025. Alternative (mor granular) approach to a DNS library. https://github.com/miekg/dns. Accessed: 2025-10-31.

[33] Arnt Gulbrandsen and Dr. Levon Esibov. 2000. A DNS RR for specifying the location of services (DNS SRV). RFC 2782. https://www.rfc-editor.org/info/rfc2782

[34] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. 2021. PrivateDrop: Practical Privacy-Preserving Authentication for Apple AirDrop. In *USENIX Security Symposium*.

[35] P. Hoffman, ICANN, P. McManus, and Mozilla. 2018. DNS Queries over HTTPS (DoH). RFC 8484. https://www.ietf.org/rfc/rfc8484.txt

[36] Jeremy Horwitz and Ben Lynn. 2002. Toward Hierarchical Identity-Based Encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[37] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). RFC 7858. https://www.rfc-editor.org/info/rfc7858

[38] Christian Huitema and Daniel Kaiser. 2020. DNS-Based Service Discovery (DNS-SD) Privacy and Security Requirements. RFC 8882. https://www.rfc-editor.org/info/rfc8882

[39] Infoblox. 2023. Themis: Open-Source Policy Engine. GitHub. https://github.com/infobloxopen/themis Accessed: 2025-02-09.

[40] Intel 2014. *Intel Software Guard Extensions Programming Reference*. Intel.

[41] istio 2025. The Istio service mesh. https://istio.io/latest/about/service-mesh/. Accessed: 2025-09-23.

[42] Michael B. Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Token (JWT). RFC 7519. https://www.rfc-editor.org/info/rfc7519

[43] Daniel Kaiser and Marcel Waldvogel. 2014. Adding Privacy to Multicast DNS Service Discovery. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.

[44] Hugo Krawczyk and Pasi Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. https://www.rfc-editor.org/info/rfc5869

[45] Sam Kumar, Yuncong Hu, Michael P. Andersen, Raluca A. Popa, and David E. Culler. 2019. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *USENIX Security Symposium*.

[46] Allison Lewko and Brent Waters. 2014. Why Proving HIBE Systems Secure is Difficult. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[47] Mengyuan Li. 2022. *Understanding and Exploiting Design Flaws of AMD Secure Encrypted Virtualization*. Ph. D. Dissertation. https://etd.ohiolink.edu/.

[48] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security Symposium*.

[49] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security Symposium*.

[50] linkerd 2025. The world's most advanced service mesh. https://linkerd.io/

[51] 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)*.

[52] Jeffrey Pang, Ben Greenstein, Damon McCoy, Srinivasan Seshan, and David Wetherall. 2007. Tryst: The Case for Confidential Service Discovery. In *Workshop on Hot Topics in Networks (HotNets)*.

[53] Scott W. Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. 2020. Zero Trust Architecture. doi:10.6028/NIST.SP.800-207

[54] Nate Sales. 2025. q: A tiny and feature-rich command line DNS client with support for UDP, TCP, DoT, DoH, DoQ, and ODoH. https://github.com/natesales/q. Accessed: 2025-09-25.

[55] Adi Shamir. 1984. Identity-based Cryptosystems and Signature Schemes. In *International Cryptology Conference (CRYPTO)*.
[56] Edgeless Systems. 2025. Welcome to EGo. https://docs.edgeless.systems/ego.
[57] CoreDNS Team. 2018. Scaling CoreDNS in Kubernetes Clusters. https://github.com/coredns/deployment/blob/master/kubernetes/Scaling_CoreDNS.md Production benchmarks with 820-8,200 services.
[58] Trail of Bits. 2019. Kubernetes Security Assessment. CNCF Security Audit Working Group. https://github.com/kubernetes/sig-security/blob/main/sig-security-external-audit/security-audit-2019/findings/Kubernetes%20Final%20Report.pdf Accessed: 2025-02-09.
[59] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
[60] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*.
[61] David J. Wu, Ankur Taly, Asim Shankar, and Dan Boneh. 2016. Privacy, Discovery, and Authentication for the Internet of Things. In *European Symposium on Research in Computer Security (ESORICS)*.
[62] Yang Yang, Robert H. Deng, Guomin Yang, Yingjiu Li, HweeHwa Pang, Minming Huang, Rui Shi, and Jian Weng. 2024. PriSrv: Privacy-Enhanced and Highly Usable Service Discovery in Wireless Communications. In *Network and Distributed System Security Symposium (NDSS)*.
[63] Yang Yang, Guomin Yang, Yingjiu Li, Pengfei WU, Rui Shi, Minming Huang, Jian Weng, HweeHwa Pang, and Robert H. Deng. 2025. PriSrv+: Privacy and Usability-Enhanced Wireless Service Discovery with Fast and Expressive Matchmaking Encryption. In *Network and Distributed System Security Symposium (NDSS)*.

A Negative Hypergeometric Distribution

The *negative hypergeometric* distribution has the *probability mass function (PMF)*:

$$\Pr[T = t] = \frac{\binom{t-1}{k-1}\binom{N-t}{m-k}}{\binom{N}{m}}, \qquad t = k, k+1, \ldots, N-m+k$$

In our setting, T is the number of queries needed to find $k \le m$ valid domain names. The corresponding mean and variance for this distribution are:

$$\mathbb{E}[T] = \frac{k(N+1)}{m+1}, \qquad \mathrm{Var}[T] = \frac{k(N+1)(m-k+1)(N-m)}{(m+1)^2(m+2)}.$$

Table 7 lists the expected number of queries and corresponding total time to brute-force a cluster with $m = 1,000$ domain names drawn from a corpus of $N = 1,000,000$ possible names, where each query takes 0.90 ms (measured in our evaluation; see §8).

Table 7: Queries needed to brute force a subset of 1,000 domain names out of 1 million possible domain names.

Discovered Domains	Expected #Queries	Standard Deviation	Total Time
1	1,000	998	< 2s
100	99,901	9,469	1m21s – 1m38s
200	199,801	12,626	2m48s – 3m11s
300	299,701	14,466	4m17s – 4m43s
400	399,601	15,467	5m46s – 6m14s
500	499,501	15,788	7m15s – 7m44s
...			
1,000	999,002	998	15m

Based on sequential queries with measured latency of 0.90 ms.