CODoH: Privacy-Preserving Caching for Oblivious DNS over HTTPS

Anonymous Author(s)

Abstract

Oblivious DNS over HTTPS (ODoH) enhances DNS privacy by routing queries through a proxy so that no single party can link a client's IP address with its DNS queries. However, because ODoH encrypts each query individually, the proxy cannot cache responses. We present CODoH (Cacheable Oblivious DNS over HTTPS), a cacheable extension to ODoH that reduces DNS resolution latency while preserving ODoH's separation of knowledge. CODoH places a proxy-side cache inside a trusted execution environment—namely, an Intel SGX enclave—and uses end-to-end encryption to ensure that the proxy never learns plaintext queries or cached responses. To prevent caching from becoming a new inference surface, CODoH defends against cache-state probing and set-difference (bracketing) attacks with *(i)* cover responses supplied by the resolver, *(ii)* batched cache updates that mix many clients' inserts, and *(iii)* an oblivious RAM (ORAM) backend that hides cache access patterns. CODoH also enforces correctness and freshness of cached records via resolver authorization and replay protection. We implement CODoH as CoreDNS extensions and evaluate it on an Azure SGX testbed. CODoH reduces median latency by 2.7× over ODoH on cache hits (17.9 vs. 48.0 ms) and by 2× under Zipf traffic, with under 1 ms of SGX overhead.

Keywords

DNS, ODoH, Intel SGX, ORAM

1 Introduction

The *Domain Name System (DNS)* [37, 38] is a critical piece of Internet infrastructure: nearly every application session begins by translating human-readable domain names into IP addresses. DNS's original design emphasized performance and deployability, relying heavily on caching and hierarchical delegation, but it provided neither security nor privacy. Although mechanisms such as DNSSEC [23] improve integrity and authentication, they do not conceal who is querying or what is being queried. Encrypted transports such as DNS-over-TLS (DoT) [24] and DNS-over-HTTPS (DoH) [22] protect DNS traffic from passive observers on the network path, but they still concentrate sensitive information at the resolver, which learns both the client's network identifier and the plaintext domain name. This concentration matters in practice because a small number of large public resolvers serve a significant fraction of global DNS traffic [17, 44], creating powerful vantage points for profiling, censorship, and monetization.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–15

https://doi.org/XXXXXXX.XXXXXXX

Oblivious DNS over HTTPS (ODoH) [28, 45] eliminates the residual privacy gap in encrypted DNS by splitting trust between two roles: an oblivious proxy that learns the client's identity but not the plaintext query, and an oblivious target resolver that learns the query but not the client. Apple's iCloud Private Relay [3] uses ODoH for DNS resolution, where Apple operates the proxy, and a partner such as Cloudflare operates the target [16]. This separation, however, comes at a cost: ODoH forfeits one of DNS's primary performance advantages: shared caching near clients. Each query is encrypted end-to-end to the target, which prevents the proxy from caching plaintext responses. The resulting lack of proxy-side caching increases latency, raises upstream load, and limits deployability, particularly for popular domains and latency-sensitive applications [26, 48]. Restoring proxy-side caching without reintroducing linkability would preserve ODoH's privacy guarantees while recovering much of DNS's practical performance benefits.

Caching introduces new side channels. Caching, however, fundamentally changes the security story. A cache introduces persistent, query-dependent state at the proxy, and that state becomes a side channel even if the proxy cannot decrypt queries. In the ODoH setting, an adversarial proxy can manipulate scheduling and concurrency, inject probe traffic as a client, restart cache components to create "fresh windows," and selectively omit cache inserts—all with the goal of learning something about a victim's query from how cache state changes over time. Worse, naïve caching enables classic bracketing (set-difference) attacks in which the proxy isolates a victim request, compares cache state before and after, and infers which name was inserted. These attacks do not arise in baseline ODoH because ODoH traffic remains intentionally non-cacheable; they arise precisely because caching creates state. Any solution therefore must *(i)* keep the proxy from learning plaintext queries and responses, *(ii)* ensure cached answers remain authorized and fresh, and *(iii)* prevent cache behavior itself—hits, misses, inserts, and access patterns—from becoming a reliable distinguisher under an active proxy adversary.

This paper. Our key insight is that ODoH can support caching by isolating only the cache inside a small, remotely attestable trusted component co-located with the proxy, while masking cache-induced side channels so that even an active proxy cannot link a client's query to specific cache state changes. We present *CODoH (Cacheable Oblivious DNS over HTTPS)*, an extension that reduces DNS latency while preserving ODoH's separation of knowledge. CODoH places the cache inside an Intel SGX enclave [25, 35] alongside the proxy and introduces a protocol that allows the target to authorize cache inserts without revealing cache contents.

To address cache-specific inference risks, CODoH combines three techniques: *(1)* it structures responses to avoid exposing explicit cache hit/miss signals, *(2)* it expands and mixes cache updates using cover responses and batched commits to prevent attribution to any single request, and *(3)* it hides cache access patterns with an

oblivious RAM (ORAM) [20, 46] backend to block architectural leakage. Together, these mechanisms recover caching's performance benefits while maintaining ODoH's trust split.

Contributions. We make the following contributions:

- **Design:** We design CODoH, a cacheable extension to ODoH that places proxy-side caching inside an Intel SGX enclave while preserving ODoH's separation between client identity and query contents.
- **Security mechanisms:** We develop cache-specific defenses against active proxy attacks, including cover responses, batched cache updates, restart/omission handling, and ORAM-backed oblivious caching to mitigate set-difference and cache-probing inference.
- **Correctness and freshness:** We enforce that cached answers remain authorized and fresh via target-authenticated cache inserts and replay protection compatible with SGX's time limitations.
- **Prototype and evaluation:** We implement CODoH as extensions to CoreDNS [5]—a graduated Cloud Native Computing Foundation (CNCF) [14] project—and evaluate it on an Azure SGX testbed using macrobenchmarks, microbenchmarks, and page-load experiments. CODoH reduces median latency by 2.7× over ODoH on cache hits and by 2× under Zipf traffic, with under 1 ms of SGX overhead.

2 Background and Motivation

2.1 ODoH

ODoH mitigates resolver-side linkability by decoupling client identity from query contents. It introduces two roles: a *proxy* that relays encrypted messages and a *target* resolver that decrypts and answers them. Using *Hybrid Public Key Encryption (HPKE)* [6], the client encrypts each query under the target's public key and sends it through the proxy, which sees only ciphertext and the client's IP address, while the target sees the plaintext query but only the proxy's IP. As long as the proxy and target do not collude (or an attacker does not compromise both), neither can link a user to a query. This design, however, comes at a cost: ODoH encrypts each exchange with fresh randomness, preventing the proxy from caching responses and thereby sacrificing one of DNS's primary performance advantages: serving popular records from a nearby shared cache.

2.2 Intel SGX

Trusted execution environments (TEEs) protect code and data from a potentially hostile software stack on the same machine. Intel SGX [25, 35] is a TEE that extends the x86-64 architecture to support isolated execution in protected regions called enclaves, allowing user-space applications to run securely even in the presence of a malicious operating system or hypervisor, as well as some physical attackers. SGX encrypts and integrity-protects enclave memory when stored in DRAM and decrypts it only within the trusted CPU package, preventing privileged software from observing or modifying enclave state. Although SGX minimizes the trusted computing base by isolating only selected portions of an application, it can introduce compatibility challenges with legacy software, which middleware frameworks [4, 7, 21, 49] help mitigate. Like other TEEs, SGX's security guarantees can be weakened by side-channel attacks [8, 9, 31–33, 50, 51].

A core feature of SGX is remote attestation, which enables a remote party to verify that specific software is running inside a genuine enclave on authentic hardware. During attestation, the processor signs a measurement of the enclave's initial state (`MRENCLAVE`) and the identity of the signing entity (`MRSIGNER`), producing a verifiable *quote*. The enclave can also bind runtime data (such as a public key) to the quote via a signed *user-data* field, enabling clients to verify both code integrity and hardware authenticity in one step.

2.3 Prior Work on DNS Privacy

CODoH builds on a long line of DNS-privacy research that tries to break the link between *who* a client is and *what* name it resolves, without making DNS too slow or too expensive to deploy. Prior systems typically trade among three levers: *(1)* hiding a query among many (with range queries / cover queries), *(2)* hiding the sender among many (mix networks), and *(3)* hiding the query from the infrastructure (using trusted hardware or cryptography). CODoH builds directly on these ideas: it preserves ODoH's low-latency, proxy-based separation of identity and query contents while restoring proxy-side caching using techniques that dilute and obscure per-query observables, without incurring the deployability constraints or high latency of earlier approaches.

Range queries and cover traffic. Early privacy-preserving DNS systems used range (dummy) queries [10, 53, 54] to hide a user's true lookup among additional queries, thereby creating an anonymity set in which an observer cannot distinguish the real query from decoys. In practice, these approaches incur significant bandwidth and latency overhead (since the client must wait for all resolutions), and require carefully chosen dummy hostnames to avoid revealing inter-query dependencies. CODoH avoids such overheads and client-side concerns by forgoing dummy queries and instead injecting only dummy responses.

Mix networks. Mix networks [11] address a complementary problem by concealing a query's origin through delayed and shuffled forwarding across multiple relays. This design offers strong resistance to traffic analysis but incurs substantial latency, making it impractical for DNS, where even small delays degrade user experience. Onion-routing approaches such as DNS-over-Tor [1, 39] similarly impose comparatively high latency for DNS workloads. Hybrid designs [19] therefore often reserve mixing for long-tail queries while serving popular names locally using alternative mechanisms, sometimes including broadcast-style techniques. Rather than multi-hop mixing or broadcast, CODoH pursues the same intuition—make any single query hard to isolate—through cache-centric mechanisms that aggregate and mask cache state changes.

Trusted execution environments for DNS. PDoT [41] takes a different approach by running the resolver inside an Intel SGX enclave to limit what the operator can observe, while using remote attestation to assure clients that they are communicating with the protected code. PDoT also highlights an important residual-leakage issue for recursive resolution: even if the recursive resolver is protected, traffic to authoritative name servers can reveal information.

CODoH draws two lessons from this line of work. First, TEEs can help enforce privacy properties even when a machine owner is not fully trusted. Second, minimizing the trusted computing base matters: instead of placing an entire resolver inside a TEE, CODoH uses trusted execution only for narrowly scoped stateful functionality associated with caching, and composes it with ODoH's proxy–target separation.

Private Information Retrieval. *Private Information Retrieval (PIR)* [12] offers an appealing "cryptography-only" approach: a client retrieves a record from a server's database without revealing which record it requested. Applied to DNS, PIR can prevent a resolver from learning query contents even without a proxy. However, PIR-based DNS designs [36, 52] face practical friction: DNS data are dynamic (TTL expiration, frequent updates), caches are not fixed databases, and handling misses and updates tends to require additional protocol machinery. Moreover, PIR can impose significant computation and bandwidth costs at resolver scale, and many proposals rely on specialized hardware acceleration to become cost-effective. These challenges make PIR an informative point of comparison, but not the design basis for CODoH's approach.

Summary. Taken together, this prior work suggests a recurring design pattern: systems that provide strong privacy rely on either *(i)* large anonymity sets created by aggregation (range queries / mixing), or *(ii)* strong server-side opacity (TEEs or heavy cryptography). CODoH's goal is to combine the spirit of these ideas while preserving the deployability and latency properties of modern encrypted DNS.

3 Goals, Assumptions, and Threat Model

CODoH *inherits* ODoH's goals, assumptions, and baseline adversary model, and then *adds* new requirements and adversarial capabilities that arise specifically from caching.

3.1 ODoH Goals and Assumptions (Inherited)

ODoH privacy goal. ODoH aims to prevent any single non-colluding party from learning both *(i)* a client identifier (i.e., IP address) and *(ii)* the plaintext DNS query/response. It achieves this separation by splitting roles between a proxy and a target resolver.

ODoH adversary model. ODoH assumes a powerful active network attacker that can observe, replay, delay, drop, and inject messages, but that cannot break standard cryptographic primitives. ODoH further assumes compromise of *either* the proxy *or* the target may occur, but not both simultaneously (equivalently, the proxy and target do not collude).

ODoH operational assumptions. ODoH makes the following operational assumptions:

- Clients obtain and authenticate the target's public key material (the bootstrapping mechanism is out of scope of the specification).
- The proxy and target operate under independent administrative control; if they collude or are jointly compromised, unlinkability is lost by design.
- Proxies do not add client-identifying information when forwarding requests (e.g., `Forwarded` headers), and clients do not send identifying state (e.g., cookies) to proxies.
- The target follows the protocol and does not intentionally introduce client-specific identifiers (e.g., via client-unique configuration) that would enable linkability.

3.2 CODoH: Added Requirements and Stronger Adversary (New)

Caching introduces persistent state at the proxy and makes cache behavior a potential side channel. CODoH therefore adds both a *performance requirement* and *cache-specific security goals*.

Performance requirement. CODoH should reduce client-perceived latency for cacheable DNS records relative to baseline ODoH, without requiring changes in the client's trust assumptions about proxy–target separation. Client latency for uncached DNS records should be similar to that with ODoH.

Primary cache-specific attack vector. A cache enables *state-based inference* attacks that do not exist in baseline ODoH because standard ODoH traffic is intentionally non-cacheable. The canonical example is a *set-difference (bracketing) attack* in which a malicious proxy:

(1) Learns (or approximates) the current cache state.
(2) Allows exactly one victim query to proceed while queuing other traffic.
(3) Learns the cache state again.
(4) Computes the difference to isolate which entry was added, thereby narrowing (and potentially identifying) the victim's query.

CODoH's security goals below target this class of attacks and related cache-manipulation strategies.

Security goals. CODoH targets the following goals in addition to ODoH's baseline unlinkability:

G1 *Authorized and fresh cached answers.* A malicious proxy must not be able to cause clients to accept cached DNS answers that the target did not authorize, or that are no longer valid. This goal rules out cache poisoning, tampering, and stale-answer replay that would create split views or incorrect resolutions.

G2 *Cache-state indistinguishability under active probing and scheduling.* The proxy must not be able to learn meaningful information about a particular client's query by manipulating or comparing cache state across time—e.g., via set-difference/bracketing, cache priming, selective withholding, restart-based "reset windows," or similar strategies that rely on proxy-controlled scheduling and concurrency.

G3 *Query equality and profiling resistance.* The proxy must not be able to determine whether two client requests correspond to the same DNS query (or otherwise build per-client/per-group query profiles) from cache behavior, beyond what is already available from observing client network identifiers and coarse traffic features (as allowed by the out-of-scope items in §3.3).

Threat model. CODoH retains ODoH's non-collusion assumption. Within that assumption, CODoH considers the following *in-scope adversary capabilities*:

- *Network control at the proxy:* Observe, delay, drop, replay, reorder, inject protocol messages on proxy-facing links.
- *Active proxy deviation:* Arbitrary protocol deviation to bias cache behavior (e.g., priming, withholding inserts, selective forwarding).
- *Scheduling control:* Manipulate concurrency and queuing at the proxy to amplify inference (e.g., isolate a victim in an "epoch").
- *Cache lifecycle control:* Restart the cache component to reset state.
- *Proxy-as-client probing:* Send probe queries to try to enumerate or test cache contents.

CODoH also considers *architectural* leakage that arises from the cache component's externally observable behavior, including:

- Page-level access patterns visible via architectural mechanisms (e.g., page faults / paging activity).
- Timing differences between different logical cache operations.
- Message size differences.

CODoH aims to prevent these channels from enabling G2/G3-style inferences.

3.3 Out of Scope Threats and Limitations

CODoH inherits several limitations from ODoH and encrypted DNS protocols more broadly, and it makes additional scoping choices for clarity.

Traffic correlation and website fingerprinting. As in ODoH, CODoH does not attempt to defeat global traffic-analysis adversaries that can correlate flows across multiple network vantage points or over long time horizons (e.g., by combining DNS observations with application-layer behavior). Furthermore, CODoH assumes client-side indistinguishability at the network layer. Distinctive client traffic patterns—such as bursts of dependent queries and characteristic timing—may still enable website fingerprinting or related correlation attacks by an observer at or near the proxy.

Salari et al. [43] demonstrate this risk for ODoH: using deep learning over encrypted traffic features (packet sizes, counts, and timing), they show that a passive adversary near the proxy can identify visited websites with high accuracy, and they propose TLS-level defenses based on padding, fragmentation, and dummy traffic. CODoH is complementary to this line of work: whereas Salari et al. analyze and defend against fingerprinting in ODoH, we address the privacy and deployability challenges introduced by adding caching. CODoH can (and does) incorporate traffic-perturbation techniques such as padding to reduce protocol-level distinguishers, but it does not by itself claim to eliminate inference driven by higher-layer client behavior.

Microarchitectural attacks on trusted hardware. CODoH does not defend against *microarchitectural* side channels or fault attacks [40] against the cache's trusted-execution substrate (e.g., transient-execution attacks such as Spectre/Meltdown-class issues [29, 34, 50], or other CPU-level leakage). We assume either that commodity mitigations suffice for these threats or that they fall outside our attacker model. Our in-scope side-channel model is limited to *architectural* channels as described in §3.2.

Malicious target and proxy–target collusion. As in ODoH, CODoH does not ensure DNS correctness against a malicious target, and it does not address active deanonymization in which a malicious target crafts answers to elicit identifying client behavior. Finally, if the proxy and target collude, ODoH's privacy goal fails by construction; CODoH cannot prevent this and instead relies on operational separation between the two roles.

4 Design

In this section, we incrementally develop the CODoH protocol, introducing additional mechanisms as needed. Figure 1 shows the system architecture, and Algorithm 1 presents pseudocode for the main steps.

4.1 Syntax and Notation

We use the following notation for HPKE, which consists of algorithms for creating an *HPKE context*, and then *context methods* for sealing, opening, and deriving (exporting) keys with that context. We use $\$\!\!\rightarrow$ to denote randomized algorithms and $\rightarrow$ for deterministic algorithms.

HPKE.SetupS($\mathsf{pk_R}$, info) $\$\!\!\rightarrow$ (enc, $\mathsf{ctx_S}$): Creates a sender context $\mathsf{ctx_S}$ and an encapsulation enc for receiver public key $\mathsf{pk_R}$ under domain separation string info.

HPKE.SetupR($\mathsf{sk_R}$, info, enc) $\rightarrow$ $\mathsf{ctx_R}$: Creates the matching receiver context $\mathsf{ctx_R}$ given private key $\mathsf{sk_R}$, info, and enc.

ctx.Seal(aad, pt) $\$\!\!\rightarrow$ ct: AEAD-encrypts plaintext pt with associated data aad using the context's internal key/nonce schedule.

ctx.Open(aad, ct) $\rightarrow$ pt | $\perp$: AEAD-decrypts and authenticates. On authentication failure, returns the error symbol $\perp$.

ctx.Export(label, ℓ) $\rightarrow$ k: Deterministically derives ℓ bytes of key material from the context using an exporter label (domain separation within context).

We assume nonces are unique per key, and that ciphertexts include the AEAD authentication tag.

Finally, let $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\ell$ denote a cryptographic hash function that maps arbitrary bit strings to strings of ℓ bits.

In ODoH, the client invokes HPKE.SetupS and ctx.Seal to encrypt a DNS query end-to-end to the target's public key, while the target invokes HPKE.SetupR and ctx.Open to decrypt the query. The domain-separation string info ensures that keys derived for different purposes are cryptographically independent, preventing cross-protocol or cross-use key reuse; for example, ODoH uses the fixed value `"odoh query"` when deriving keys for query encryption. In both ODoH and CODoH, the aad field binds unencrypted protocol metadata—such as nonces and cryptographic suite identifiers—to the encrypted payload.

4.2 Base Protocol

Bootstrap. As in baseline ODoH, we assume that clients securely obtain the target's public key $\mathsf{pk_T}$ before issuing queries. In addition, upon startup the enclave generates a public–private key pair $(\mathsf{pk_E}, \mathsf{sk_E})$ and produces an attestation quote that cryptographically binds the public key $\mathsf{pk_E}$ to the enclave's measurement as *user-data*. Clients retrieve the enclave's public key and validate its attestation quote before use. The mechanism for discovering and distributing

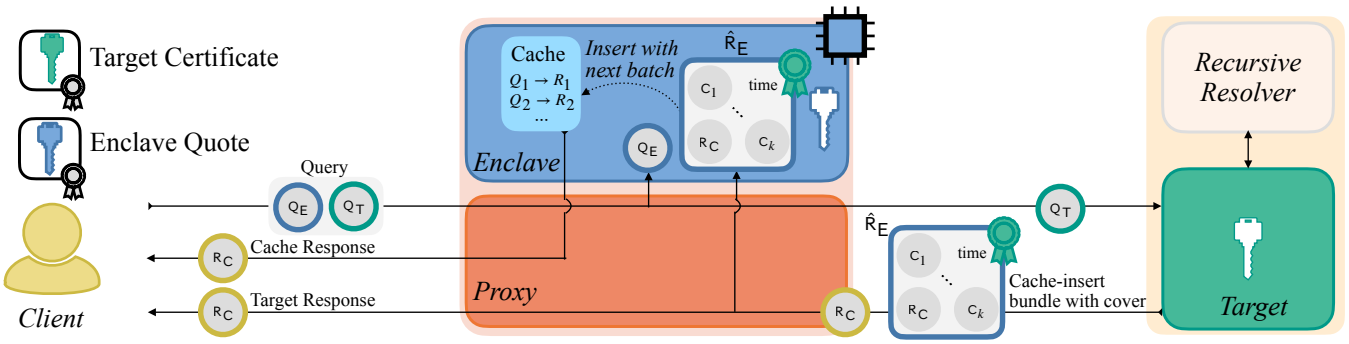

Figure 1: CODoH architecture. Prior to issuing queries, the client retrieves and verifies the target's certificate and the enclave's attestation quote (binding their respective public keys). In this diagram, the target is co-located with a recursive resolver.

this keying material is outside the scope of this paper, but likely involves fetching it from a well-known service.

Query. CODoH uses two independent HPKE exchanges per query: one to the target and one to the enclave, resulting in ciphertexts Q_T and Q_E, respectively.

Response. Upon receiving a request, the proxy forwards Q_E to the enclave, which decrypts it to recover the query and checks its in-memory cache. The enclave derives a response key from the request's HPKE context via Export, encrypts either the cached response or a cache-miss indicator under that key, and returns the resulting ciphertext to the proxy, which forwards it to the client.

In tandem with forwarding the request to the enclave, the proxy forwards Q_T to the target. The target responds as in baseline ODoH by encrypting the DNS response to the client, forming R_C. In addition, it constructs a *cache-insert bundle* $\hat{R}_E$ encrypted to the enclave's public key pk_E. This bundle contains the DNS response plus its TTL and timestamp (see §4.4), and is authenticated with the target's signature. Upon receiving the reply, the proxy forwards the baseline ODoH response R_C to the client and delivers the enclave-encrypted $\hat{R}_E$ to the enclave.

As in baseline DNS, the response includes the original question, which allows the enclave to unambiguously associate each response with its query. The enclave decrypts $\hat{R}_E$, verifies the target's signature, extracts the query from the DNS response, and inserts the resulting (query, response) pair into the cache.

Message padding. Differences in size (and processing time) between cache-hit and cache-miss responses could reveal cache status to the proxy or a network observer. To prevent this leakage, CODoH pads responses to fixed-size buckets (e.g., the next 1 KB, 2 KB, or 4 KB boundary), and performs the same HPKE operations on the resultant message regardless of whether the enclave's response is a cache hit or miss message. Since this padding also helps mitigate website fingerprinting attacks (such as those demonstrated by Salari et al. [43]), CODoH likewise pads the query messages.

4.3 Preventing Set-Difference Attacks

Returning both a cache response and a resolver response prevents the proxy from trivially determining whether a query caused a cache hit or miss. However, this mechanism does not stop a *set-difference attack*: the proxy can still *(1)* enumerate the cache, *(2)* process a single victim query while queuing all other queries, *(3)* re-enumerate the cache, and *(4)* compute the difference between the two states, uncovering the victim query.

Cover responses. A first step in mitigating set-difference attacks is to inject *cover responses* into the cache so that cache state changes are larger and harder to attribute to a single query. Since the enclave cannot generate responses on its own, CODoH uses the resolver as a source of cover traffic. For each query, the cache-insert bundle $\hat{R}_E$ that the resolver returns contains not only the real response, but also k cover responses for randomly selected domains.

Critically, cover domains must be indistinguishable from real queries. If the resolver samples covers uniformly from a "Top 1M" list while the real query targets a long-tail domain, a proxy could identify the long-tail entry as genuine during a set-difference attack. To prevent this, the resolver should draw covers from a mixture distribution that spans both popular and long-tail names. Formally, the cover distribution D should approximate the empirical query distribution Q, as by bounding the likelihood ratios $Q(x)/D(x)$ across domain names.

A natural alternative is to cache only an allowlist of popular domains. This reduces cache-state sensitivity but limits utility and adaptability: it misses long-tail reuse, wastes capacity when popularity shifts, and may not match the popularity distribution of specific client populations.

Batching cache updates. Even with cover responses, a set-difference attack remains possible because a single victim query produces only a small cache update. CODoH mitigates this by delaying insertions and committing them in *batches* for a configurable number of queries B, so that each victim's entries are mixed with many others. Specifically, the enclave accumulates candidate insertions from recent queries—B real responses and their Bk associated covers—and applies them atomically in batches of size $B(1+k)$.

Batch boundaries define the "epoch" over which cache state changes. If the proxy can force or deterministically predict these epochs, set-difference attacks become significantly easier. For example, if commits occur exactly every B arrivals, the proxy could queue other clients, allow a single victim query to enter a fresh batch, and

Algorithm 1 Base Protocol

function Client.MakeQuery(pk_T, pk_E, query, aad)
- ▷ *Encrypt to target* ◁
- $(enc_T, ctx_T) \leftarrow\$$ HPKE.SetupS(pk_T, "odoh query")
- $ct_T \leftarrow\$ ctx_T$.Seal(aad, Pad(query))
- $Q_T \leftarrow (enc_T, ct_T)$
- $k_t \leftarrow ctx_T$.Export("odoh response", ℓ)
- ▷ *Encrypt to enclave* ◁
- $(enc_E, ctx_E) \leftarrow\$$ HPKE.SetupS(pk_E, "codoh cache query")
- $ct_E \leftarrow\$ ctx_E$.Seal(aad, query)
- $Q_E \leftarrow (enc_E, ct_E)$
- $k_e \leftarrow ctx_E$.Export("codoh cache response", ℓ)
- ▷ *Return the two HPKE ciphertexts and response keys* ◁
- **return** (Q_T, k_t, Q_E, k_e)

function Enclave.Lookup(Q_E, aad)
- $ctx_E \leftarrow$ HPKE.SetupR(sk_E, "codoh cache query", Q_E.enc)
- query $\leftarrow ctx_E$.Open(aad, Q_E.ct)
- $k_e \leftarrow ctx_E$.Export("codoh cache response", ℓ)
- resp $\leftarrow$ Cache.Get(Unpad(query))
- nonce $\leftarrow\$ \{0,1\}^\ell$
- $R_C \leftarrow\$$ AEAD.Enc(nonce, aad, Pad(resp))
- **return** R_C

function Target.Resolve(Q_T, aad)
- $ctx_T \leftarrow$ HPKE.SetupR(sk_T, "odoh query", Q_T.enc)
- query $\leftarrow ctx_T$.Open(aad, Q_T.ct)
- $k_t \leftarrow ctx_T$.Export("odoh response", ℓ)
- resp $\leftarrow$ Resolve(Unpad(query))
- nonce $\leftarrow\$ \{0,1\}^\ell$
- $R_C \leftarrow\$$ AEAD.Enc(nonce, aad, Pad(resp))
- **yield** R_C ▷ *Return response for client*
- covers $\leftarrow$ ResolveCovers(k)
- bundle $\leftarrow$ (resp, covers, timestamp)
- $\sigma \leftarrow$ Sign(sk_T, H(bundle))
- $(enc_E, ctx_E) \leftarrow\$$ HPKE.SetupS(pk_E, "codoh cache insert")
- $ct_E \leftarrow\$ ctx_E$.Seal(aad, bundle $\|$ σ)
- $\hat{R}_E \leftarrow (enc_E, ct_E)$
- **return** $\hat{R}_E$ ▷ *Return response for enclave*

fill the remaining $B-1$ slots with its own probes, reducing uncertainty about the inserted entries. It could also align cache enumeration with commit times to minimize noise and improve attribution. To prevent this, the enclave selects commit points independently of proxy-controlled events (e.g., probabilistically choosing a commit point around B).[1]

To prevent low traffic from delaying inserts and thus committing records with expired TTLs (see Appendix A for an analysis of real-world TTL values), CODoH uses both size and time thresholds for batching: the enclave commits when either *(1)* the batch accumulates roughly B queries or *(2)* a maximum delay $T_{\max}$ has elapsed since the first pending insert.[2] Let B_{eff} denote the number of real queries included in a time-triggered commit; CODoH requires $B_{\text{eff}} \geq B_{\min}$ whenever cache hits are enabled. This yields large, well-mixed batches under high load while bounding delay under low load. When traffic is insufficient to form a safe batch (fewer than $B_{\min}$ distinct client queries within $T_{\max}$), the enclave disables cache hits and temporarily falls back to baseline ODoH, thereby avoiding small, easily isolatable cache updates.

4.4 TTLs and Replay Protection

As in baseline DNS, each cached record carries a *time-to-live (TTL)* that determines how long it remains valid. However, Intel SGX does not provide a trusted clock, since all time sources originate outside the enclave. This limitation complicates safe cache expiration and replay protection.

CODoH addresses this challenge by having the resolver include a signed timestamp with every cache insert. Inside the enclave, CODoH maintains a monotonic notion of time by tracking the largest timestamp accepted so far, denoted t_{latest}. The enclave evaluates expiration and freshness relative to this value. To defend against replay attacks—such as a malicious proxy attempting to reinsert stale records to maintain incorrect answers or create split views—the enclave rejects any cache insert with timestamp $t < t_{\text{latest}} - \delta$, where δ is a configurable tolerance (e.g., a few seconds) accounting for benign network delay or message reordering.

4.5 Restart Handling

A malicious proxy can restart the enclave to clear its state and create fresh cache windows that simplify set-difference attacks. To limit this risk, the enclave generates a new key pair (pk'_E, sk'_E) on restart and produces an attestation report that cryptographically binds the new public key pk'_E to the enclave's measurement. Publishing a new key enables accountability, since clients and the target can detect frequent restarts.

After a restart, the enclave rejects messages encrypted under the old key pk_E and returns a key error. The enclave then enters a warm-up period during which it performs enough batch insertions to refill the cache to a configurable threshold before serving cache hits. During this period, the enclave always returns a cache miss, effectively reverting to baseline ODoH.

4.6 Cache Omission Attacks

A malicious proxy can withhold resolver responses from being inserted into the cache. While this mainly harms performance and accountability, it also creates a privacy risk: by selectively omitting inserts, the proxy can keep the cache in a controlled, predictable state. For instance, during an enumeration attack, the proxy could suppress the responses for its probing queries to avoid polluting the cache with those entries and their associated cover responses, making cache-state changes easier to isolate.

To mitigate this threat, the enclave tracks queries for which it has not received matching resolver responses. If the number of outstanding queries exceeds a threshold, the enclave enters the

[1] In principle, a commit could create a burst of (ORAM, see §4.7) accesses observable to the host. Our design does not require commits to appear as a contiguous "write storm": an implementation can amortize inserts using ORAM batch-write interfaces or spread writes over subsequent fixed-rate maintenance accesses, so that epoch boundaries are not trivially exposed by instantaneous access volume.

[2] To avoid making time-triggered commits fully predictable under low traffic, the enclave may randomize the timeout (e.g., $T_{\max} + \Delta$, for small Δ drawn from a bounded distribution).

same defensive mode used during a restart warm-up, temporarily disabling cache hits and reverting to baseline ODoH behavior.

4.7 Oblivious Caching

Even if the proxy cannot read cache contents, it may attempt to infer whether two client requests correspond to the same query by monitoring which entries are accessed, how often, or in what order. Such signals could strengthen enumeration or correlation attacks by turning the cache into a query-equality oracle.

Although SGX protects the confidentiality and integrity of cached data, it does not conceal memory access patterns from a malicious host that can observe page faults or other architectural side channels. CODoH thus implements the cache inside the enclave using ORAM. ORAM addresses this gap by making memory access patterns computationally indistinguishable across different logical queries, up to leakage of the total number of accesses. Additionally, CODoH performs a fixed number of ORAM operations per lookup/insert (or uses constant-time mode) to reduce timing variation. As a result, the proxy cannot determine which logical cache entry was accessed or whether accesses repeat.

5 Security Analysis

This section argues that CODoH meets its cache-specific security goals (G1–G3) against the stronger proxy-side adversary introduced by caching. We structure the analysis around these three goals, namely: *(G1)* authorization and freshness of cached answers, *(G2)* resistance to cache-state inference (with an explicit bound for set-difference attacks as a function of batch size B and number of cover responses k), and *(G3)* resistance to query equality/profiling via cache behavior. Our arguments assume standard cryptographic security of HPKE and authenticated encryption, unforgeability of the target's signature on cache inserts, and the access-pattern indistinguishability of the ORAM construction.

5.1 Authorized and Fresh Cached Answers (G1)

Authorized cached answers. CODoH ensures that the proxy cannot cause clients to accept a cached DNS response that the target did not authorize. Concretely, the target produces a cache-insert bundle $\hat{\mathsf{R}}_{\mathsf{E}}$ encrypted to the enclave's public key and authenticated with a target signature. The enclave verifies this signature before inserting (query, response) into the cache. Under EUF-CMA unforgeability of the target's signature scheme, a malicious proxy cannot generate a new cache-insert bundle that the enclave will accept without the target's signing key. Under IND-CCA security of HPKE and the authenticated encryption protecting this ciphertext, the proxy also cannot tamper with a valid cache insert—such as modifying the embedded response—without detection during verification. Consequently, any answer served from the enclave cache must originate from a target-authorized DNS response.

Freshness and replay resistance. A malicious proxy could replay stale cache inserts to retain outdated records or create inconsistent views. CODoH prevents such replay by requiring the target to attach a signed timestamp to each cache insert and by maintaining a monotonic "latest accepted time" within the enclave. Let t_{latest} denote the largest timestamp accepted so far. The enclave rejects any insert with timestamp $t < t_{\text{latest}} - \delta$, where δ allows for minor reordering and network delay. Since the proxy cannot forge the target's signature, it cannot create fresh-looking stale records; it can only replay old inserts, which fall outside the acceptance window as t_{latest} advances. This limits replay to a bounded window of size δ and prevents indefinite stale reinsertion. If the proxy restarts the enclave and clears its state, the enclave generates a new key pair $(\mathsf{pk}'_{\mathsf{E}}, \mathsf{sk}'_{\mathsf{E}})$ on restart, thus rejecting any stale inserts encrypted under the old key.

5.2 Cache-state Indistinguishability under Active Probing (G2)

We analyze CODoH's primary cache-specific threat: a set-difference attack in which a malicious proxy attempts to isolate a victim query by comparing cache state across time. CODoH reduces the value of any cache-state snapshot by ensuring that each commit event changes cache state by many entries whose identities remain unknown to the proxy.

Model. CODoH's policy is to commit a batch to the cache only when it can form a safe batch (at least $B_{\min}$ unique client queries within $T_{\max}$). When the enclave cannot form a safe batch within $T_{\max}$, it disables cache hits and reverts to baseline ODoH; in this mode, the proxy does not obtain a cache-hit oracle. Let $B_{\text{eff}} \geq B_{\min}$ denote the number of effective queries in the batch commit.

Worst-case adversary scheduling. CODoH explicitly considers a proxy that can queue traffic and inject its own queries. In the worst case, the proxy causes a batch to contain exactly one victim query and $B_{\text{eff}} - 1$ adversary-chosen queries. The proxy knows the domains it queried for those $B_{\text{eff}} - 1$ adversary queries, but it does not know any of the k cover domains selected by the target (for either victim or adversary queries), because covers appear only inside the target-to-enclave encrypted bundle.

Let S denote the inserted-domain multiset produced by the batch commit, and let the proxy remove from S the $B_{\text{eff}} - 1$ known adversary real domains. The remaining candidate multiset S' contains: *(1)* the victim's real domain (one element), and *(2)* the $B_{\text{eff}}k$ cover domains. Ignoring accidental duplicate domains (we discuss collisions below), $|S'| = 1 + B_{\text{eff}}k$.

Identification bound under indistinguishable covers. If cover domains are indistinguishable from real queried domains from the proxy's perspective (i.e., they follow the same effective distribution once padding removes obvious size-based cues), then the victim's real domain is exchangeable with the $B_{\text{eff}}k$ cover domains in S'. In this case, no adversary can identify the victim domain from S' with probability better than random guessing:

$$\Pr[\text{proxy identifies victim domain}] \leq \frac{1}{1 + B_{\text{eff}}k} \leq \frac{1}{1 + B_{\min}k}.$$

This bound captures the core privacy–performance knob: increasing $B_{\min}$ (mixing more queries per commit) and/or increasing k (adding more covers per query) decreases the proxy's best-possible success probability even under highly adversarial scheduling.

Membership testing and false positives. A proxy may instead try to test whether the victim queried a particular target domain d^*. In the worst-case cache-delta model, the proxy can check whether

$d^* \in S'$. However, cover inserts intentionally induce false positives: even if the victim did not query d^*, covers may include d^*. If covers are sampled independently from distribution D, then the probability that d^* appears at least once among the $B_{\text{eff}}k$ covers is:

$$\Pr[d^* \text{ appears in covers}] = 1 - (1 - D(d^*))^{B_{\text{eff}}k}.$$

Thus, increasing $B_{\text{eff}}k$ increases the rate at which any specific domain appears as a cover, directly reducing the proxy's confidence in membership testing.

Collisions and effective anonymity. Domain collisions among covers reduce $|S'|$ and therefore weaken the $1/(1 + B_{\text{eff}}k)$ bound. In practice, collisions remain rare when the cover domain is large and $B_{\text{eff}}k$ stays moderate, but the system can also sample covers without replacement within a batch or enforce deduplication inside the enclave when constructing S'. Either approach increases the number of distinct candidates and strengthens the anonymity set.

Target misbehavior. Our set-difference analysis assumes that the target samples cover domains from a distribution D that approximates the system's empirical query distribution Q (e.g., with bounded likelihood ratios $Q(x)/D(x)$). This ensures that covers are indistinguishable from real inserts from the proxy's perspective. If the target instead biases covers toward a narrow subset—such as only highly popular domains—then out-of-distribution inserts become more likely to be real, weakening the privacy bound even without explicit proxy–target collusion. Although CODoH does not protect against a malicious target, deployments can mitigate accidental or covert bias by statistically auditing cover selection, for example by estimating divergence from a published distribution D; the enclave itself could perform such checks against an agreed-upon reference distribution.

If the target provides incorrect freshness metadata—such as inaccurate TTLs or timestamps—this primarily impacts correctness and cache effectiveness. Stale timestamps cause inserts to be rejected, while extreme forward jumps can prematurely age out otherwise valid entries, reducing hit rates and potentially disabling caching. These behaviors do not enable the proxy to learn plaintext queries, but they can reduce the mixing benefit of batching by suppressing inserts and thus shrink the effective anonymity set. We therefore treat correct cover selection and freshness metadata as part of the target's trusted behavior.

5.3 Query Equality and Profiling Resistance (G3)

A malicious proxy may try to infer whether two client requests correspond to the same query by observing cache behavior (e.g., hit/miss patterns, timing differences, or access-pattern leakage). CODoH addresses these channels in two ways. First, CODoH avoids exposing an explicit hit/miss signal to the proxy by ensuring that requests still trigger resolver activity and by structuring client-visible behavior so that the proxy cannot trivially distinguish cache hits from misses. Second, CODoH uses ORAM to hide cache access patterns from a malicious host: even if the proxy can monitor page faults, memory traffic, or other architectural signals outside the enclave, ORAM makes the access pattern computationally indistinguishable across different logical queries (up to leakage of the total number of accesses). Together, these mechanisms prevent the proxy from turning caching into a query-equality oracle.

6 Implementation

We implement CODoH as a set of plugins for CoreDNS [5], an extensible DNS server written in Go. CoreDNS's plugin architecture lets each protocol variant—plain DoH, ODoH, and CODoH—run as a distinct plugin configuration, enabling controlled comparisons under identical server infrastructure. The enclave component runs as a standalone binary built with EGo [47], a framework for developing Intel SGX enclaves in Go. On the client side, we extend Cloudflare's `odoh-client-go` library [15] with CODoH support.

6.1 System Architecture

CODoH deploys as three processes: a *proxy*, an *enclave*, and a *target*. The proxy and target are CoreDNS plugins (`codohproxy` and `codohtarget`); the enclave is a standalone EGo binary co-located with the proxy.

The proxy communicates with the enclave over a Unix domain socket using a compact binary IPC protocol with a narrow interface: forwarding encrypted client queries, delivering signed cache-insert bundles, and retrieving the enclave's public key. If the enclave is unreachable, the proxy degrades gracefully to baseline ODoH.

The target operates as a standard ODoH resolver. When it receives a query relayed through a CODoH proxy (indicated by the presence of the enclave's public key in the request), it additionally resolves k cover domains, assembles the real and cover responses into a signed bundle, encrypts it to the enclave's public key, and POSTs the resulting blob back to the proxy's cache-insert endpoint. This asynchronous path runs after the ODoH response has already been sent to the client, so it does not add latency to the critical path.

6.2 Cryptographic Instantiation

We instantiate the HPKE suite as DHKEM(X25519, HKDF-SHA256) with HKDF-SHA256 and AES-128-GCM, matching the mandatory-to-implement ciphersuite in RFC 9180 [6]. Cache-insert bundles are signed with Ed25519. The target signs the SHA-256 digest of the plaintext bundle before HPKE encryption, and the enclave verifies this signature after decryption, ensuring that the proxy cannot forge or tamper with cache contents.

The per-query response keys (k_e and k_t) are each 16-byte values derived via the HPKE Export interface. The enclave encrypts cached responses under k_e using AES-128-GCM with a random 12-byte nonce. On a cache miss, the enclave instead returns a dummy payload of identical size (284 bytes: 12-byte nonce, 256-byte random fill, and 16-byte authentication tag), which is indistinguishable from a real encrypted response after bucket padding.

6.3 ORAM Cache

We implement the oblivious cache using Path ORAM [46] with a bucket size of 5 and constant-time mode enabled to prevent timing side channels on individual accesses. Each cache entry is serialized into a fixed-size ORAM block containing the canonical query, the DNS response, an insertion timestamp, and a TTL.

TTL expiry is lazy: on each lookup, the enclave checks whether the entry's insertion time plus its TTL exceeds the current logical

time t_{latest}, and treats expired entries as misses. Batch insertions use the ORAM library's batch-write interface, which amortizes path reads and evictions across multiple writes within a single commit.

6.4 Cover Responses

For each query, the target samples k cover domains and resolves them alongside the real query. Cover domains are drawn from the Cisco Umbrella Top-1M list [13] using a two-tier mixture: with probability p, a domain is sampled from the popular top-N tier, and with probability $1 - p$, from the remaining long-tail tier. Cover domains are resolved in parallel over UDP. If a cover response returns `NXDOMAIN`, the enclave applies RFC 2308 [2] negative caching rules and derives the TTL from the zone's `SOA` record, capped by an enclave-defined upper bound to prevent excessively long lifetimes.

The real response and k cover responses are serialized into a single bundle, encrypted to pk_E, and signed with the target's Ed25519 key. The enclave receives and processes the bundle atomically, so individual entries are never exposed to the proxy.

6.5 Padding and Wire Format

We pad all queries and responses to fixed-size buckets to prevent length-based distinguishers. Each padded message consists of a 2-byte little-endian length prefix, the original data, and random bytes up to the bucket boundary. Based on our analysis of Cisco Umbrella response sizes (Appendix A), we pad enclave responses—both cache hits and dummy payloads—to a 2 KB bucket, which accommodates the largest HPKE-encrypted response observed among the top-10k domains (1616 bytes). For queries, both Q_T and Q_E are padded to a single 256-byte bucket.

6.6 SGX Attestation and Key Provisioning

On startup, the enclave generates a fresh HPKE key pair and obtains a remote attestation quote from the SGX hardware that binds pk_E to the enclave's measurement (`MRENCLAVE`) via the quote's user-data field. The proxy caches this public key and serves it to clients at a well-known endpoint.

Before the target will send cache-insert bundles, it must verify the enclave's identity. The target fetches the attestation quote, verifies it against Intel's attestation infrastructure, checks that the `MRSIGNER` matches the expected enclave signer, and extracts pk_E from the user-data field. Once verified, the target provisions its Ed25519 signing public key to the enclave, completing the mutual key exchange needed for authenticated cache inserts.

6.7 Client

The client extends the Cloudflare `odoh-client-go` library [15] with CODoH support. For each query, the client constructs two independent ciphertexts: Q_T via the standard ODoH encryption path, and Q_E by HPKE-encrypting the canonical query string (e.g., `"example.com"`) to the enclave's public key. The client sends Q_T as the HTTP POST body and Q_E as a Base64-encoded `X-CoDOH-Query` header, both padded to 256 bytes.

The proxy returns the enclave and target responses as tagged chunks that identify their source, and the client accepts the first valid response. If the target's response arrives first, the client still attempts to decrypt the enclave's chunk to detect key rotation (e.g., after an enclave restart).[3] Specifically, the proxy signals rotation via the `X-CoDOH-Key-Rotated` header, which alerts the client to refetch pk_E before issuing its next query.

Table 1: Benchmark configurations. Configs 1–2 are standard protocol baselines. Config 3 is a strawman caching proxy that adds an enclave-based cache to ODoH without CODoH's defense mechanisms. Configs 4 and 5 are CODoH deployments: Config 4 runs the cache as a plain process, while Config 5 is the full CODoH system with the cache inside an SGX enclave.

#	Name	Procs	Cache	Covers	Batching	SGX
1	DoH	1	—	—	—	—
2	ODoH	2	—	—	—	—
3	Cached ODoH	2	LRU	✗	✗	✓
4	CODoH-noTEE	3	ORAM	✓	✓	✗
5	CODoH	3	ORAM	✓	✓	✓

7 Evaluation

We evaluate CODoH's end-to-end latency overhead relative to standard DoH and ODoH. Our experiments answer three questions: *(1)* How does even naïve caching affect latency, and how much further overhead do CODoH's defense mechanisms (covers, ORAM, batching) impose? *(2)* What is the performance of CODoH relative to standard ODoH? *(3)* How much of CODoH's overhead is attributable to the TEE runtime versus the protocol-level defenses?

7.1 Experimental Setup

Testbed. We deploy the system across three Azure VMs in separate regions to capture realistic wide-area network latencies: the *client* in US North Central, the *proxy* in US East, and the *target* in US Central. The proxy runs on a Standard DC4s v2 (4 vCPUs, 16 GiB, 112 MiB EPC) with Intel SGX support, the target on a Standard D2s v3 (2 vCPUs, 8 GiB), and the client on a Standard D2alds v6 (2 vCPUs, 4 GiB). All VMs run Ubuntu 24.04 LTS with Go 1.25. The enclave is built and signed with EGo 1.8.1.

Upstream resolver. The target VM runs a local Unbound [30] recursive resolver on port 5353, configured with default caching. Between each workload, we flush Unbound's cache (using the command `unbound-control flush_zone .`) to ensure independent measurements across workloads.

Domain lists. We draw query domains from the Cisco Umbrella Top-1M popularity list. We pre-filter the list to retain only domains that resolve successfully (return at least one `A` record), producing a 10,000-domain list for cold and cover workloads and a 1,000-domain subset for Zipf workloads.

7.2 Configurations

We compare five configurations, summarized in Table 1. Each successive CODoH configuration adds protocol complexity, enabling us to attribute overhead to specific defense mechanisms.

[3] In this case, if both responses validate but differ, the client treats it as an error.

Config 1: *DoH.* A single CoreDNS instance serves DNS-over-HTTPS (RFC 8484), forwarding queries directly to the upstream resolver. This configuration provides the best-case latency without any oblivious encryption and serves as a performance ceiling.

Config 2: *ODoH.* A standard Oblivious DoH deployment with two CoreDNS instances: an ODoH proxy and an ODoH target. The client encrypts queries end-to-end to the target via the proxy, which cannot observe query contents. This is the privacy baseline against which we measure CODoH's overhead.

Config 3: *Cached ODoH.* A strawman caching proxy that adds an SGX-hosted LRU cache to the ODoH proxy without any of CODoH's defense mechanisms: no cover responses, no ORAM, and no batched insertions. The enclave runs as a combined proxy–cache process: it terminates the client's HTTPS connection, performs HPKE decryption to check the cache, and forwards the ODoH-encrypted query to the target. This two-process architecture (proxy and target) reflects the simplest viable caching design—before the defense mechanisms described in §4 necessitate the three-process split used in Configs 4 and 5. Comparing Configs 2 and 3 isolates the benefit of adding a cache to ODoH, while comparing Configs 3 and 5 shows the cost of CODoH's full defense stack over naïve caching.

Config 4: *CODoH-noTEE.* The full CODoH protocol with all defense mechanisms—ORAM cache (N=1024), cover responses (k=3), batched insertions (B=10—chosen to limit queuing delay given the sequential single-client workload—commit probability p=0.1), and bucketed response padding—but with the cache process running as a plain userspace process rather than inside an SGX enclave. The cache process and proxy communicate over a Unix domain socket; the proxy and target communicate over a remote TLS connection. Comparing Configs 4 and 5 cleanly isolates the performance cost of SGX hardware, since both share identical defense mechanisms and code paths.

Config 5: *CODoH.* The complete CODoH system: identical to Config 4 but with the cache process running inside an Intel SGX enclave via EGo. On startup, the enclave produces a remote attestation quote binding its HPKE public key; the target verifies this quote before provisioning its Ed25519 signing key. This is the production configuration as described in §4 and §6. Comparing Configs 4 and 5 measures the cost of TEE isolation; comparing Configs 2 and 5 gives the total overhead of CODoH over standard ODoH.

7.3 Workloads

For each configuration, we measure latency under three workloads that span the cache-performance spectrum:

Cold. 10,000 unique domains queried in rank order from the pre-filtered Umbrella top-10k list. Every query is a cache miss, requiring full upstream resolution. This workload represents the worst case for CODoH, where the cache provides no benefit and all defense overhead (cover resolution, ORAM writes, encryption) is fully exposed.

Zipf. 10,000 queries drawn from a Zipf distribution (s=1.0) over 1,000 domains. This models realistic DNS traffic, where a small number of popular domains account for most queries [27]. Under this workload, the CODoH cache absorbs repeated queries, and the cache-hit ratio determines how much latency is saved relative to ODoH.

Warm. 10,000 queries for a single domain (`google.com`). After the first query populates the cache, all subsequent queries are cache hits, measuring the pure protocol overhead of CODoH with no upstream resolution on the critical path.

7.4 Measurement Methodology

Client tool. We use a Go benchmarking client, forked from Cloudflare's `odoh-client-go` [15] with added CODoH support, that issues sequential HTTPS requests and records per-query wall-clock latency, including client-side encryption and response decryption where applicable. TLS sessions are reused across queries within each workload. The proxy maintains a persistent HTTPS connection pool to the target, so TLS sessions between proxy and target are reused across queries. For each workload, the client writes a CSV of individual query latencies and a JSON summary with percentile statistics (p50, p95, p99), mean latency, and throughput.

Isolation protocol. Between configurations, the orchestrator terminates all server processes and performs a clean restart. Between workloads within a configuration, the orchestrator flushes the upstream Unbound cache to prevent cross-workload contamination. Workloads run in order: cold, Zipf, warm. The enclave's internal cache is not flushed between workloads, so the warm workload benefits from entries inserted during prior workloads.

Sensitivity analysis. To evaluate the impact of key defense parameters, we re-run Config 5 with varied ORAM capacity ($N \in \{256, 512, 1024, 2048\}$) and cover count ($k \in \{1, 3, 5\}$), holding all other parameters at their default values. We omit the cold workload, as all queries are cache misses regardless of N or k.

7.5 Results

Table 2 summarizes the end-to-end latency for all configurations under each workload. Speedup factors (in parentheses) are relative to ODoH (Config 2).

Warm workload. On the warm workload (all cache hits), CODoH achieves 0.37× ODoH latency at the median (17.94 vs. 47.97 ms)—a 2.67× speedup. Cache hits are served locally by the enclave, eliminating the proxy-to-target round trip that dominates ODoH. Relative to DoH (8.79 ms), CODoH adds 9.15 ms—mostly the different network path (client to proxy in US East, rather than directly to the target in US Central) plus protocol overhead (HPKE, ORAM, IPC $\approx$ 0.29 ms)—while providing oblivious routing that DoH lacks entirely. CODoH's P99 (43.39 ms) is higher than its median, reflecting occasional ORAM access latency spikes; these remain well below ODoH's median and do not affect the common case.

Zipf workload. Under Zipf traffic, CODoH maintains 0.46× ODoH at the median (22.34 vs. 48.60 ms). The cache absorbs repeated queries for popular domains, and the benefit grows at the tails: CODoH's P99 is 98.47 ms (0.45× ODoH) compared to 218.53 ms for ODoH, where every query—including repeats—incurs full upstream resolution. Compared to Cached ODoH, which lacks CODoH's defense mechanisms, CODoH has a slightly higher median (22.34 vs.

Table 2: Latency comparison across configurations (ms). C3 = Cached ODoH, C4 = CODoH-noTEE, C5 = CODoH. Speedup factors are relative to ODoH. Each workload issues 10,000 queries.

	Cold			Zipf			Warm		
Config	p50	p95	p99	p50	p95	p99	p50	p95	p99
DoH	66.72	346.54	781.33	14.31	55.67	163.08	8.79	9.07	10.09
ODoH	107.45	386.05	806.16	48.60	103.89	218.53	47.97	48.86	51.12
C3	108.99 (1.01×)	386.76 (1.00×)	763.14 (0.95×)	17.32 (0.36×)	92.29 (0.89×)	198.00 (0.91×)	21.54 (0.45×)	22.05 (0.45×)	22.57 (0.44×)
C4	94.28 (0.88×)	350.64 (0.91×)	720.28 (0.89×)	21.71 (0.45×)	48.32 (0.47×)	81.29 (0.37×)	17.39 (0.36×)	22.30 (0.46×)	41.52 (0.81×)
C5	90.41 (0.84×)	351.28 (0.91×)	710.33 (0.88×)	22.34 (0.46×)	59.41 (0.57×)	98.47 (0.45×)	17.94 (0.37×)	20.93 (0.43×)	43.39 (0.85×)

17.32 ms) due to ORAM overhead, but delivers tighter tails (P95: 59.41 vs. 92.29 ms; P99: 98.47 vs. 198.00 ms).

Cold workload. Under the cold workload (all cache misses), upstream recursive resolution dominates and all configurations converge: medians range from 66.72–108.99 ms, P95 from 346.54–386.76 ms, and P99 from 710.33–806.16 ms. CODoH achieves 0.84× ODoH at the median and 0.88–0.91× at the tails, confirming that its defense mechanisms impose no meaningful cost on cache-miss queries already bottlenecked by upstream resolution.

SGX overhead. Comparing CODoH-noTEE and CODoH isolates the cost of TEE execution. At the median, SGX overhead is under 1 ms on cache-hit workloads: 0.55 ms on warm (17.94 vs. 17.39 ms) and 0.63 ms on Zipf (22.34 vs. 21.71 ms). On cold, the two configurations are within noise (90.41 vs. 94.28 ms), as upstream variance dwarfs any TEE cost. Zipf tail latencies show more separation: P95 differs by 11.09 ms (59.41 vs. 48.32 ms) and P99 by 17.18 ms (98.47 vs. 81.29 ms), likely due to ORAM access variance under SGX memory encryption. On warm and cold, P95 differences are within measurement noise (≤1.37 ms). Overall, SGX adds modest overhead at the median (< 1 ms on cache-hit workloads) that is dominated by network and resolution variance in practice.

Parameter sensitivity. We re-run Config 5 with varied ORAM capacity and cover count to evaluate the impact of key defense parameters (full results in Table 7, Appendix B). *ORAM capacity* (N) primarily affects Zipf performance: increasing N from 256 to 2048 raises the cache-hit rate from 40.6% to 75.7% and reduces mean latency from 44.2 to 27.8 ms, as a larger ORAM accommodates more of the Zipf tail. However, per-access ORAM cost also rises: warm-workload hit p50 increases from 18.1 ms at N=256 to 21.9 ms at N=2048, consistent with the microbenchmark EPC paging pressure observed at N=2048 (Table 4). At N=1024 (our default), the warm hit rate is 98.9% with a hit p50 of 18.0 ms, balancing capacity against per-access cost. *Cover count* (k) has a more modest effect. Under Zipf, hit rates are stable across $k \in \{1, 3, 5\}$ (65.0%, 64.4%, 62.6%) because Zipf's long tail of unique domains cannot be anticipated by cover insertions. On warm traffic, k=1 yields a lower hit rate (96.6% vs. 98.9% at k=3), while k=5 adds latency (hit p50: 22.4 vs. 18.0 ms) without improving hit rate. We use k=3 as the default, which achieves the highest warm hit rate with moderate overhead.

7.6 Microbenchmarks

To explain the macrobenchmark results, we measure the latency of individual operations on the critical path of a CODoH query. All microbenchmarks run on the proxy VM and report the median of 10,000 iterations.

Table 3: Per-operation cryptographic latency (μs). All operations use the HPKE suite DHKEM(X25519, HKDF-SHA256) with AES-128-GCM, matching the CODoH deployment.

Operation	Plain (μs)	SGX (μs)	Overhead
HPKE Encrypt (query)	57.0	67.7	1.19×
HPKE Decrypt (query)	36.8	41.8	1.14×
AES-GCM Seal (resp.)	0.87	3.46	3.98×
AES-GCM Open (resp.)	0.64	0.75	1.17×
Ed25519 Sign	21.5	21.0	0.98×
Ed25519 Verify	49.3	50.6	1.03×

Table 4: Cache read (Get) latency vs. capacity N (μs). LRU is $O(1)$; ORAM is $O(\log N)$ per access.

N	LRU	ORAM (Plain)	ORAM (SGX)	Overhead
256	0.11	108	223	2.1×
512	0.11	103	222	2.2×
1024	0.12	99	228	2.3×
2048	0.13	107	377	3.5×

Cryptographic operations. Table 3 reports the per-operation latency of the cryptographic primitives used in CODoH.

HPKE dominates the per-query crypto cost: encrypting a query (KEM setup + AEAD seal) takes 68 μs under SGX, while the symmetric AES-GCM operations are sub-microsecond outside the enclave. Under SGX, AES-GCM Seal rises to 3.5 μs (4×), but this is on a sub-microsecond base and adds only 2.6 μs in absolute terms. Ed25519 signing and verification show near-zero SGX overhead (≤1.03×), as these are CPU-bound and do not pressure enclave memory.

Cache access latency. Table 4 compares the per-access read (Get) latency of the ORAM cache against a plain LRU cache across capacities $N \in \{256, 512, 1024, 2048\}$. LRU access is sub-microsecond and constant regardless of N, while ORAM access ranges from 99–108 μs outside SGX. Inside the enclave, ORAM latency roughly doubles for $N \leq 1024$ but jumps to 3.5× at N=2048, suggesting EPC paging pressure at that capacity. For ORAM, Put latencies are within 10% of Get; LRU Put is also sub-microsecond, though up to 33% higher than Get at larger N.

Table 5: IPC round-trip latency (μs).

IPC path	Plain (μs)	SGX (μs)	Overhead
Health check (baseline)	9.0	121	13.4×
Cache-hit lookup	86	292	3.4×

Table 6: Mean padding overhead per message (Umbrella top-10k, single-bucket scheme). Waste is the mean unused bytes per padded message.

Message	Bucket (B)	Size (B)	Waste (B)	Overhead
Q_E (to enclave)	256	108	147	57.5%
Q_T (to target)	256	95	160	62.6%
Response (to client)	2048	274	1773	86.6%

IPC round-trip. Table 5 reports the round-trip latency of the Unix domain socket IPC channel between the proxy and enclave processes. We measure two paths: a health check (minimal processing, isolating serialization and socket overhead) and a cache-hit lookup (HPKE decryption, ORAM read, response encryption).

Latency decomposition. The IPC cache-hit latency under SGX (292 μs $\approx$ 0.29 ms) accounts for HPKE decryption, ORAM lookup, and response encryption in a single round trip. The warm-workload macrobenchmark median for CODoH is 17.94 ms, so the SGX cache (including HPKE, ORAM, and encryption) contributes only ~1.6% of end-to-end latency; the remaining ~17.65 ms is the client-to-proxy round trip (network RTT, TLS, and HTTP processing). This confirms that CODoH's defense mechanisms—ORAM, HPKE, padding—are not the bottleneck; network latency dominates. The 121 μs health-check baseline isolates pure IPC overhead (serialization, Unix socket, deserialization), showing that the cryptographic and ORAM work within a cache-hit adds ~171 μs on top.

Padding overhead. The preceding microbenchmarks measure latency; we now quantify the bandwidth cost of bucketed padding (§4). We resolve the Umbrella top-10k domains and measure the wire size of each query and response before and after padding to a single 256-byte (query) or 2048-byte (response) bucket (Table 6). Query overhead is moderate: the mean Q_E ciphertext is 108 B, yielding 57.5% waste in a 256 B bucket. Response overhead is higher: the mean encrypted response is 274 B, yielding 86.6% waste in a 2048 B bucket. This is the cost of maximum privacy—both single-bucket schemes reduce size entropy to zero bits, ensuring that message lengths reveal nothing about query content.

7.7 Page Load Benchmarks

In addition to the macro and microbenchmarks above, we also evaluate CODoH's impact on real-world page load times. To evaluate page load time, we use Playwright [42] to automate browser interactions across a set of popular websites, and dnscrypt-proxy [18] as the client interface for the ODoH and CODoH protocols. Our testbed mirrors the Azure VM setup used in the macrobenchmarks: the client VM runs the benchmark script, Playwright, and dnscrypt-proxy, while additional VMs are configured as proxies and targets as required by each protocol.

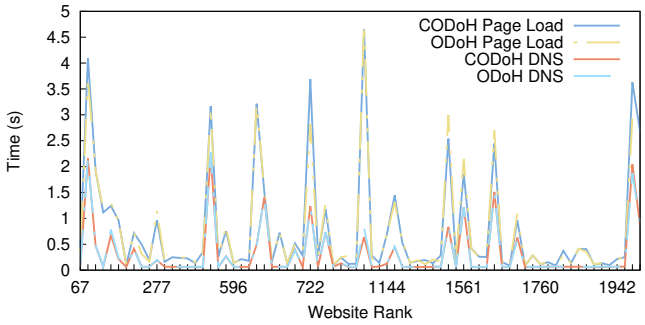

Figure 2: Comparison of average Page Load and DNS fetch times between CODoH and ODoH.

For our dataset, we randomly sample 100 sites from the top 2,000 successfully resolving domains. To account for network variability, we load each page five times in a randomized order. After discarding the highest and lowest latency values for each site, we report the average page load time under both protocols.

As summarized in Figure 2, CODoH and ODoH perform identically for simple websites requiring fewer distinct domain resolutions. However, for websites necessitating a larger number of domain lookups, the reduced DNS resolution time in CODoH directly translates to faster overall page load times compared to ODoH. For example, the page load time for `conviva.com` drops from 3.0 s under ODoH to 2.5 s under CODoH, while `opera.com` sees a similar reduction from 1.3 s to 1.18 s.

8 Conclusion

In this paper, we presented CODoH, a cacheable extension to Oblivious DNS-over-HTTPS that restores proxy-side caching to improve performance while preserving ODoH's separation between client identity and query contents. CODoH builds on prior DNS-privacy work by combining cover traffic and batching with a TEE-protected, ORAM-backed cache to mitigate cache-induced inference without incurring the latency of mix-based designs. To support further research, we will be making our code publicly available.

Acknowledgments

The authors used ChatGPT-5.2 to revise the text in this paper to correct typos, grammatical errors, and awkward phrasing.

References

[1] 2025. DNS over Tor. https://developers.cloudflare.com/1.1.1.1/other-ways-to-use-1.1.1.1/dns-over-tor/ Accessed: 2026-01-26.
[2] Mark P. Andrews. 1998. Negative Caching of DNS Queries (DNS NCACHE). RFC 2308. https://www.rfc-editor.org/info/rfc2308
[3] Apple iCloud Private Relay 2023. About iCloud Private Relay. https://support.apple.com/en-us/102602. Accessed: 2026-02-24.
[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*.
[5] The CoreDNS Authors. 2025. CoreDNS: DNS and Service Discovery. https://coredns.io. Accessed: 2025-07-25.
[6] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. 2022. Hybrid Public Key Encryption. RFC 9180. https://www.rfc-editor.org/info/rfc9180

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] 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*.

[9] 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)*.

[10] Sergio Castillo-Perez and Joaquin Garcia-Alfaro. 2008. Anonymous Resolution of DNS Queries. In *On the Move to Meaningful Internet Systems (OTM)*.

[11] David L. Chaum. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM* 24, 2 (feb 1981).

[12] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *Symposium on Foundations of Computer Science (FOCS)*.

[13] Cisco. 2026. Umbrella Popularity List. http://s3-us-west-1.amazonaws.com/umbrella-static/top-1m.csv.zip. Accessed: 2026-02-28.

[14] Cloud Native Computing Foundation 2026. Cloud Native Computing Foundation. https://cncf.io. Accessed: 2026-02-24.

[15] Cloudflare. 2021. odoh-client-go. https://github.com/cloudflare/odoh-client-go.

[16] Cloudflare Oblivious DNS over HTTPS 2025. Oblivious DNS over HTTPS. https://developers.cloudflare.com/1.1.1.1/encryption/oblivious-dns-over-https/. Accessed: 2026-02-24.

[17] Wouter B. de Vries, Roland van Rijswijk-Deij, Pieter-Tjerk de Boer, and Aiko Pras. 2020. Passive Observations of a Large DNS Service: 2.5 Years in the Life of Google. *IEEE Transactions on Network and Service Management* 17, 1 (2020).

[18] DNSCrypt. 2026. DNSCrypt. https://dnscrypt.info/. Accessed: 2026-02-28.

[19] Hannes Federrath, Karl-Peter Fuchs, Dominik Herrmann, and Christopher Piosecny. 2011. Privacy-Preserving DNS: Analysis of Broadcast, Range Queries and Mix-based Protection Methods. In *European Symposium on Research in Computer Security (ESORICS)*.

[20] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*.

[21] Stephen Herwig, Christina Garman, and Dave Levin. 2020. Achieving Keyless CDNs with Conclaves. In *USENIX Security Symposium*.

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

[23] Paul E. Hoffman. 2023. DNS Security Extensions (DNSSEC). RFC 9364. https://www.rfc-editor.org/info/rfc9364

[24] 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

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

[26] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. 2002. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on Networking* 10, 5 (2002).

[27] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. 2002. DNS Performance and the Effectiveness of Caching. In *IEEE/ACM Transactions on Networking*, Vol. 10. IEEE, 589–603.

[28] Eric Kinnear, Patrick McManus, Tommy Pauly, Tanya Verma, and Christopher A. Wood. 2022. Oblivious DNS over HTTPS. RFC 9230. https://www.rfc-editor.org/info/rfc9230

[29] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*.

[30] NLnet Labs. 2026. Unbound. https://www.nlnetlabs.nl/projects/unbound/about/. Accessed: 2026-02-28.

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

[32] 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*.

[33] 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*.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[35] 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)*.

[36] Samir Jordan Menon and David J. Wu. 2022. Spiral: Fast, High-Rate Single-Server PIR via FHE Composition. In *IEEE Symposium on Security and Privacy*.

[37] Paul Mockapetris. 1987. Domain Names - Concepts and Facilities. RFC 1034. https://www.rfc-editor.org/info/rfc1034

[38] Paul Mockapetris. 1987. Domain Names - Implementation and Specification. RFC 1035. https://www.rfc-editor.org/info/rfc1035

[39] Alec Muffett. 2020. DoHoT: making practical use of DNS over HTTPS over Tor. https://medium.com/@alecmuffett/dohot-making-practical-use-of-dns-over-https-over-tor-ef58d04ca06a Accessed: 2026-01-26.

[40] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*.

[41] Yoshimichi Nakatsua, Andrew Paverd, and Gene Tsudik. 2019. PDoT: Private DNS-over-TLS with TEE Support. In *Annual Computer Security Applications Conference (ACSAC)*.

[42] Playwright. 2026. Playwright. https://playwright.dev/. Accessed: 2026-02-28.

[43] Mohammad Amir Salari, Abhinav Kumar, Federico Rinaudi, Reza Tourani, Alessio Sacco, and Flavio Esposito. 2025. Privacy Analysis of Oblivious DNS over HTTPS: A Website Fingerprinting Study. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[44] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. 2020. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *ACM SIGCOMM*.

[45] Sudheesh Singanamalla, Suphanat Chunhapanya, Jonathan Hoyland, Marek Vavruša, Tanya Verma, Peter Wu, Marwan Fayed, Kurtis Heimerl, Nick Sullivan, and Christopher Wood. 2021. Oblivious DNS over HTTPS (ODoH): A Practical Privacy Enhancement to DNS. In *Privacy Enhancing Technologies Symposium (PETS)*.

[46] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM Conference on Computer and Communications Security (CCS)*.

[47] Edgeless Systems. 2025. Welcome to EGo. https://docs.edgeless.systems/ego.

[48] Martino Trevisan, Idilio Drago, Paul Schmitt, and Francesco Bronzino. 2023. Measuring the Performance of iCloud Private Relay. In *Passive and Active Measurement Conference (PAM)*.

[49] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*.

[50] 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*.

[51] 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*.

[52] Yunming Xiao, Chenkai Weng, Ruijie Yu, Peizhi Liu, Mattero Varvello, and Aleksandar Kuzmanovic. 2023. Demo: PDNS: A Fully Privacy-Preserving DNS. In *ACM SIGCOMM*.

[53] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Analysis of Privacy Disclosure in DNS Query. In *International Conference on Multimedia and Ubiquitous Engineering (MUE)*.

[54] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Two-Servers PIR Based DNS Query Scheme with Privacy-Preserving. In *Internaional Conferences on Intelligent Pervasive Computing (IPC)*.

A Parameter Selection

Response padding. To motivate our choice of padding bucket sizes, we query the top 700k domains of the Cisco Umbrella Top-1M for `A` records and record the size of the response. Typically, this response includes the question, one or many `A` records, and sometimes a chain of `CNAME` records. Figure 3 shows the sizes of these responses as compared to possible bucket sizes of 1 KB, 2 KB, and 4 KB. Notably, only 53 of these responses exceed 1 KB, and only 7 exceed 2 KB.

TTLs and batch interval. To estimate how often batching could delay inserts past their TTL, we measured TTLs for `A` responses for the top 150k domains in the Cisco Umbrella Top-1M. Because recursive resolvers return decremented TTLs, we collect authoritative TTLs by first querying `NS` records via a recursive resolver and then querying the corresponding authoritative nameservers

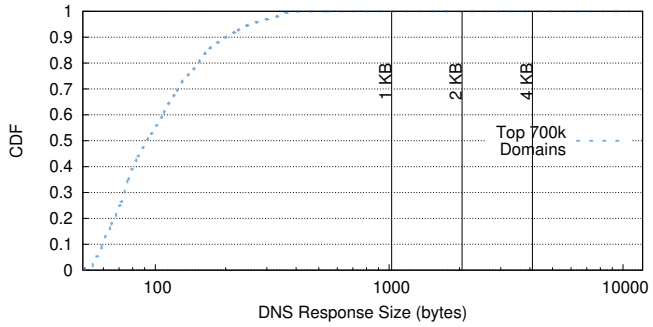

Figure 3: CDF of sizes of responses to `A` requests for Cisco Umbrella's top 700k domains (log scale).

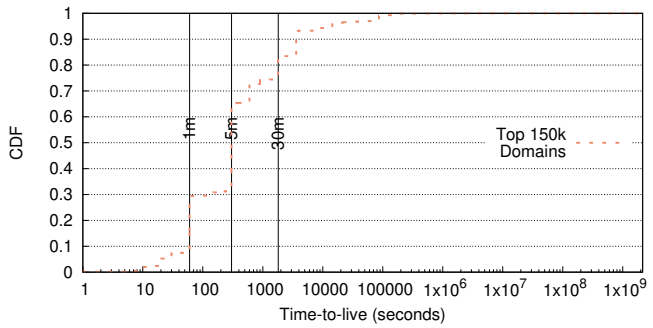

Figure 4: CDF of TTLs for `A` or `CNAME` records for Cisco Umbrella's top 150k domains (log scale).

for the desired record. Figure 4 plots the resulting TTL CDF, which is highly quantized with common values at 1, 5, and 30 minutes, and at 1–2 hours. These measurements suggest that the commit delay should be well below the smallest common TTL (1 minute); in practice, this means choosing B and $T_{\max}$ so that batches typically commit within tens of seconds at the expected query rate.

B Sensitivity Analysis Details

Table 7 reports the full latency distributions for the ORAM capacity and cover count sweeps discussed in §7. Each row corresponds to a Config 5 deployment with the indicated parameter varied; all other parameters are held at their defaults (N=1024, k=3, B=10, p=0.1). Each workload issues 10,000 sequential queries.

Table 7: Sensitivity analysis: latency (ms) under varied ORAM capacity N and cover count k. Default configuration is N=1024, k=3 (bold). Hit/miss latencies are reported separately; "Rate" is the cache-hit percentage.

			Cache Hits				Cache Misses			
Sweep	Workload	Rate (%)	p50	p95	p99	mean	p50	p95	p99	mean
ORAM capacity sweep (k=3 fixed)										
N=256	Warm	97.3	18.1	22.5	23.4	18.9	47.0	54.3	57.6	49.4
	Zipf	40.6	21.8	23.0	24.8	20.8	56.2	77.1	159.7	60.1
N=512	Warm	98.0	17.8	22.5	23.3	18.8	55.2	56.5	58.3	55.8
	Zipf	52.6	21.9	23.0	24.5	21.7	48.9	71.9	149.2	52.6
N=1024	**Warm**	**98.9**	**18.0**	**26.9**	**27.7**	**20.7**	**50.0**	**50.8**	**51.4**	**52.0**
	Zipf	**64.4**	**21.5**	**22.7**	**23.9**	**20.3**	**45.6**	**73.8**	**159.5**	**51.9**
N=2048	Warm	98.0	21.9	22.7	23.8	21.4	59.0	59.9	65.1	61.8
	Zipf	75.7	18.1	22.3	24.7	19.2	48.3	75.4	179.8	54.8
Cover count sweep (N=1024 fixed)										
k=1	Warm	96.6	21.7	24.9	26.1	21.5	48.6	49.2	50.0	49.6
	Zipf	65.0	18.1	22.4	24.7	19.1	48.4	89.2	172.9	54.9
k=3	**Warm**	**98.9**	**18.0**	**26.9**	**27.7**	**20.7**	**50.0**	**50.8**	**51.4**	**52.0**
	Zipf	**64.4**	**21.5**	**22.7**	**23.9**	**20.3**	**45.6**	**73.8**	**159.5**	**51.9**
k=5	Warm	98.6	22.4	26.6	27.7	23.5	52.9	57.1	57.8	55.1
	Zipf	62.6	18.1	22.3	23.1	19.0	48.2	55.4	126.0	49.9