When API Keys Leak: Securing AI Services with Post-Quantum Proof-of-Possession

Sunwoo Lee, Hyuk Lim, and Seunghyun Yoon
Korea Institute of Energy Technology (KENTECH), Republic of Korea
{sunwoolee, hlim, syoon}@kentech.ac.kr

***Abstract*—API keys remain the de facto authentication mechanism for AI services, yet modern software supply chains routinely expose them through container images, build artifacts, and automation pipelines. In AI platforms, a single leaked key often acts as a high-privilege machine identity enabling access to model inference, retrieval pipelines, and tool integrations, turning credential exposure into an enterprise-scale security incident. We propose a leak-resilient authentication architecture that preserves existing provider APIs while preventing unauthorized use under realistic client-side key leakage. Our design enforces post-quantum proof-of-possession at an organizational gateway and combines workload identity, KMS-backed non-exportable signing, DPoP-bound OAuth tokens, and gateway-side verification before provider API key injection. The architecture removes provider API keys and private signing keys from leak-prone client artifacts and breaks the direct path from artifact leakage to provider-side abuse. We describe the end-to-end system design and an evaluation methodology based on realistic key leakage and replay scenarios in AI service settings.**

***Index Terms*—API Key Leakage, AI Service Security, Post-quantum cryptography, Key Management System, ML-DSA.**

I. Introduction

API keys remain the de facto authentication mechanism for AI services because they are simple to deploy and widely supported. At the same time, modern software supply chains frequently expose these keys through container images, CI/CD pipelines, and configuration artifacts [1], [2]. When an API key is used as a bearer credential, disclosure allows any party holding the string to replay requests until the key is revoked. The impact is especially severe in contemporary AI platforms. A single key often functions as a high-privilege machine identity that authorizes not only model inference but also retrieval-augmented generation (RAG) resources, tool invocations, and internal service integrations. As a result, key exposure can lead to unauthorized data access, tool misuse, and cost-bearing abuse even without exploiting software vulnerabilities. Preventive controls such as secret scanning and policy enforcement help, but they do not eliminate supply-chain leakage in practice, so a robust design must remain secure even when client-side artifacts are exposed.

This paper proposes a leak-resilient authentication architecture for AI services that preserves provider compatibility while preventing unauthorized use under realistic key leakage scenarios. The design introduces an organizational gateway that mediates access to external AI providers, and the provider API key is stored and used only at the gateway. Provider-facing interfaces remain unchanged and continue to accept conventional API-key-based requests, but client-side artifacts do not embed provider API keys. Instead, each invocation to the gateway includes a proof-of-possession that demonstrates control over a signing key associated with the requesting service and the request context, and includes freshness information to prevent replay. The gateway verifies the proof and forwards the request only after successful validation, using the provider API key on behalf of the client. This breaks the direct path from artifact leakage to provider-side abuse while maintaining operational compatibility with existing AI providers.

We instantiate proof-of-possession using post-quantum digital signatures to provide long-term robustness and to align with emerging standards. Our implementation builds on OAuth 2.0 [3] and the Demonstrating Proof-of-Possession (DPoP) mechanism [4], and replaces classical signatures with NIST-standard post-quantum signatures (ML-DSA) [5]. In deployments that require stronger key isolation, the signing key can be protected by hardened key management such as key management system (KMS) so that private key material is not embedded in container images and can be centrally rotated and audited. Our contributions are summarized as follows:

- We characterize API key leakage as a first-class threat in AI service deployments and analyze its amplified impact due to AI-specific integrations such as RAG pipelines and tool invocations.
- We design a deployable, gateway-mediated post-quantum proof-of-possession architecture that preserves legacy API-key-based AI provider interfaces while preventing unauthorized use under client-side key leakage.
- We implement and evaluate the approach under realistic key leakage and replay scenarios, quantifying security benefits and deployment overhead in an AI service setting.

II. Proposed Framework

A. Problem: Bearer Token Vulnerabilities

Many AI services rely on bearer-style API keys in `Authorization` headers. This practice exhibits two weaknesses: possession of the key string suffices for authentication, allowing any party with a leaked key to replay requests until revocation; and API keys are often long-lived and broadly scoped, amplifying exposure impact.

Leakage frequently occurs through software supply chains. When API keys are embedded in container images via con-

figuration files or environment variables, publishing the image to a public registry discloses replayable credentials. Figure 1 illustrates this attack path: an adversary obtains a published image, extracts the embedded key through image inspection, and replays API requests to abuse the AI service. Internet-scale measurements confirm that credential leakage in container images is widespread [1].

Trust model. Our design confines ML-DSA private keys to a KMS boundary, avoiding embedding in container images or client artifacts. Clients authenticate using organization-issued workload identities and delegate signing to the KMS via remote APIs. The gateway and authorization server validate proof-of-possession using registered public keys with freshness and request-binding checks, while private keys remain non-exportable and centrally manageable.

Threat model. We focus on supply-chain leakage where adversaries extract embedded provider API keys from published container images, and replay-oriented misuse where stolen bearer credentials are reused from arbitrary locations. We assume TLS between system components. Attackers may reuse stolen tokens or replay observed requests from logs or compromised endpoints, but cannot forge ML-DSA signatures without accessing private keys protected by the KMS boundary. Compromise of the authorization server, KMS infrastructure, or workload identity roots is outside our scope.

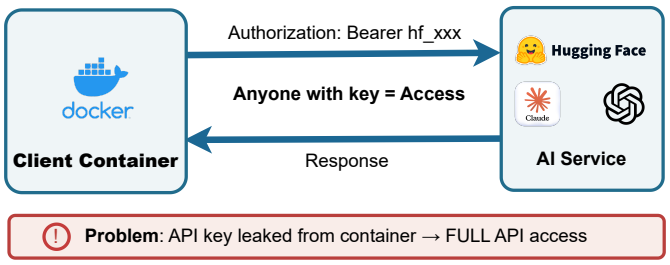

Fig. 1: Problem - Bearer tokens leak from container images

B. Solution: KMS-based Signing with Workload Identity

We propose a gateway-mediated architecture that enforces proof-of-possession for all client invocations while preserving unmodified AI provider interfaces. Workload identity authorizes bootstrap and signing privileges, and a KMS-backed signing service ensures that private key material remains non-exportable and is never embedded in container images or client-side artifacts.

Architecture components. The design consists of four components: *Identity Issuer*, *KMS*, *Authorization Server*, and *Gateway*. The *Identity Issuer* issues short-lived workload credentials (ML-DSA signed JWTs) that are used only to authenticate to internal control-plane services. The *KMS* verifies these credentials via ML-DSA signature verification, generates non-exportable ML-DSA keypairs, and provides remote signing APIs; clients obtain public keys and opaque key handles, while private keys remain confined to the KMS boundary. The *Authorization Server* verifies DPoP proofs and issues DPoP-bound OAuth 2.0 access tokens, binding tokens to the client public key via RFC 7800 confirmation claims (`cnf.jkt`). The *Gateway* verifies ML-DSA DPoP proofs with freshness and request binding and, upon successful verification, injects the provider API key and forwards requests to external AI providers.

Security properties. The architecture removes provider API keys from leak-prone client artifacts and prevents their direct reuse by requiring valid proof-of-possession at the gateway. Private ML-DSA signing keys are non-exportable and confined to the KMS boundary. Workload credentials are short-lived and distinct from PoP signing keys, so they cannot be used to generate DPoP proofs. Each request includes a fresh nonce and request-binding fields to prevent replay and cross-endpoint misuse. We instantiate PoP using NIST-standard ML-DSA (FIPS 204), enabling post-quantum-resistant authentication throughout the system.

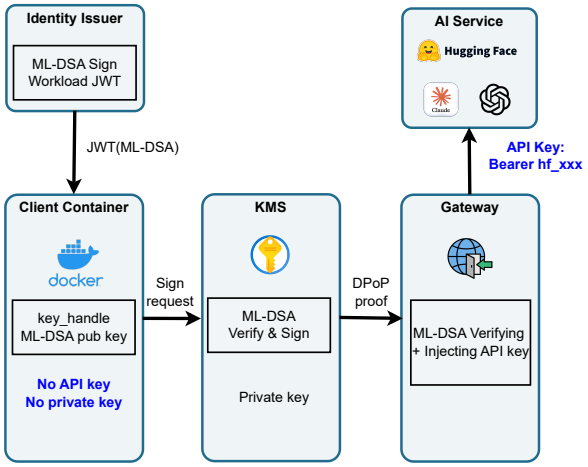

Fig. 2: Proposed multi-component authentication architecture with KMS-based signing and gateway-enforced PoP

C. Protocol Instantiation

We implement our design using OAuth 2.0 [3] with DPoP request proofs [4]. The architecture comprises an Identity Issuer for workload authentication, a KMS-backed signing service that manages non-exportable ML-DSA keys, an Authorization Server that issues DPoP-bound access tokens, and a Gateway that verifies DPoP proofs and injects provider API keys to access unmodified external AI providers. Tokens are bound to the client public key using RFC 7800 confirmation claims (`cnf.jkt`) [6].

D. Protocol Design

Protocol overview. The protocol proceeds in three phases: Phase 0 for workload bootstrap and KMS key registration, Phase 1 for DPoP-bound token acquisition, and Phase 2 for gateway-mediated API requests with fresh DPoP proofs.

Notation. Table I summarizes the entities, identifiers, cryptographic materials, and token/proof fields used in the protocol description. We use `jti` as the per-proof nonce for replay detection.

TABLE I: Notation used in the protocol description

Symbol/Term	Meaning
Entities	
$\mathcal{C}$	Client (service workload)
$\mathcal{I}$	Identity Issuer
$\mathcal{K}$	Key Management System (KMS)
$\mathcal{A}$	Authorization Server
$\mathcal{G}$	Gateway
$\mathcal{P}$	External AI Provider
Workload identity and key material	
`JWT_w`	Short-lived workload identity JWT issued by $\mathcal{I}$
`workload_id`	Workload identity asserted in `JWT_w`
`key_handle`	Opaque handle referencing a non-exportable ML-DSA private key in $\mathcal{K}$
$(pk_{\text{dsa}}, sk_{\text{dsa}})$	PoP public/private keypair (sk_{dsa} stays inside $\mathcal{K}$)
`jkt`	SHA-256 thumbprint of pk_{dsa} used for token/proof binding
Tokens and proofs	
`client_id`	OAuth 2.0 client identifier registered at $\mathcal{A}$
T	DPoP-bound access token (contains `cnf.jkt`)
P	DPoP proof JWT (signed via KMS)
DPoP claims	
`htm`, `htu`	HTTP method and URI bound to P
`iat`, `jti`	Issued-at time and nonce (replay key)
`ath`	Token hash claim: $\text{SHA256}(T)$
Provider credential	
`API_key`	Provider API key stored and used only at $\mathcal{G}$

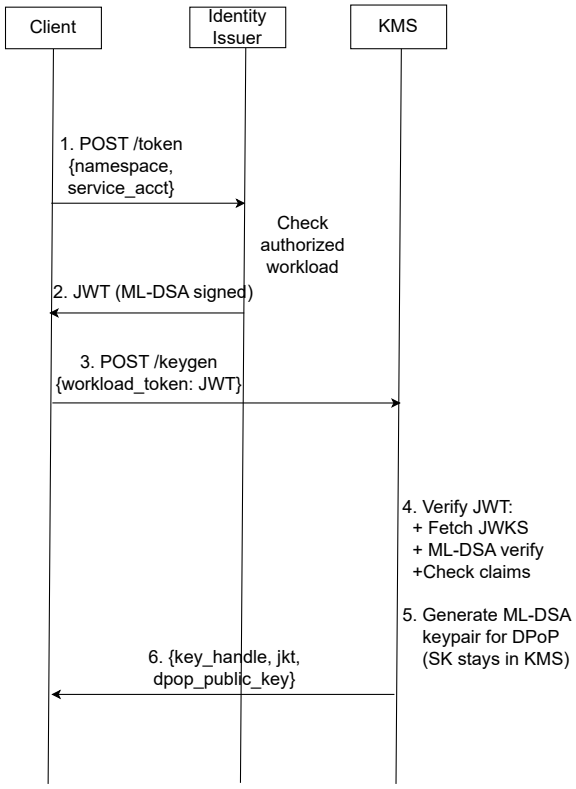

Fig. 3: Phase 0 - Bootstrap with Workload Identity

Phase 0 - Workload Identity and KMS Registration. Figure 3 illustrates the initial registration phase in which containers obtain delegated signing capability without receiving private keys.

A container requests a workload identity JWT from the Identity Issuer using $(\text{namespace}, \text{service\_acct})$ credentials, and the Identity Issuer verifies the authorization policy and issues a workload identity JWT (JWT_w) signed with ML-DSA-44. The container sends JWT_w to the KMS `/keygen` endpoint to request key registration, and the KMS fetches the Identity Issuer's JWKS, verifies the ML-DSA-44 signature, validates claims, and checks the `kms:keygen` permission. The KMS generates an ML-DSA-44 keypair $(pk_{\text{dsa}}, sk_{\text{dsa}})$ with sk_{dsa} stored exclusively in KMS, associates the generated `key_handle` with the corresponding `workload_id`, computes $\texttt{jkt} = \text{SHA256}(pk_{\text{dsa}})$, and returns $(\texttt{key\_handle}, \texttt{jkt}, pk_{\text{dsa}})$ to the container. The container subsequently registers $(\text{JWT}_w, \texttt{jkt}, pk_{\text{dsa}})$ with the Authorization Server and obtains a `client_id`. The Gateway is configured to recognize the binding between `client_id` and `cnf.jkt` so that subsequent DPoP proofs can be verified and provider API keys can be injected only after successful validation. This ensures containers never possess provider API keys or private signing keys, and only hold public information and opaque handles required for delegated signing.

Phase 1 - Token Acquisition. Figure 4 illustrates the token acquisition flow with KMS signing delegation.

The client (service workload) creates a DPoP JWT header (e.g., $\{\text{typ} : \text{“dpop+jwt”}, \text{alg} : \text{“ML-DSA-44”}\}$) and claims $\{\text{jti}, \text{htm}, \text{htu}, \text{iat}\}$ for the token endpoint, then sends a signing request $(\texttt{key\_handle}, \text{payload})$ to the KMS `/sign` endpoint. The KMS validates the previously issued workload identity JWT, including signature and expiration, verifies `key_handle` ownership, and signs the payload with ML-DSA-44 using the KMS-resident private key; the KMS then returns the ML-DSA-44 signature. The client sends the DPoP JWT and JWT_w to the Authorization Server `/token` endpoint, using JWT_w for workload authentication and `client_id` for client identification. The Authorization Server verifies JWT_w using the Identity Issuer public keys and checks that JWT_w corresponds to the registered `client_id` and `cnf.jkt` binding. The Authorization Server issues an access token with `cnf.jkt` binding to the client's public key and sets `token_type`=“DPoP”.

Phase 2 - API Request. Figure 5 illustrates the API request flow. The client (service workload) creates fresh DPoP JWT claims including $\texttt{ath} = \text{SHA256}(\text{access\_token})$ to bind the proof to the specific access token and sends a signing request with $(\texttt{key\_handle}, \text{payload})$ to the KMS `/sign` endpoint. The KMS validates the previously issued workload identity JWT, verifies the `kms:sign` permission, and returns an ML-DSA-44 signature. The client constructs the request to the Gateway with an `Authorization: DPoP <token>` header containing the DPoP-bound access token and a `DPoP:`

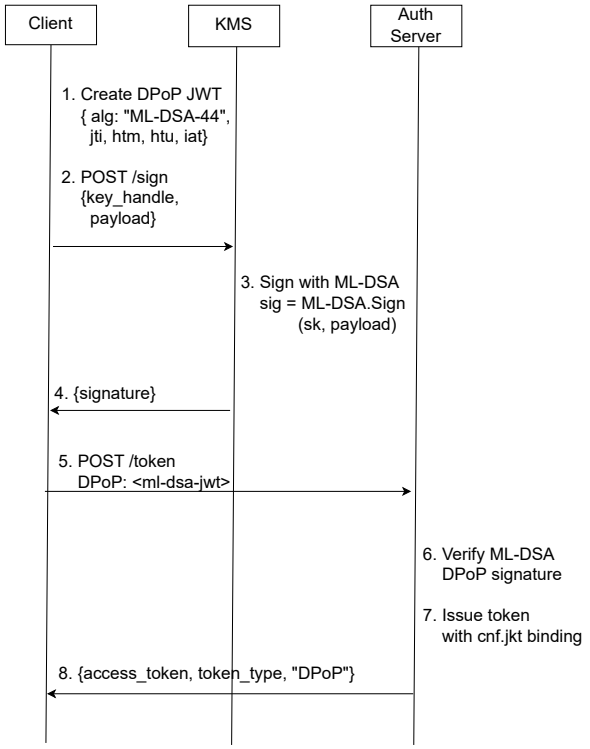

Fig. 4: Phase 1 - Token Acquisition with KMS Signing

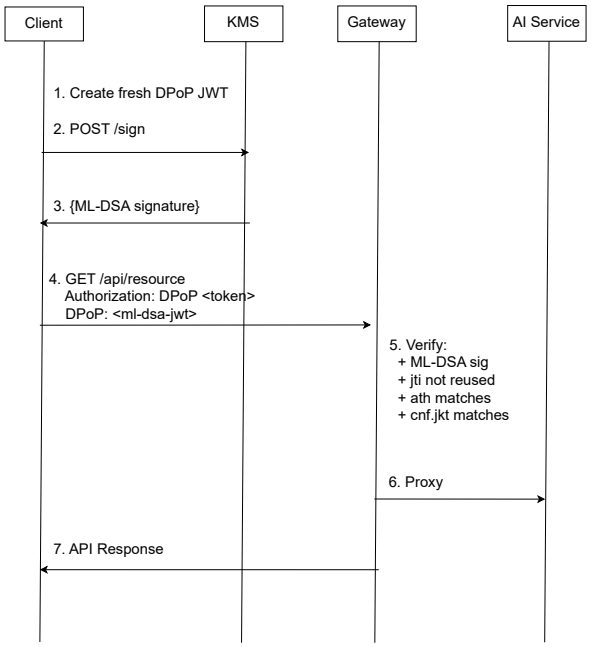

Fig. 5: Phase 2 - API Request with Fresh DPoP Proof

`<ml-dsa-jwt>` header containing the fresh signed proof. At request time, the Gateway verifies the ML-DSA-44 signature using the registered public key, checks `jti` uniqueness via a replay cache, validates the `ath` binding to prevent token substitution, and confirms the `cnf.jkt` match with the access token. The Gateway injects the provider API key (managed per tenant at the gateway) and proxies the request to the AI provider using bearer authentication, and the AI service processes the request and returns the response via the Gateway to the client.

E. Security Properties

Our architecture protects against credential leakage, token theft, and unauthorized access. We formalize the security guarantees as follows:

No provider API keys and no private signing keys in containers. Let $(sk_{\text{dsa}}, pk_{\text{dsa}})$ denote the ML-DSA keypair used for proof-of-possession, where sk_{dsa} is non-exportable and stored exclusively in the KMS. Let C denote container image storage, M volatile process memory, and H KMS storage. Define the sensitive credential set

$$K = \{sk_{\text{dsa}}, \text{API\_key}\}, \tag{1}$$

$$C \cap K = \emptyset, \tag{2}$$

$$M \cap K = \emptyset, \tag{3}$$

$$sk_{\text{dsa}} \in H. \tag{4}$$

The above conditions state that neither provider API keys nor private PoP signing keys appear in container images (C) or in process memory (M), while private signing keys remain confined to the KMS storage (H) as non-exportable material. Here, API_key denotes the provider-issued API key stored only at the gateway. Containers retain only public information and opaque handles required for delegated signing, such as `key_handle`, pk_{dsa}, and configuration metadata. Leakage of container images therefore exposes no usable credentials for direct provider-side API abuse.

Workload identity binding. Let h denote a key handle, m a message, and τ a workload identity token. We write $\text{Sign}(h, m)$ to denote a KMS signing request on message m using the key referenced by handle h. A KMS signing request is rejected unless τ is valid and authorized for h under the KMS access-control policy. Here, $\text{Auth}(\tau, h)$ denotes that τ is unexpired and grants permission to use h. For any adversary $\mathcal{A}$ attempting KMS operations:

$$\neg\text{Auth}(\tau, h) \implies \text{Sign}(h, m) \text{ is rejected}, \tag{5}$$

$$\text{Sign}(h, m) \text{ succeeds} \implies \text{Auth}(\tau, h). \tag{6}$$

Thus, possession of a key handle alone is insufficient to invoke KMS signing. Short-lived identity tokens further limit exposure.

Token binding with proof-of-possession. Let T denote an access token bound to the PoP public key pk_{dsa} via `cnf.jkt`. For any adversary $\mathcal{A}$ holding a stolen token, a successful request implies that $\mathcal{A}$ can generate a valid PoP proof under pk_{dsa}. Here, P denotes a PoP proof (e.g., a DPoP proof), and $\text{Verify}_{pk_{\text{dsa}}}(P)$ denotes successful verification under pk_{dsa} with the required freshness and request-binding checks.

$$\text{ValidRequest}(T) \implies \exists P \text{ s.t. } \text{Verify}_{pk_{\text{dsa}}}(P) = 1. \tag{7}$$

In our architecture, producing such a proof requires invoking KMS signing with an authorized workload identity token and the corresponding key handle. Without KMS access and a valid workload identity, stolen tokens are unusable. Replay is prevented via `jti` uniqueness and `ath` binding to T.

Request binding and replay protection. Each PoP proof binds to request context $(\text{htm}, \text{htu}, \text{ath}, \text{jti})$. Here, $\text{Verify}_{\text{G}}(P)$ denotes successful verification of the PoP signature and key binding (including `cnf.jkt`), together with validation of request-binding fields. For any proof P, let $n := \texttt{jti}$ denote its nonce stored in the replay cache:

$$\text{Verify}_{\text{G}}(P) \wedge n \in \text{ReplayCache} \implies \text{Reject}, \tag{8}$$

$$\text{Verify}_{\text{G}}(P) \wedge \text{htm} \neq \text{actual\_method} \implies \text{Reject}, \tag{9}$$

$$\text{Verify}_{\text{G}}(P) \wedge \texttt{ath} \neq \text{SHA256}(T) \implies \text{Reject}. \tag{10}$$

These checks block replay, cross-endpoint, and token-substitution attacks.

Quantum resistance. PoP and workload-identity signatures are instantiated with ML-DSA, providing post-quantum security consistent with NIST FIPS 204. Forging workload identities or PoP proofs therefore requires breaking ML-DSA. Unlike bearer-token systems, access in our architecture requires a valid workload identity, KMS-mediated signing, and per-request cryptographic proof, with automatic expiration and revocation points at multiple layers.

F. Results

We evaluated security by implementing traditional bearer token authentication and our KMS-based approach against HuggingFace AI services. Figure 6a and Figure 6b illustrate the architectural differences.

TABLE II: Security comparison across authentication methods

Scenario	Authentication	Result
Traditional	API key directly (bearer token)	HTTP 200 (Vulnerable)
Proposed	KMS-signed DPoP (zero secrets)	HTTP 200 (Authenticated)

Traditional approach vulnerabilities. Figure 6a demonstrates current practice: clients embed API keys in container environment variables or configuration files, and container inspection via `docker inspect` exposes API keys in plaintext. Once stolen, these keys grant unlimited access until manual revocation, and the mechanism provides no cryptographic proof of legitimate ownership. As a result, bearer token authentication may succeed with HTTP 200 while providing zero protection against leakage.

Proposed architecture security. Figure 6b illustrates our zero-secret design: containers possess only non-secret artifacts such as key_handle, public key, and endpoint configuration, so container inspection reveals zero API keys or private keys. All signature operations are delegated to KMS with workload identity validation, and DPoP proofs bind to specific request contexts with nonce-based replay protection. At the Gateway, cryptographic proofs are validated before injecting the provider API key, such that legitimate requests succeed with HTTP 200 only after multi-layer verification.

G. Runtime Overhead

Our architecture adds minimal cryptographic overhead to AI service requests. We measured performance across three phases on Intel x86_64 with Ubuntu 22.04 LTS.

Bootstrap overhead (one-time). Table III shows initial registration costs totaling approximately 20ms for network round-trips plus 82μs for ML-DSA-44 keypair generation in KMS. This one-time cost is negligible for container lifecycle (hours to days).

TABLE III: Bootstrap overhead (one-time per container)

Component	Operation	Time
Identity Issuer	JWT request	$\sim$5 ms
KMS	JWT verify + KeyGen	$\sim$5 ms
	(ML-DSA-44 KeyGen)	(0.082 ms)
Auth Server	Client register	$\sim$5 ms
Gateway	DPoP key register	$\sim$5 ms
Total	**Bootstrap**	**$\sim$20 ms**

Token acquisition overhead (per-session). Table IV reports that per-session token acquisition adds approximately 10 ms. Most of this overhead comes from network latency to the KMS; the signing and verification computations contribute 0.606 ms and 0.315 ms, respectively.

TABLE IV: Token acquisition overhead (per-session)

Component	Operation	Time
Client	DPoP JWT creation	0.002 ms
Client $\rightarrow$ KMS	Network + Sign	$\sim$5 ms
	(ML-DSA-44 Sign)	(0.606 ms)
Auth Server	JWT parse + Verify	0.316 ms
	(ML-DSA-44 Verify)	(0.315 ms)
Auth Server	Token generation	0.001 ms
Total (crypto only)		**0.921 ms**
Total (with network)		**$\sim$10 ms**

API request overhead (per-request). Table V reports per-request costs: 0.917 ms for cryptographic processing and approximately 5 ms for KMS network latency.

TABLE V: API request overhead (per-request)

Component	Operation	Time
Client	SHA-256 (ath)	0.001 ms
Client	DPoP JWT creation	0.002 ms
Client $\rightarrow$ KMS	Network + Sign	$\sim$5 ms
	(ML-DSA-44 Sign)	(0.606 ms)
Gateway	JWT parse + Verify	0.316 ms
	(ML-DSA-44 Verify)	(0.311 ms)
Total (crypto only)		**0.917 ms**
Total (with network)		**$\sim$6 ms**

Overhead analysis. The per-request overhead is primarily driven by network latency to the KMS. The cryptographic

```
[Container] curl -X GET https://huggingface.co/api/whoami-v2 \
            -H "Authorization: Bearer hf_nGL..."

[Container] Response: 200

{
  "type": "user",
  "id": "xxxxxxxxxxxxxxxxxxxxxxx",
  "name": "xxxxxxx",
  "fullname": "xxx",
  "isPro": false,
  "avatarUrl": "/avatars/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.svg",
  "orgs": [],
  "auth": {
    "type": "access_token",
    "accessToken": {
      "displayName": "xxxxxxxxxxxxxxx",
      "role": "fineGrained",
      "createdAt": "2025-11-27T11:23:54.520Z",
    }
  }
}
```

(a) Traditional bearer token (vulnerable to leakage)

```
[Container] === TOKEN REQUEST ===
[Container] 1. Created DPoP proof for token endpoint
            DPoP: eyJ0eXAiOiAiZHBvcCtqd3QiLCAiYWxnIjogIk1MLURTQS00NCJ9.eyJqdGk...
[Container] 2. Requesting access token...
            access_token: ShfB07smGutmcXXXtO6v3_0Lg1o1kG...
[Container] === API REQUEST ===
[Container] Calling /api/whoami-v2 with DPoP-bound token...

[Container] Response: 200

{
  "type": "user",
  "id": "xxxxxxxxxxxxxxxxxxxxxxx",
  "name": "xxxxxxx",
  "fullname": "xxx",
  "isPro": false,
  "avatarUrl": "/avatars/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.svg",
  "orgs": [],
  "auth": {
    "type": "access_token",
    "accessToken": {
      "displayName": "xxxxxxxxxxxxxxx",
      "role": "fineGrained",
      "createdAt": "2025-11-27T11:23:54.520Z",
    }
  }
}
```

(b) Proposed KMS-based architecture with no provider API keys and no private signing keys in containers

Fig. 6: Comparison of authentication architectures

component is 0.917 ms, of which ML-DSA-44 signing contributes 0.606 ms and verification accounts for the remainder.

Impact on AI services. On HuggingFace, where an API call has a 50 ms baseline, the end-to-end overhead is approximately 6 ms. For inference workloads with latencies in the 200–2000 ms range, this additional cost becomes marginal in practice. With token caching enabled, the overhead reduces to 0.917 ms, which is below 1% for most workloads.

Comparison. Bearer tokens introduce no additional overhead but provide no protection against credential leakage. In contrast, our architecture adds approximately 6 ms while enabling zero-secret containers, proof-of-possession, replay protection, and quantum resistance.

III. Conclusion

We presented a leak-resilient authentication framework for AI services that removes provider API keys and private PoP signing keys from client-side artifacts. The framework enforces gateway-mediated proof-of-possession using KMS-backed ML-DSA signing and does not require changes to provider-side interfaces. We evaluated a prototype with HuggingFace and found that extending the approach to additional providers is primarily a gateway-side integration effort. Overall, the results support practical leak-resilient AI service authentication with limited overhead.

Acknowledgment

This work was partially supported by the Korea Evaluation Institute of Industrial Technology (KEIT) grant funded by the Korean government (MOTIR) (RS-2025-02634277) and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2025-25457382).

References

[1] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 797–811.

[2] M. Meli, M. R. McNiece, and B. Reaves, "How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories," in *NDSS*, 2019.

[3] D. Hardt, "RFC 6749: The OAuth 2.0 Authorization Framework," 2012.

[4] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite, "RFC 9449: OAuth 2.0 Demonstrating Proof of Possession (DPoP)," 2023.

[5] National Institute of Standards and Technology, "FIPS 204: Module-Lattice-Based Digital Signature Standard," NIST, Tech. Rep., Aug. 2024, federal Information Processing Standards Publication 204.

[6] M. Jones, J. Bradley, and H. Tschofenig, "RFC 7800: Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)," 2016.