

JANUS: Practical Load Balancing for Confidential Cloud Services

Qi Guo^{1†}, Alice Dethise², Ruichuan Chen², Istemi Ekin Akkus², Ivica Rimac², Lieven Trappeniers²

¹Max-Planck-Institut für Informatik

²Nokia Bell Labs

Abstract

Confidential computing enables a cloud service to prove, by remote attestation, that it runs inside a Trusted Execution Environment (TEE). Recent work has bound this attestation to Transport Layer Security (TLS) so that a client can verify TEEs via the TLS channel, but these designs assume a client talking to a single specific attested server. Realistic cloud services, however, are not run on single servers: they are load-balanced, structured as a frontend fronting an elastic pool of backend servers. Extending current attested-TLS designs onto this pool is problematic, forcing either custom client logic, a Certificate Authority (CA) certificate issuance per backend server, or a fresh attestation on every connection.

We present JANUS, a practical load-balancing architecture for confidential cloud services. Specifically, a small attested frontend, running inside a TEE, holds the only client-facing TLS identity: it obtains a CA-issued X.509 certificate that carries its attestation evidence in a certificate extension. The frontend then admits backend servers, attesting each and signing its public key using a short-lived RFC-9345 Delegated Credential so that every backend serves client traffic under the frontend's single attested identity. This also allows each backend to be attested only once at admission rather than per client connection. JANUS piggybacks on what the TLS ecosystem has deployed or is developing — X.509 certificate extensions, ACME certificate issuance, and Delegated Credentials — so it complies with the current networking practices and trends, and changes nothing in the TLS protocol. It supports standard load-balancing modes (i.e., proxy and redirection). Our evaluation shows that the marginal connection-establishment cost of JANUS is as low as 4 ms (in proxy mode) and 12 ms (in redirection mode), compared with vanilla TLS.

CCS Concepts: • Security and privacy → Trusted computing; Distributed systems security; • Computer systems organization → Cloud computing.

[†]Work partly done while the author was an intern at Nokia Bell Labs.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857052>

Keywords: Confidential computing, trusted execution environment, delegated credential, cloud service, load balancing

ACM Reference Format:

Qi Guo, Alice Dethise, Ruichuan Chen, Istemi Ekin Akkus, Ivica Rimac, and Lieven Trappeniers. 2026. JANUS: Practical Load Balancing for Confidential Cloud Services. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15–18, 2026, Hong Kong, China. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3828523.3857052>

1 Introduction

Confidential computing has become widely available in the cloud [6, 7, 10, 11, 35, 64], enabling users to run sensitive workloads on a third-party cloud infrastructure with strong guarantees of confidentiality and integrity [20, 21]. Major cloud operators not only allow users to deploy their own confidential instances, but also offer a wide variety of services powered by confidential computing. For instance, Apple Private Cloud Compute ensures private data processing [11], Azure Confidential AI Inferencing secures AI model execution [78], AWS Clean Rooms protects multi-party data collaboration [9], and Signal ensures user privacy during contact synchronization [62] and secret key recovery [22].

Confidential computing is achieved via the hardware-based Trusted Execution Environment (TEE) technologies on both CPUs and GPUs, such as Intel SGX [45] and TDX [48], AMD SEV-SNP [2], ARM CCA [12], and NVIDIA confidential GPUs [70]. They provide guarantees for confidentiality and integrity when executing code on data within a TEE. This confidential computing paradigm addresses the critical need for organizations to outsource compute infrastructure for scalability and cost efficiency, without trusting the cloud operator's software stack including the privileged hypervisor and operating system (OS). Through the mechanism of remote attestation [23, 39], clients receive hardware-rooted proof that their code and data are securely executed within a genuine TEE and have not been tampered with. For this attestation mechanism to be reliable, the client must establish a connection that terminates inside the TEE being attested. Otherwise, an adversary could intercept the connection, presenting a genuine TEE quote (i.e., attestation evidence) while routing the connection through a non-TEE relay [52, 90].

To mitigate this risk, the state-of-the-art studies [3, 50–52, 89, 90] have enabled the TEE remote attestation through Transport Layer Security (TLS) by binding the attestation to the TLS context, thereby ensuring the trustworthiness of

the TEE’s validity. However, these studies focus on a scenario where a client connects to a *single* specific server (i.e., a single client-server connection). These approaches conflict with the architecture of many modern cloud services that utilize multiple *load-balanced* servers [8, 40, 69] to fulfill client requests, in which a specific *frontend* dispatches client requests to an elastic pool of *backend* servers. The frontend receives client connections and routes them to these backend servers which, in confidential computing scenarios, perform application logic within TEEs. The backend server pool can dynamically scale with workload changes. Adapting existing ‘single client-server’ solutions to such environments will require non-trivial changes to client logic and TLS/TEE practices (details in §2.2, §4 and §7), leading to compatibility, scalability and efficiency issues.

With the increasing deployment of confidential cloud services, there is a critical need for a remote attestation solution that supports the aforementioned frontend-backend architecture of modern cloud services, as well as complies with current networking practices and trends, without increasing connection latency. The primary challenge is to ensure that every backend server a client may connect to is verifiably executing the correct code within a genuine TEE. This challenge is further complicated by the practical need for compatibility: the attestation mechanism must integrate with existing networking ecosystems, including the TLS protocol and Public Key Infrastructure (PKI), to avoid the overhead of each confidential cloud service requiring a custom client. In addition, a practical solution should not impose excessive performance costs on the cloud service and the clients, particularly by avoiding delays during connection establishment.

This paper introduces JANUS, a practical remote attestation design of a load-balancing architecture for confidential cloud services. Specifically, JANUS enables trusted secure channels between clients and the TEE-enabled backend servers in the cloud via correct remote attestation and efficient connection establishment, all while maintaining compatibility with existing network protocols. Distinct from prior work, JANUS offers scalability and efficiency for an elastic pool of backend servers, embodying principles of trust, scalability, compatibility and performance.

JANUS’s core design employs a small, attested *frontend* responsible for routing connections to the *backend* servers. Here, both frontend and backends run within their respective TEEs. The frontend’s TLS certificate establishes the cloud service’s visible TLS identity, directly tied to the code running within the frontend’s TEE. This frontend utilizes an ACME-issued [1] X.509 certificate, which includes its attestation evidence in a certificate extension, aligned with the IETF LAMPS efforts [72, 73]. The frontend supports two standard load-balancing modes (i.e., proxy and redirection).

Once the frontend has started and configured its certificate, it admits individual backend servers to a pool for processing

client requests. Each backend, upon connecting to the frontend, must prove its identity and the validity of its TEE to the frontend. Following successful attestation, the frontend additionally grants an individual, short-lived RFC-9345 Delegated Credential (DC) [16] to the backend, specifically in the redirection mode. This DC ensures that all backends serve client traffic under the same TLS identity, as defined by the frontend that attested them. JANUS leverages the standard TLS mechanisms, including X.509 certificate extensions [73], ACME issuance [1] and DCs [16], to enable remote attestation of a multi-server infrastructure without introducing new primitives, thereby maintaining compatibility with existing networking stacks and trends.

Altogether, JANUS is designed to ensure four critical properties. First, it ensures *scalability*: JANUS supports a prevalent frontend-backend architecture that is widely used in today’s load-balanced cloud services where the backend server pool can scale to deal with varying client traffic. Second, it maintains *confidentiality*: every backend, which hosts a cloud service, reached by a client runs within a valid, attested TEE, where both the proof of confidentiality and the associated TLS identity are traceable and verifiable. Third, it ensures *compatibility*: JANUS is intentionally designed to be compatible with today’s networking ecosystems (e.g., TLS and PKI) and trends, thus readily deployable. Fourth, it achieves *efficiency*: each backend server requires only a one-time attestation upon registering to the frontend rather than per client connection, thus moving the expensive attestation outside the critical path of client connection establishment. Our evaluation with realistic applications shows that JANUS achieves all these properties with a marginal cost as low as 4 ms (in proxy mode) and 12 ms (in redirection mode) at connection establishment, compared with vanilla TLS.

Contributions. Our contributions are as follows:

- We design JANUS which is, to our knowledge, the *first* load-balancing architecture for a confidential cloud service that is served by an elastic pool of attested servers under a single service identity. It is scalable, compatible with the current networking ecosystem, and efficient (§3, §4).
- We implement the end-to-end JANUS system (§5), and evaluate it against vanilla TLS and four state-of-the-art baselines across three representative workloads (§6). Our results show that JANUS serves a confidential server pool with a marginal per-connection cost as low as 4 ms (in proxy mode) and 12 ms (in redirection mode) compared with vanilla TLS (much shorter than the state-of-the-art).

2 Background

2.1 Remote Attestation and TLS Binding

A TEE provides a hardware-based isolated execution environment, protecting its confidentiality and integrity against an adversary which can control the privileged operating system (OS), hypervisor, BIOS, and peripherals [20, 21]. Modern TEE

platforms, such as Intel SGX [45] and TDX [48], AMD SEV-SNP [2], ARM CCA [12] and NVIDIA confidential GPUs [70], root this isolation in hardware keys stored within the CPU or GPU. These platforms expose a *remote attestation* primitive: on demand, the TEE generates a hardware-signed *quote* (i.e., attestation evidence). This quote binds a measurement of the executing code, the TEE platform identity and security version number, as well as a REPORTDATA field configurable by the software running inside the TEE [20, 21].

A client can verify the attestation quote by using the TEE manufacturer’s PKI to validate the hardware key that signed the quote. In practice, this verification is often delegated to a third-party Attestation Service (AS) such as Microsoft Azure Attestation (MAA) [65], Intel Trust Authority [47] or Google Confidential Space [36], which itself runs inside TEEs using Intel Data Center Attestation Primitives [44, 46] or similar techniques. The AS checks the attestation quote and returns a verifier-friendly JSON Web Token (JWT) summarizing the attestation claims which include the hash of the executing code, the hardware details, and the REPORTDATA value.

An attestation quote, however, does not inherently indicate which network endpoint the attested TEE is hosting. For confidential cloud services communicating with remote clients, the attestation must therefore be *bound* to the secure communication channel (e.g., TLS) terminating inside the TEE [90]. This allows the client to verify, in a single check, both that it is communicating with a genuine TEE and that its communication channel terminates within that TEE. A standard binding method involves writing a digest of the TEE-controlled TLS public key into the attestation quote’s REPORTDATA field. As a result, the attestation quote contains both the secure hash of the code that generates and manages the public/private keypair (indicating the secure storage of the private key in the TEE) and the hash of the public key (guaranteeing the correct binding of the public key to the TEE via REPORTDATA and that the corresponding private key belonging to the TEE).

2.2 TLS Termination in Load-Balanced Deployments

Existing work on binding remote attestation to TLS context assumes that the client establishes a connection to a single specific server which processes the request [3, 50–52, 89, 90]. However, this assumption often fails in reality. Typically, a cloud service infrastructure does not expose backend servers directly. Instead, it places a frontend, e.g., a load balancer, in front of a pool of backend servers. The frontend receives client requests and distributes them across the backends, usually by proxying or by redirecting the client.

Proxy Mode. In this mode, the frontend functions as a reverse proxy. It terminates the client’s TLS connection locally and, based on its internal logic, selects a backend to process the client’s session. The frontend then establishes a new internal connection to the selected backend and forwards the client request over this new connection. This model is

widely deployed using the reverse proxy software or managed load balancer [8, 40, 69]. Centralizing TLS termination at the frontend load balancer simplifies policy enforcement. However, this approach turns the load balancer’s data path into a potential bottleneck and a single point of failure. In the context of remote attestation through TLS connections, this approach has an important implication, because the connection is intercepted at a server separate from the application logic, which may not possess the same level of trust.

Redirection Mode. In this mode, the frontend load balancer participates only in the initial routing decisions. After admitting a client request and selecting a backend server, the load balancer redirects the client to a new address via mechanisms such as DNS, HTTP redirect, or an application-level discovery protocol [17, 28]. The client then establishes a new connection directly to the selected backend server. This approach removes the data-path bottleneck at the frontend and enables backend throughput to scale with the number of backend servers. However, it forfeits per-request Layer-7 visibility at the load balancer after the initial routing. Crucially, in the context of this work, this mode also necessitates that every backend server presents its own TLS identity to clients during the TLS handshake.

These two load-balancing modes create very different scenarios for the TLS architecture. In proxy mode, the client-facing TLS remains centralized at the frontend, and the client has no direct visibility into the backends that process application logic. In redirection mode, TLS connections are distributed across a pool of backend servers. In either case, it is impractical to simply generate a long-lived Certificate Authority (CA)-issued TLS certificate for each backend server to bind its TEE attestation to its TLS identity. Such an approach would turn the certificate signing requests (CSRs) into a bottleneck when scaling the backend server pool. One problem that JANUS needs to address is how to project a single attested TLS identity in both load-balancing modes.

3 Actors, Assumptions and Goals

This section establishes the context for the JANUS design.

3.1 Example Scenario

We consider a prevalent scenario for a confidential cloud service, where the service needs to be horizontally scalable. Among many others, one example scenario is a privacy-preserving large language model (LLM) inference service [11, 78]. The service is deployed across a pool of load-balanced servers with TEE support. In this scenario, when a client makes a request, it first reaches the frontend load balancer, which routes the request to a backend server through either proxy mode or redirection mode. The backend server, which hosts the LLM, exposes an API to process the request in a privacy-preserving way. For a client to trust that the backend server will process the request confidentially and faithfully

inside a genuine TEE, the client must remotely attest the backend before sending a request or receiving any data.

This setting can be generalized beyond the exemplified LLM inference service. Any scenario, in which a confidential cloud service (e.g., contact discovery [62], multi-party data collaboration [9]) runs in a third-party cloud infrastructure, involves similar processing steps.

3.2 Actors and Trust Assumptions

Here, we describe the actors in our scenario and the assumptions about them. There are 5 types of actors that, explicitly or implicitly, participate in the client request processing:

- ① **Client.** The client is an entity that makes a request to the confidential cloud service. The confidentiality of the request data must be protected; therefore, the client must be convinced that the cloud service is running correctly inside a TEE by performing remote attestation. The client can be an application, a networking library, or a standard browser.
- ② **Service Provider.** The service provider is an entity that provides the application code, deploys the code into the cloud, and operates the corresponding cloud service. Such code may need to stay confidential due to business reasons. The service provider is responsible for proving to the clients that this code is faithfully used to process client requests. Note that, distributing the expected measurement value of the application code to clients is outside the scope of this work, and we assume clients can obtain such a measurement value via existing transparency channels (e.g., Sigstore’s Rekor [68], CT-style logs [54, 55], or vendor programs such as Apple’s transparency log [11]).
- ③ **Cloud Operator.** The cloud operator supplies the necessary cloud resources and TEE infrastructure to the service provider. It is otherwise oblivious to the cloud service and client requests. Neither the client nor the service provider trusts the cloud operator to see their respective confidential assets (e.g., client requests, cloud service’s code). The cloud operator can access the cloud infrastructure stack such as the privileged host OS, hypervisor and BIOS. An adversarial cloud operator may observe, modify, drop, or inject traffic to and from the frontend and backends, and may tamper with the software running on the hosts in the cloud.
- ④ **Certificate Authority.** The certificate authority (CA) (e.g., DigiCert [27], GlobalSign [31], Let’s Encrypt [59]) is an entity that provides a client with the root of trust for the identity of a service provider. We do not expect a CA to perform any tasks beyond its existing provisions. Our only assumption is that the CA preserves the certificate extension related to attestation quote (i.e., does not strip it before signing the certificate), which aligns with the current IETF LAMPS efforts the community has been developing [72, 73]; if not supported, a small local patch suffices (§5).
- ⑤ **Attestation Service.** The attestation service (AS) (e.g., Azure MAA [65], Intel Trust Authority [47], Google Confidential Space [36]) is an entity that is responsible for helping

the client to verify a TEE quote and attest whether the said quote is valid. As a result, the AS acts as the root of trust for TEE-related activities via the mechanisms provided by the TEE manufacturers. We assume that the AS signing keys rotate infrequently, a fact we empirically confirm through a measurement of the production ASes (§6).

We note that the CA and AS have complementary but separate responsibilities: the CA is tasked to verify the identity of a service provider, whereas the AS verifies the TEE claims (e.g., genuine and valid TEEs). In real world, both of them are *already* trusted by the clients and service providers.

TEE Resilience. TEEs ensure the code and data loaded into them are kept confidential and integrity-protected [20, 21]. We note that there are attacks against TEEs (e.g., side- and covert-channel attacks [38, 56, 57, 60, 67, 80, 86, 87, 91], physical attacks [18, 24, 82], and rollback attacks [75, 84]). To defend against these attacks, the TEE manufacturers and the community have designed various mechanisms across the infrastructure stack [19, 22, 41–43, 71, 81, 88]. In this paper, we make the standard assumption as in the literature that the TEE primitives are resilient against these attacks, maintaining their confidentiality and integrity guarantees, and are kept up-to-date, but note that these defense mechanisms are orthogonal and complementary to our design.

JANUS Service Code. The JANUS service code run on the frontend and backend is open-sourced and auditable by all actors to gain their trust. All TLS private keys used in JANUS are generated and sealed inside TEEs. Consequently, a valid private key for each TLS certificate or Delegated Credential exists only within the original TEE that generated it. We assume that, for both the frontend and backend, the JANUS service code does not exfiltrate their own private keys (e.g., by writing to an unprotected storage). This property can be verified since the service code is open-sourced.

3.3 Design Goals

Goal 1: Scalability. JANUS should support the prevalent frontend-backend architecture widely used by today’s cloud services where a pool of backend servers scales to handle dynamic client traffic (as compared to the ‘single client-server scenario’ in prior attested-TLS designs). New backend servers must be able to register, be attested, and be decommissioned quickly to adapt to client traffic fluctuations.

Goal 2: Confidentiality. To ensure the confidentiality of the client requests and service responses, JANUS must ensure that the client establishes a TLS connection which terminates within a TEE, and that the valid attestation and TLS handshake are cryptographically bound to each other.

Goal 3: Compatibility. JANUS must be compatible with today’s networking ecosystems (e.g., TLS and PKI) and their trends (e.g., the IETF LAMPS efforts [72, 73]). Only the extra checks of TEE attestation quotes might necessitate the JANUS-aware verification logic, but even this logic must be expressed as a thin layer over existing, unmodified TLS APIs.

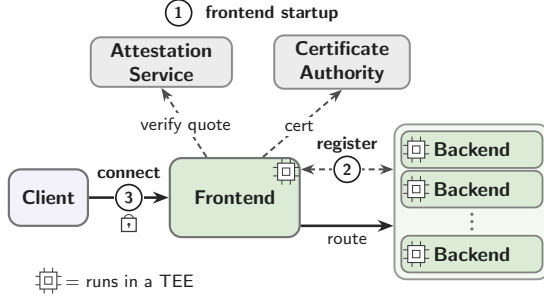


Figure 1. Overview of the JANUS architecture.

Goal 4: Efficiency. The attestation of any server joining the pool of backends is essential, but the generation of an attestation report can be costly; thus, it should be minimal. This report must be secure to attest to the running code, yet general enough for reuse across all client connections. Similarly, the involvement of certificate authorities and attestation services on the critical path should also be minimized.

4 System Design

We give the architecture of JANUS, followed by its detailed mechanisms employed in three stages: frontend startup, backend registration, and client connection establishment.

4.1 Architecture Overview

Figure 1 shows an overview of JANUS. The service provider deploys a frontend and a pool of backend servers for its cloud service. The *frontend* runs inside a small, attested TEE. It holds the sole client-facing X.509 certificate issued by a *Certificate Authority* (CA), and is responsible for remotely attesting backend servers and issuing each with a short-lived RFC-9345 Delegated Credential (DC) [16]. Each *backend* also runs inside a TEE and hosts the application code from the service provider to serve client requests assigned by the frontend. TEEs in JANUS can be process-level enclaves (e.g., enabled by Intel SGX [45]) or more powerful Confidential Virtual Machines (CVMs) (e.g., enabled by Intel TDX [48] or AMD SEV-SNP [2]), potentially utilizing confidential GPUs [70]. Finally, a *client* performs standard TLS validation using its own chain-of-trust certificates against the frontend and backends, augmented by a small number of TEE-related checks utilizing the *Attestation Service* (AS). The JANUS workflow then consists of three main stages.

Stage 1: Frontend Startup (§4.2). When a frontend starts, the JANUS service code generates the frontend’s internal TLS keypair and obtains a long-lived X.509 certificate from the CA. This certificate contains the standard identity information used as the sole client-visible identity for the entire cloud service, as well as the frontend’s verifiable attestation evidence in an X.509 certificate extension [72, 73]. The certificate and keypair are sealed using the standard TEE sealing mechanisms, so that they can be safely stored in untrusted

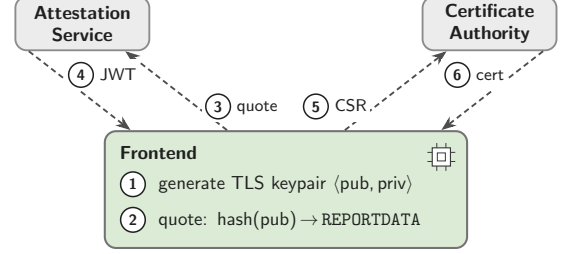


Figure 2. Frontend startup process.

storage [20, 21]. In case of a failure at the frontend, it can simply be restarted, so that it can unseal its own certificate and keypair without requiring a new request to the CA.

Stage 2: Backend Registration (§4.3). The frontend can, at any time, admit new servers to the backend pool. When joining, the JANUS service code on the new server must produce a TEE attestation evidence bound to its freshly-generated TLS keypair during the remote attestation by the frontend. The frontend then verifies this evidence through the AS. Upon successful verification, the frontend signs the new server’s public key through the issuance of an RFC-9345 Delegated Credential (DC) [16]. Issuing a DC involves only a local signature without involving a CA, enabling backends to be added and removed at seconds-scale.

Stage 3: Client Connection Establishment (§4.4). A client initiating a new connection to the cloud service first contacts the frontend, which presents the cloud service’s public identity. Specifically, the frontend provides its X.509 certificate, including the CA’s signature and the frontend’s attestation evidence, which the client verifies via the AS’s public key. The client is then routed to a backend server via one of the two load-balancing modes (see §2.2): In proxy mode, the frontend acts as a proxy to forward the connection to a selected backend. In redirection mode, the client first connects to the frontend to receive connection information of a selected backend, and then opens a second, direct connection to that backend. In either case, the connection terminates at a TLS endpoint that is bound to the TEE attestation evidence.

4.2 Frontend Startup

Each cloud service has its own JANUS frontend. Figure 2 shows how a frontend starts in six steps.

Step 1: Upon its initial startup, the frontend generates a fresh TLS keypair within the TEE.

Step 2: To bind this keypair to the TEE, the frontend generates an attestation quote, embedding the SHA-256 hash of its public key into the quote’s REPORTDATA field (§2.1). This quote generation cryptographically binds the REPORTDATA to the TEE attestation evidence, ensuring the resulting quote is specific to this exact TLS keypair within this precise TEE instance. Because the JANUS service code does not leak the private key (§3.2), this TEE-to-TLS binding guarantees that

only this TEE instance can access the private key and decrypt messages encrypted with the associated public key.

Note that, the attestation quote generation in this step does not use a nonce, as the TLS keypair’s random generation within the TEE prevents its reuse by another (different) TEE instance. For per-client-connection secrecy and replay protection, JANUS leverages the TLS handshake protocol, which inherently includes a per-connection nonce. Thus, the attestation quote establishes identity (i.e., TLS keypair and TEE measurement), while the TLS protocol guarantees freshness and forward secrecy.

Step ③: The frontend submits the attestation quote to an Attestation Service (AS).

Step ④: The AS verifies the quote against the TEE manufacturer’s criteria (e.g., platform hardware and software version). Upon successful verification, it returns an AS-signed JSON Web Token (JWT). This JWT acts as the endorsement to the correctness of the attestation claims, including the measurement of the executing code (e.g., MRENCLAVE in Intel SGX), the TEE instance details (e.g., the identity of manufacturer, software version), and the REPORTDATA field.

Step ⑤: The frontend issues an X.509 Certificate Signing Request (CSR) using its TLS keypair, embedding the AS-signed JWT as an extension [72, 73]. If the frontend employs a redirection mode (§2.2), the CSR additionally includes a DelegationUsage extension [16], authorizing the resulting certificate to sign Delegated Credentials (DCs) later (§4.3).

Step ⑥: The CA verifies the CSR, and binds the JWT to the certificate’s TLS identity. Therefore, the attestation evidence and the TLS identity become inseparably bound. Following the standard TLS identity verification, the CA returns the signed certificate to the frontend, enabling it to serve client requests and emit DC signatures.

Persistence and Fault Tolerance. After the frontend starts, JANUS stores the issued certificate, the JWT, and the TLS keypair to a *sealed* storage using the sealing mechanism [20, 21]. This storage can only be decrypted by the exact same TEE measurement running on the same physical hardware. This mechanism ensures that a different or tampered image cannot derive or decrypt the sealed secrets.

This sealing capability is critical to the frontend’s fault tolerance, making it resilient to crashes and restarts. After a failure (e.g., a process crash, an OS reboot, or a redeployment of the same TEE image on the same hardware), the newly-started frontend unseals its certificate, JWT and TLS keypair to resume serving immediately. This recovery requires no new attestation quote generation, no AS round trip, and no CA certificate issuance. This recovery is also non-disruptive to the rest of the JANUS system: client trust remains anchored in the unchanged certificate, backends do not need to re-register, and all outstanding backend DCs remain valid as they were signed by the same, recovered frontend key. The frontend re-attests and re-issues its certificate only under specific, rare conditions: when the JWT

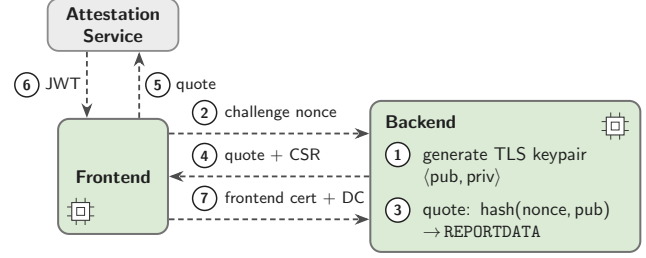


Figure 3. Backend registration process.

or certificate nears its expiry, when the frontend software gets updated, or when a new TEE platform patch is applied; these operations are not required during normal operation. In the end, our frontend startup design ensures that the TEE attestation and CA certificate issuance normally occur only once at the frontend startup, and remain outside the critical path of each client connection (**Goal 4** in §3.3).

Small and Stable Trusted Computing Base (TCB). The frontend’s limited functionality (with no service provider’s application logic) ensures a small TCB, allowing it to be open-sourced and auditable in terms of its correctness. As a result, its codebase and its measurement value remain fairly stable, which also helps reducing the re-attestation and re-issuance of new certificates for the frontend.

4.3 Backend Registration

After starting a frontend, JANUS proceeds to register its backend server pool (**Goal 1**). The registration of new backend servers is managed via the remote attestation mechanism. The backend runs a TEE which loads the cloud service’s application code from the service provider. Figure 3 shows the six common steps to register a backend with the frontend, and another step specifically for the redirection mode.

Step ①: The backend generates a fresh TLS keypair in TEE.

Step ②: The backend contacts the frontend which responds with a unique challenge nonce for replay protection.

Step ③: The backend produces an attestation quote, with its REPORTDATA field committing to the hash of the challenge nonce concatenated with the backend’s TLS public key, ensuring the quote is both fresh and unambiguously bound to this backend’s TLS identity.

Step ④: The backend constructs a Certificate Signing Request (CSR) using its TLS keypair, and transmits the CSR along with the attestation quote to the frontend.

Step ⑤: Upon receipt, the frontend verifies the freshness of the nonce (along with the backend’s public key) by checking the hash embedded in the attestation quote’s REPORTDATA field. It then submits the quote to an Attestation Service (AS).

Step ⑥: The AS verifies the quote against the TEE manufacturer’s criteria (e.g., platform hardware and software version). Upon successful verification, it returns an AS-signed JWT that summarizes the attestation claims including the

hash of the executing code, the hardware details, and the REPORTDATA value. This attestation is done only once per backend registration and can be reused across client connections (**Goal 4**), in contrast to the per-client-connection attestation in the state-of-the-art [3, 50–52, 89, 90].

Load Balancing Modes. JANUS supports two load-balancing modes, i.e., proxy mode and redirection mode (§2.2), with different means for clients to authenticate backend servers. In proxy mode, the backend registration ends here.

In redirection mode, a client needs to communicate directly with a backend server selected by the frontend. Therefore, the backend server must prove its trustworthiness to the client. A naïve approach would be to provision each backend with the frontend’s long-lived private key; however, this approach compromises key confidentiality. Alternatively, minting a unique CA-issued certificate for each backend would introduce scalability issues due to rate limits at public CAs. One approach would be to use a non-public CA as often done in enterprise settings; however, this approach requires this CA’s certificate to be installed and trusted at the clients.

JANUS sidesteps these limitations by leveraging the RFC-9345 *Delegated Credentials (DCs)* [16]. Specifically, in redirection mode, the frontend additionally signs the registering backend server’s public key through the issuance of a DC, and then returns this DC together with the frontend’s certificate to the backend (**Step 7**). These, served as a trust chain, will be used to prove the trustworthiness of the backend during the client connection establishment (§4.4).

Delegated Credential (DC) in Redirection Mode. DC is a standardized TLS 1.3 extension [16] that enables a certificate holder (i.e., frontend in JANUS) to delegate authentication to another server (i.e., backend) without exposing the long-lived private key. This gives the frontend the authority to emit signatures for the backends’ TLS keypairs, using its own key as a link in the PKI chain of trust. Issuing a DC requires only a local signature by the frontend, with no CA involvement. Essentially, this approach projects the frontend’s single attested TLS identity onto a pool of backends, efficiently managing their registration; thus, enabling each backend to serve client requests under the frontend’s TLS identity. The DC support is already integrated into BoringSSL [33], NSS [66] and Fizz [63], and is enabled by default in Firefox [49], making it readily deployable.

In JANUS, the backend presents a DC with its public key, together with the frontend’s certificate (which embeds the AS-signed JWT). The DC is short-lived (up to 7 days [16]), signed by the frontend’s private key, and carried in the backend’s TLS Certificate message [16, 77]. Unlike X.509 proxy certificates [85], a DC is not a CA-issued certificate and does not extend the X.509 certification path. Instead, the client validates the frontend’s certificate chain as in standard TLS, and then verifies the DC against the validated frontend certificate. Thus, JANUS preserves the usual PKI validation while enabling short-lived delegation without extra round trips.

Handling Backend Irregularities. Our design does not require a revocation mechanism during backend decommissioning: the frontend simply ceases renewing the backend’s DC, allowing it to lapse within its short validity window. In the event of a backend compromise, the frontend not only ceases renewing its DC but also immediately excludes it from the backend pool. Re-registration of the backend is then required, but without the involvement of the CA. A similar mechanism also handles backend crashes: the frontend’s health check drops an unresponsive backend from its in-service pool and routes traffic around it. A restarted backend then re-registers precisely as a fresh one would. Finally, when the frontend’s TLS certificate is renewed (§4.2), it proactively re-signs all in-service DCs and sends them to the backends. All these operations to handle backend irregularities are simple and efficient, enabling the backend pool to be scaled, recycled, and patched at auto-scaling speed.

4.4 Client Connection Establishment

The establishment of a connection between a client and a backend server, which runs the actual application logic, depends on the load-balancing mode used by the cloud service.

Proxy Mode. In this mode, the frontend is the sole client-facing endpoint. As shown in Figure 4a, when a client wants to access a cloud service, it establishes a TLS connection with the TEE of the cloud service’s associated frontend. The client then runs the following checks during the TLS handshake:

1. Validate the standard TLS certificate chain of the frontend. This step anchors the trust chain into the PKI, so the existing TLS stack can be used (**Goal 3**).
2. Extract the JWT from the certificate extension and verify its signature with the AS signing key to ensure that the JWT is genuine.
3. Check that the JWT’s REPORTDATA field contains the SHA-256 hash of the public key advertised in the frontend’s TLS certificate, ensuring the TEE-to-TLS binding and that the TLS terminates within the frontend’s TEE (**Goal 2**).
4. Check the claims of the TEE attestation including the measurement of the executing code (e.g., MRENCLAVE) and the hardware details (e.g., TEE platform identity and security version number) against the client’s trust policy, if any.

The client stops the connection if any of these checks fails. Otherwise, the frontend further establishes a TLS connection to a selected¹ backend’s TEE, as it has all necessary TLS- and TEE-related information about the selected backend during backend registration (§4.3). The frontend then transparently proxies client requests to the selected backend, thus serving as an intermediary to bridge the client and the selected backend over a TEE-bound TLS session (**Goal 2**). Note that, the TLS connection between frontend’s and backend’s TEEs can be established on a per-request basis or maintained as a

¹The backend selection criteria are outside the focus of this work.

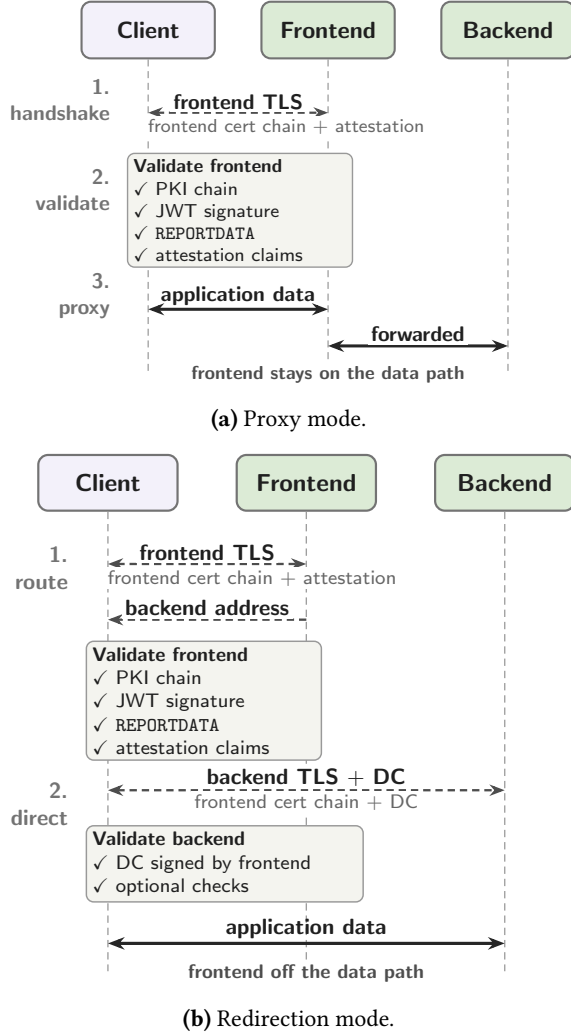


Figure 4. Client connection establishment process with two different load-balancing modes.

single persistent connection between the frontend and each backend, with connection multiplexing.

Redirection Mode. In this mode, as shown in Figure 4b, when a client accesses a cloud service, it first establishes a TLS connection with the TEE of the cloud service’s associated frontend and, as in proxy mode, performs the same four checks during the TLS handshake. In redirection mode, the frontend additionally returns the selected backend server’s address. If all checks pass, the client establishes another TLS connection directly with this backend’s TEE. During the handshake, the backend’s Certificate message [77] includes its Delegated Credential (DC) containing the backend’s public key and validity period, signed by the frontend’s private key. Therefore, the client verifies the backend’s DC by checking its signature with the frontend’s public key (embedded in the frontend’s TLS certificate). This step anchors the trust chain back into the PKI via the frontend’s

TLS certificate and the DC mechanism (**Goal 3**), without involving a CA (**Goal 4**). This enables all client traffic to be served under the single attested TLS identity of the frontend. Additional checks on the backend’s JWT, REPORTDATA and attestation claims (similar to Checks 2-4 in proxy mode) are not required at the client side because they were performed during backend registration by the frontend (§4.3).

Finally, the client starts sending requests to the backend without involving the frontend. This mode takes the frontend off the data path, so the overall system throughput scales with the size of the backend server pool (**Goal 1**).

Caching AS Signing Key. Although most operations in both proxy and redirection modes are kept local and off the per-client-connection path, fetching the AS signing key to validate the signature of JWTs could intrude on validation performance. In JANUS, each client caches this AS signing key in its first connection and uses it for subsequent connections, as all JWTs from an AS are signed by the same key. Additionally, for pre-configured or well-known ASes (e.g., Azure MAA [65], Intel Trust Authority [47], Google Confidential Space [36]), AS signing keys can be pre-fetched. This approach minimizes contact with the AS (**Goal 4**), mirroring how CA public keys in today’s PKI are often bundled with the networking libraries and browsers. As a result, the marginal per-connection cost over vanilla TLS is purely computational: no additional network round-trips on the cached path, and only a small constant number of operations to check certificates, DCs, JWTs and attestation claims.

Note that, the cached AS signing keys must be refreshed upon their expiry. Our caching strategy assumes that the AS signing keys rotate infrequently, a fact we empirically confirm through a measurement of production ASes (§6). Alternatively, this signing key refresh can also be triggered passively whenever a JWT check fails.

4.5 Chain-of-Trust and Discussions

Chain-of-Trust. The three stages in JANUS form a chain of trust rooted in what the client trusts (§3.2): the public CA, the AS signing key, and the TEE measurements. When a client connects to the frontend, it checks the validity of the frontend’s TLS certificate and TEE status, as well as the TEE-to-TLS binding and the measurement of the executing code (§4.4), all of which were established during frontend startup (§4.2). After checking, the client is certain that the frontend was deployed by a verified service provider in an attested TEE with the expected code, and that the TLS terminates in this TEE; thus, establishing trust in the frontend. The backend registration further extends this trust to backends (§4.3): the frontend admits a backend to the server pool only after successfully verifying its freshness and TEE status.

There are two load-balancing modes between a client and a backend. In proxy mode, the frontend serves as an intermediary to bridge the client and the backend over a TEE-bound

TLS session. In redirection mode, after the trust establishment in the frontend and obtaining the necessary connection information for a backend, the client connects directly to the backend’s TEE and verifies its Delegated Credential (DC), thus anchoring the chain of trust back into the PKI via the TLS and DC mechanisms.

Increased Availability with Multiple Frontends. In JANUS, each cloud service is associated with one frontend (§4.2), which potentially creates a single point of failure. To increase availability for a cloud service, multiple frontends can run behind an ordinary Layer-4 load balancer that forwards TCP without terminating TLS, and therefore, holds no secret and accesses no plaintext messages. Each such frontend has its own attested public key, certificate and backend pool.

Compromised Frontend or Backend. A compromised frontend could break the admission integrity of backends. In proxy mode, it also exposes traffic traversing the frontend. In redirection mode, it could redirect client requests to non-authenticated backends, though the client connections already established with genuine backends still keep their protection since the frontend never holds their session keys. For these reasons, each frontend runs only the JANUS service code with a small, auditable TCB (§4.2), so that this code can be easily maintained and hardened.

A compromised backend affects only its own connections, and never the frontend’s private signing key. The backend’s short-lived DC stays valid until it expires, as RFC-9345 has no revocation [16]; an urgent revocation means replacing the frontend’s TLS certificate or a fresh frontend startup.

Post-Quantum Safety. JANUS adds no new cryptographic construction of its own. Its post-quantum exposure is inherited from the PKI, TLS, DC and TEE stacks, meaning their post-quantum upgrades can seamlessly carry over.

5 Implementation

We have implemented and open-sourced² the JANUS system.

Frontend. We implemented the frontend as a Python service on Flask [74] that exposes the REST APIs to handle backend registrations and client connections. We built it to run on Intel SGX [45] with the Gramine [37] library OS. The frontend runs only the open-sourced, auditable JANUS service code without application code from the service provider, making its TCB small and stable (§4.2).

Backend. The backend was implemented as a Python service on Flask, and deployed on AMD SEV-SNP [2] (i.e., a VM-level TEE) which allows the backend to host the unmodified application code of a cloud service. Since Python’s standard TLS stack lacks support for the RFC-9345 Delegated Credential (DC) [16], we developed a specialized reverse proxy for the redirection mode. This proxy, written in C atop BoringSSL [33], terminates DC-credentialed TLS on the client-facing side within the TEE, and relays messages to

the application code over a Unix domain socket. In doing so, the DC mechanism is introduced without changes to the application code of the cloud service. Note that, our TEE selection is non-binding: the JANUS design is TEE-agnostic and can operate with any TEEs that provide an attestation quote with the REPORTDATA primitive or similar.

Client. To demonstrate both the browser-integrated and custom scenarios, we implemented two distinct clients: a Firefox extension and a C client built on NSS [66]. We chose Firefox and NSS because they have integrated the client-side support for Delegated Credential (DC).

Certificate Authority (CA). As discussed in §3.2, we require that the CA preserve the attestation-related CSR extensions rather than stripping them, aligned with the current IETF efforts [72, 73]. For ease of evaluation, we deployed the Pebble CA [58] from Let’s Encrypt. Pebble does not natively support this feature; therefore, we patched it with only around 50 lines of code to preserve these CSR extensions.

6 Evaluation

We evaluate JANUS to answer three questions:

1. **Overheads.** What overheads does JANUS impose as compared to the state-of-the-art systems? (§6.2)
2. **Scalability.** How does JANUS scale as compared to these systems? (§6.3)
3. **Applications.** Does JANUS preserve efficiency for three representative confidential workloads — web browsing, LLM inference, and microservice? (§6.4)

6.1 Experimental Setup

We evaluated JANUS in Microsoft Azure cloud. The frontend was deployed on an Intel SGX machine (DC4s_v3). We used up to 32 AMD SEV-SNP CVMs (DC2ads_v5) for backends, and a separate VM as the load generator. For LLM inference, we employed an extra AMD SEV-SNP CVM with NVIDIA H100 in TEE mode (NCC40ads_H100_v5) [26, 70]. The Attestation Service (AS) is Microsoft Azure Attestation [65], and returns RS256 JWTs whose signing keys are cached by clients (§4.4). Base RTT between all machines is <1 ms; larger RTT is emulated by injecting artificial delays using `tc-netem` [61]. These RTTs represent a client co-located with the cloud service (LAN), a typical intra-continental client-to-region distance (40 ms), transatlantic (80 ms) and transpacific (120 ms) connections, respectively.

We compare JANUS’s two load-balancing modes (i.e., proxy and redirection) against two state-of-the-art attested-TLS baselines, **RA+TLS** [90] and **HTTP/2** [51], as well as two other approaches with attestation outside the TLS handshake, **Oak** [34] and **OHTTP+KMS** [78]. RA+TLS uses a fresh quote within the TLS 1.3 handshake, while HTTP/2 employs a post-handshake attestation exchange. **Oak** attests on every connection, but establishes its channel over TLS

²<https://doi.org/10.5281/zenodo.23240477>

Table 1. Median latencies of frontend startup and per-backend registration.

Step	Frontend (SGX)	Backend (SNP)
TLS keypair generation	5 ms	3 ms
TEE quote generation	19 ms	1.9 s
AS verification	112 ms	38 ms
CA certificate issuance	7.0 s	—
DC issuance (only in redirection)	—	0.03 ms
End-to-end total	~7.1 s	~2.0 s

without load balancing. **OHTTP+KMS** uses a Key Management Service (KMS), a gateway in each backend attesting to the KMS for every epoch key, and a non-TEE load balancer relaying ciphertext. We report ‘OHTTP+KMS-warm’, where the client and gateway already hold their keys, and ‘OHTTP+KMS-cold’, where a key rotation just happened. All systems utilize the *same* backend, ensuring that any observed differences reflect pure protocol overheads.

6.2 Overheads

We measure JANUS’ system overheads regarding frontend startup (§4.2), backend registration (§4.3), and client connection establishment (§4.4). Each overhead is evaluated over 50 runs, with a fresh start initiated for every run. Since our patched Pebble CA (§5) runs co-located with the frontend, we inject a representative 40 ms RTT on the frontend–CA path to model a remote CA.

Frontend Startup Overhead. The frontend startup is dominated by the duration of obtaining a certificate from the CA (Table 1), which is primarily due to the ACME domain validation. This overhead is incurred only once for the duration of a cloud service’s deployment or the certificate’s lifetime (whichever is shorter), even in the event of a restart (§4.2). Extrapolation from this duration also highlights the inefficiency of a naïve solution, in which each backend would receive its own CA-signed certificate upon registration. Such an approach would incur slow requests to CAs, harming the scalability of the confidential backend pool as well as the efficiency of managing it. In addition, additional monetary costs may be incurred for every certificate issuance.

Backend Registration Overhead. Table 1 also shows that the overhead incurred when a backend registers itself to the frontend is dominated by its TEE quote generation. This process involves fetching the SEV-SNP VCEK certificate chain, which dominates the latency: caching the chain reduces the TEE quote generation from 1.9 s to 140 ms.

Even when a backend server registers itself without a cached certificate chain, the quote generation cost is paid only *once* per registration. After the frontend verifies the backend quote via the AS, it admits the backend into the server pool, allowing the clients to transitively trust the backend, as they have already verified the frontend’s TEE

Table 2. Median client connection establishment latencies (ms) with different RTT settings. Values in parentheses show the speedups over the state-of-the-art ‘RA+TLS’ system.

RTT (ms)	Vanilla TLS	JANUS-proxy	JANUS-redir.	RA+TLS	HTTP/2	OHTTP+KMS warm	OHTTP+KMS cold	Oak
0 (LAN)	2	6 (17.5×)	14 (7.5×)	105	107	13	210	126
40	83	86 (3.2×)	217 (1.3×)	275	312	136	465	292
80	163	167 (2.6×)	417 (1.1×)	438	524	257	712	457
120	243	247 (2.5×)	617 (0.99×)	608	724	377	967	623

quote. In JANUS’s redirection-mode load balancing, the overhead for the frontend to issue a DC to the backend (i.e., an ECDSA operation) is negligible (0.03 ms).

Note that, the overheads of frontend startup and backend registration, by design, are kept off the per-client-connection path. TEE quote generation and CA certificate issuance occur only at the frontend startup and backend registration, never on a client connection.

Client Connection Establishment Overhead. A key user-perceived performance metric for confidential cloud services is the client connection establishment latency, which is defined as the time required to establish an attested-TLS connection to the server that runs the application code to serve client requests. Table 2 reports the connection establishment latencies of JANUS (two modes), vanilla TLS and the state-of-the-art baselines, under different RTT settings (§6.1).

In a LAN, JANUS-proxy establishes an attested-TLS connection 17.5× faster than RA+TLS (and even more for HTTP/2), with only 4 ms overhead over vanilla TLS. JANUS-redirection establishes two TLS connections: one to the frontend and one to the backend. As a result, in redirection mode, JANUS’s speedup is smaller: 7.5× over RA+TLS, with a 12 ms overhead over vanilla TLS due to extra RTTs. Similarly, these gains in redirection mode shrink in WAN settings (i.e., RTTs of 40/80/120 ms) also due to the fact that JANUS-redirection needs to establish two TLS connections. Note that, in practice, the frontend-connection cost can be amortized by returning multiple backends and caching them for later connections. In addition, Oak, attesting on every connection, sits with RA+TLS at 126 ms on a LAN and 623 ms at 120 ms RTT. OHTTP+KMS-warm attests nothing in its handshake, and the client verifies nothing on this connection until the sealed response is decrypted. Table 2 reports its time to the first verified byte, 1.5 to 2.2× of JANUS-proxy. Every further request costs 4.6 to 6.5 ms more than JANUS-proxy for the sealing and backend gateway hop. OHTTP+KMS-cold re-attests the backend gateway after a key rotation and costs at least 3.9× of JANUS-proxy. JANUS pays neither cost, as each backend is attested once at registration.

Figure 5 shows the breakdown of client connection establishment latencies under varying RTTs. The higher overheads of RA+TLS and HTTP/2 stem from two factors: the fresh server quote generation (~85 ms), independent of RTT, and the AS-based verification latency, which grows with RTT.

■ TCP+TLS handshake ■ Backend TLS+DC ■ Noise handshake ■ Quote generation
■ Frontend redirect ■ KMS key fetch ■ Sealed request ■ Attestation (MAA)

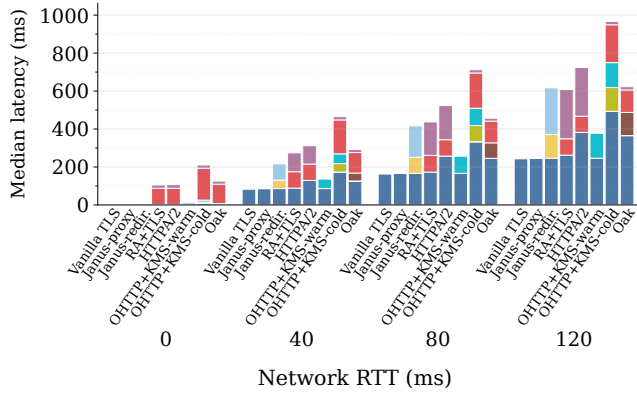


Figure 5. Median client connection establishment latencies for vanilla TLS, JANUS and other baselines over 200 runs.

In contrast, these attestation operations are off the client-connection path in both JANUS modes: quote and JWT generation for the frontend occur at frontend startup, while quote generation and verification for backends occur when they register to the frontend. Consequently, the client connection establishment in JANUS incurs only the TLS-handshake overheads and the local signature checks (i.e., AS signature verification on the JWT, and the DC verification). Due to the AS signing key caching at the client (§4.4), there is no additional request from the client to the AS except for the very first connection; therefore, these client-side checks cost a negligible 0.6-0.8 ms, constant across all RTTs. The client connection establishment latency then grows primarily with RTT in proportion to the number of round trips. Note again that, JANUS-redirection’s extra RTT comes from the initial frontend connection and can be amortized across many backend connections. As for Oak, it incurs a 100 ms quote delay on every connection, whereas OHTTP+KMS-cold adds overheads of the key fetch from the KMS and the backend gateway’s quote verification, on which the sealed request waits. There is no attestation delay for OHTTP+KMS-warm.

AS Signing Key Stability. JANUS’s client-side AS signing key caching (§4.4) assumes infrequent key rotation. A one-month study of three production ASes (i.e., Azure MAA [65], Intel Trust Authority [47], Google Confidential Space [36]) shows zero key changes in Azure MAA and Intel Trust Authority. Google shows an 8-day rotation with a ~19 h dual-validity overlap. A background refresh within this overlap keeps the cache entries fresh, ensuring the AS round trip remains off the critical client-connection path.

Bandwidth Overhead. JANUS adds two static items to each connection: the AS JWT, and an RFC-9345 DC on the handshake. The main bandwidth overhead stems from including the JWT as an X.509 certificate extension (Table 3). Besides

Table 3. Bandwidth overhead of JANUS over TLS.

	Vanilla TLS	JANUS adds	Increase
Certificate (AS JWT)	783 B	+4,697 B	+600%
Handshake (DC)	1,090 B	+175 B	+16%
Total per connection	1,873 B	+4,872 B	+260%

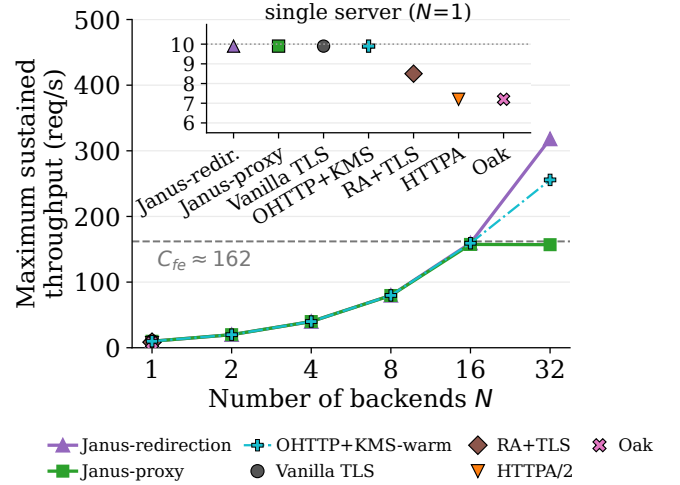


Figure 6. Maximum sustained throughput with a pool of backend servers.

the attestation quote, verification metadata and the AS signature, the JWT includes a key ID, allowing clients to use the corresponding key to verify the signature without contacting the AS again. Note that, the transfer of the attestation quote is inevitable in any approach combining TEEs and TLS.

6.3 Scalability

Scalability of the Backend Pool. A key contribution of JANUS is its ability to serve a confidential cloud service from a pool of load-balanced backend servers under a single attested identity, enabling the service to scale as demand changes. Figure 6 shows the scalability of JANUS and baselines using a synthetic workload where each server can sustain a throughput of 10 req/s with no protocol overhead, defined as the steady-state completion rate. Existing attested-TLS systems, including RA+TLS and HTTP/2, support only the ‘single client-server’ scenario, and do not offer a solution to scaling and load balancing; thus, their maximum sustained throughput is what a single server can offer (i.e., $N=1$, top zoom-in of Figure 6). Due to connection establishment overheads, they can achieve 8.5 and 7.2 req/s, respectively. Similarly, Oak (also a ‘single client-server’ system) stays below the cap, at 7.2 req/s, as each connection blocks on a fresh quote.

In contrast, OHTTP+KMS attests its backend gateway only once per key rotation. As a result, OHTTP+KMS-warm and JANUS-proxy perform the same up to $N=16$, whereas JANUS-proxy reaches its maximum forwarding capacity at

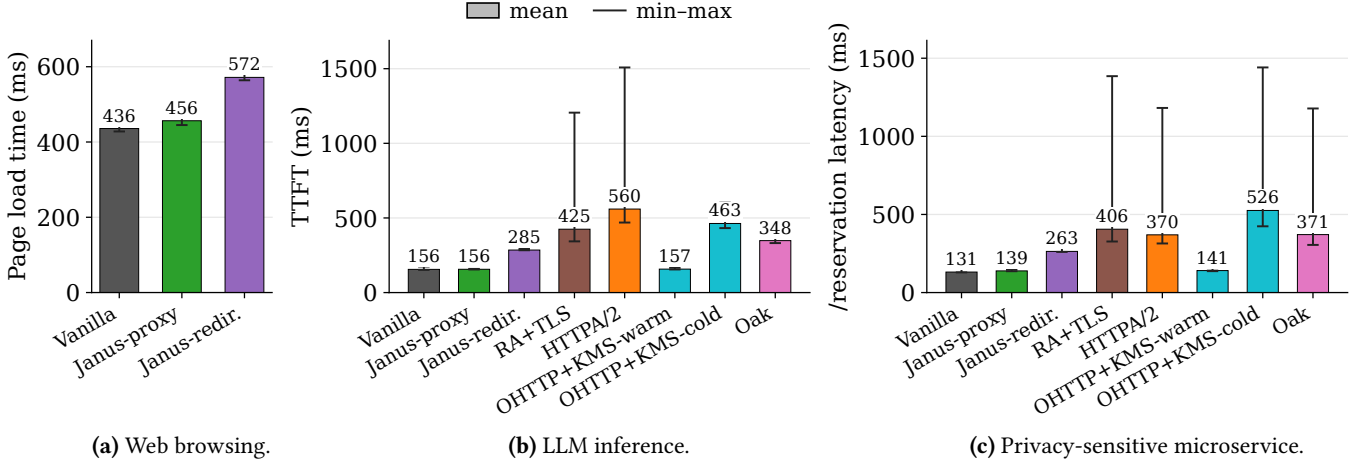


Figure 7. Request latencies on (a) web browsing, (b) LLM inference, and (c) microservice. Web browsing cannot be evaluated for RA+TLS, HTTP/2, OHTTP+KMS and Oak, because none of them can be used by an off-the-shelf browser.

162 req/s and cannot utilize additional backends when $N = 32$. OHTTP+KMS-warm continues to serve more requests with a maximum of 255 req/s at $N = 32$, as its proxy runs similar forwarding logic as in JANUS frontend, but without SGX. Note that, we only show the OHTTP+KMS-warm performance as its steady state is reached between key rotations. In comparison, JANUS-redirection keeps the frontend off the data path and scales proportionally with the backend pool size (318 req/s with $N = 32$).

Note also that, putting an ordinary load balancer in front of the attested-TLS backends in RA+TLS or HTTP/2 would still require each client connection to re-attest its assigned backend, or necessitate each backend to have its own client-facing TLS certificate. In contrast, by signing each registered backend’s public key into a DC under a single frontend identity, JANUS can utilize an elastic backend pool without unnecessary attestation and certificate issuance costs while providing the same guarantees for unmodified TLS clients and browsers. As a result, JANUS enables confidential cloud services to be horizontally scalable.

Channel Throughput. Once established, a JANUS channel is an ordinary TLS channel over a single connection and incurs no steady-state penalty. In contrast, HTTP/2’s nested, doubly-encrypted channel has a $\sim 22\%$ performance overhead (0.10 vs. 0.13 GiB/s on a co-located 1 GiB transfer).

6.4 Applications

We validate the performance of JANUS and other baselines. We use three representative confidential applications and set RTT to 40 ms, a typical client-to-region RTT. Figure 7 reports the first-request latency on a new connection (i.e., connection establishment plus the first request), which captures the minimum case in which the attestation overhead is incurred.

Web Browsing. We measure the page-load time for a confidential Flask [74] application loaded by our Firefox extension.

The client is an off-the-shelf Firefox browser, which performs the DC handshake while our Firefox extension validates the JWT. Importantly, JANUS requires no patch to the browser or TLS library, being compatible with the standard TLS 1.3 stack. In contrast, RA+TLS and HTTP/2 cannot run in an off-the-shelf browser, because they require a custom TLS extension or a post-handshake exchange to establish the connection; similarly, neither OHTTP+KMS nor Oak can run without their own client software. Therefore, these four systems are absent from this experiment. As shown in Figure 7a, JANUS-proxy and JANUS-redirection add 20 ms and 136 ms over vanilla TLS, respectively. JANUS-redirection’s extra latency is mainly caused by its requirement of two client connection establishments.

LLM Inference. For LLM inference, we measure the time-to-first-token (TTFT) of a Llama-3.1-8B-Instruct vLLM [53] on an NVIDIA H100 in TEE mode [70], driven by the ShareGPT [92] prompts. As shown in Figure 7b, JANUS-proxy closely matches the vanilla TLS performance of 156 ms, as its per-connection overhead is negligible compared to the GPU’s pre-fill time. JANUS-redirection takes 285 ms, while Oak, RA+TLS and HTTP/2 exhibit higher latencies at 348, 425 and 560 ms, respectively, as they attest every connection. OHTTP+KMS-warm takes 157 ms, whereas OHTTP+KMS-cold takes 463 ms, with a key refetch and a fresh backend gateway attestation.

Privacy-Sensitive Microservice. We evaluate the DeathStar-Bench [30] HotelReservation benchmark, which consists of 22 services within a CVM. Here, sensitive data (e.g., credentials, booking details) is handled by the /reservation endpoint, which we load. Figure 7c shows that JANUS-proxy takes 139 ms compared to vanilla TLS’s 131 ms. JANUS-redirection takes 263 ms, while Oak, RA+TLS and HTTP/2 take 371, 406 and 370 ms, respectively. OHTTP+KMS-warm takes 141 ms and OHTTP+KMS-cold takes 526 ms.

Across all these applications, JANUS keeps overhead near vanilla TLS by removing quote generation and AS requests from the client-connection path. It also yields predictable latency: the baselines that attest every connection (i.e., RA+TLS, HTTP/2, Oak) or the first request after a key rotation (i.e., OHTTP+KMS-cold) see the worst-case requests of 1.2-1.5 s (Figure 7), whereas JANUS and OHTTP+KMS-warm stay within a few milliseconds of their means.

7 Related Work

To our knowledge, JANUS is so far the only system to present an elastic pool of attested servers under a single TLS identity that unmodified TLS stacks, including browsers, can validate with standard TLS and TEE mechanisms. Here, we compare JANUS to existing work.

Modifying the TLS Handshake. Knauth *et al.* [52] embed a fresh SGX quote in a self-signed certificate at handshake. RATLS [89] extends that solution to TLS 1.3 and other TEEs. RA+TLS [90] runs the attestation exchange during handshake and binds the attestation result to the TLS session’s ephemeral key for failure independence. All these systems attest TEEs *per connection*, while JANUS employs a single persistent attestation *per server instance*, and delegates per-connection authority via a TEE-rooted RFC-9345 DC [16], amortizing remote attestation costs over many connections.

Application-Layer Protocols. This line of work treats attestation as a Layer-7 primitive on HTTPS. HTTP/2 [50] and HTTP/2 [51] run a nested attestation exchange above the TLS channel. RA-WEBs [3] delegates TEE verification to a trusted third-party verifier service to enable browser compatibility. In contrast, JANUS avoids application-layer protocol changes by riding attestation on the TLS certificate path.

Certificate-Extension Proposals. The IETF LAMPS efforts [72, 73] use TEE attestation evidence as a standard X.509 certificate extension. Earlier work binds TPM or enclave identity to the certificate [13, 14, 32]. JANUS follows this path but composes the attestation extension with the RFC-9345 DC DelegationUsage, more directly tying to the load-balancing architecture.

DNS-Based Attestation. aDNS [25] distributes attestation evidence through DNS. In comparison, JANUS distributes attestation evidence through the X.509 certificate extension and binds directly to the existing PKI.

Key-Release Designs and Custom Sessions. In Azure’s confidential inferencing [78], an attested Key Management Service (KMS) releases the rotating OHTTP private key only to backend gateways whose attestation evidence it has verified. The client, after verifying the KMS’s attestation evidence, encrypts every request with OHTTP [83] and Hybrid Public Key Encryption (HPKE) [15]. The client receives an encryption key rather than a TLS identity: it needs its own protocol code and key configuration, its TLS session ends at an untrusted frontend, and replay protection is left to

the application [83]. OHTTP is request-response only, so each message is sealed on its own and an interactive application has to carry its session context in every message. In contrast, JANUS consumes the DC inside the handshake the client already performs, so the session ends inside the TEE, carries any application protocol, and inherits the replay protection from TLS. Oak [34] nests a Noise handshake over TLS using an evidence-backed key that a custom client verifies per server, lacking a unified service identity and precluding transparent load balancing across servers. CCF [79] gives a replicated service a single identity, but a self-signed one that clients pin out of band; its ACME certificates carry no attestation binding.

Confidential Computing Systems. Duet [4] proposes the controller/worker split, with a crude redirection mode to backend servers. CACTEE [5] adapts a similar model with a proxy mode to backends. Both systems use self-signed certificates and require custom clients. Site-isolated TEE designs [76] explore complementary directions in confidential web deployments. The closest in spirit is Edgeless Systems’ Contrast [29], which deploys confidential containers on Kubernetes behind a central coordinator that verifies each pod and issues it a certificate from a per-deployment root CA. Contrast and JANUS share a similar controller/worker pattern, but differ on two axes that matter at deployment time. First, Contrast operates an entire bespoke PKI (i.e., root, intermediate, and mesh CA) inside the deployment, forcing each client to add the root certificate to its trust store out of band. In contrast, JANUS piggybacks on the existing PKI, issuing one CA-signed certificate to the frontend and projecting that identity onto backends via standard RFC-9345 Delegated Credentials, so unmodified TLS stacks of clients can validate the trust chain without any per-deployment trust configuration. Second, Contrast does not specify a load-balancing mechanism for routing client traffic to the verified server pool, whereas JANUS treats load balancing as a first-class concern and supports both proxy and redirection modes under the same attested identity. To our knowledge, JANUS is the first system to combine a TEE-rooted, CA-issued TLS identity with RFC-9345 DCs to support an attested multi-server pool behind a single client-visible service identity.

8 Conclusion

JANUS is the first load-balancing architecture for multi-server confidential cloud services that preserves compatibility with existing TLS and PKI ecosystems. By combining a CA-certified, TEE-attested frontend with the RFC-9345 Delegated Credentials, JANUS presents a single attested service identity across an elastic pool of confidential backend servers. JANUS supports the two standard load-balancing modes (i.e., proxy and redirection), and keeps the per-connection overhead low — within 4 ms and 12 ms of vanilla TLS in proxy mode and redirection mode, respectively.

References

- [1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, et al. 2019. Let's Encrypt: an automated certificate authority to encrypt the entire web. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [2] Advanced Micro Devices. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. AMD White Paper. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf> Accessed on 2026-05-20.
- [3] Kosei Akama, Yoshimichi Nakatsuka, Koray Luke, Masaaki Sato, and Keisuke Uehara. 2024. RA-WEBs: Remote Attestation for WEB services. *arXiv preprint arXiv:2411.01340* (2024).
- [4] Istemi Ekin Akkus and Ivica Rimac. 2024. Duet: Combining a trustworthy controller with a confidential computing environment. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.
- [5] Istemi Ekin Akkus and Ivica Rimac. 2026. CACTEE: Confidential Asset Certification using Trusted Execution Environments. In *Proceedings of the 9th Workshop on System Software for Trusted Execution*.
- [6] Alibaba Cloud. 2026. Build an SGX Encrypted Computing Environment. <https://www.alibabacloud.com/help/en/ecs/user-guide/build-an-sgx-encrypted-computing-environment>. Accessed on 2026-06-10.
- [7] Amazon. 2026. AWS Confidential Computing. <https://aws.amazon.com/confidential-computing/>. Accessed on 2026-06-10.
- [8] Amazon Web Services. 2026. Application Load Balancer Documentation. <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/>. Accessed on 2026-05-07.
- [9] Amazon Web Services. 2026. AWS Clean Rooms. <https://aws.amazon.com/clean-rooms/>. Accessed on 2026-06-10.
- [10] Amazon Web Services. 2026. AWS Nitro Enclaves. AWS Product Page. <https://aws.amazon.com/ec2/nitro/nitro-enclaves/> Accessed on 2026-05-20.
- [11] Apple Security Engineering and Architecture. 2024. Private Cloud Compute: A New Frontier for AI Privacy in the Cloud. Apple Security Research Blog. <https://security.apple.com/blog/private-cloud-compute/> Accessed on 2026-05-20.
- [12] Arm Limited. 2026. Arm Confidential Compute Architecture. Arm Product Page. <https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture> Accessed on 2026-05-20.
- [13] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. 2008. An efficient implementation of trusted channels based on OpenSSL. In *ACM Workshop on Scalable Trusted Computing*.
- [14] Norazah Abd Aziz, Nur Izura Udzir, and Ramlan Mahmod. 2014. Extending TLS with Mutual Attestation for Platform Integrity Assurance. *Journal of Communications* 9, 1 (2014).
- [15] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. 2022. Hybrid Public Key Encryption. RFC 9180. <https://www.rfc-editor.org/rfc/rfc9180>
- [16] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla. 2023. Delegated Credentials for (D)TLS. RFC 9345. <https://www.rfc-editor.org/rfc/rfc9345.html>
- [17] T. Brisco. 1995. DNS Support for Load Balancing. RFC 1794. <https://www.rfc-editor.org/rfc/rfc1794.html>
- [18] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium*. 699–716.
- [19] Pau-Chen Cheng, Kevin Eykholt, Zhongshu Gu, Hani Jamjoom, KR Jayaram, Enriquillo Valdez, and Ashish Verma. 2024. DeTA: Minimizing Data Leaks in Federated Learning via Decentralized and Trustworthy Aggregation. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 219–235.
- [20] Confidential Computing Consortium. 2022. Confidential computing: Hardware-based trusted execution for applications and data. *Confidential Computing Consortium Technical Report v1.3* (2022).
- [21] Confidential Computing Consortium. 2022. A Technical Analysis of Confidential Computing. *Confidential Computing Consortium Technical Report v1.3* (2022).
- [22] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. 2024. Secret Key Recovery in a Global-Scale End-to-End Encryption System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 703–719.
- [23] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).
- [24] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *IEEE Symposium on Security and Privacy (S&P)*.
- [25] Antoine Dignat-Lavaud, Cédric Fournet, Kapil Vaswani, Manuel Costa, Sylvan Clebsch, and Christoph M Wintersteiger. 2025. Transparent Attested DNS for Confidential Computing Services. In *USENIX Security Symposium*.
- [26] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Michael O'Connor, Rob Nertney, and Phil Rogers. 2023. Creating the First Confidential GPUs. *ACM Queue* 21, 4 (2023), 68–93.
- [27] DigiCert. 2026. DigiCert. <https://www.digicert.com/>. Accessed on 2026-06-10.
- [28] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. 2002. Globally Distributed Content Delivery. *IEEE Internet Computing* 6, 5 (2002), 50–58.
- [29] Edgeless Systems. 2026. Contrast: Deploy and manage confidential containers on Kubernetes. Project documentation. <https://docs.edgeless.systems/contrast/> Accessed on 2026-05-24.
- [30] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rath, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ACM international conference on architectural support for programming languages and operating systems (ASPLOS)*.
- [31] GlobalSign. 2026. GlobalSign. <https://www.globalsign.com/>. Accessed on 2026-06-10.
- [32] Kenneth Goldman, Ronald Perez, and Reiner Sailer. 2006. Linking remote attestation to secure tunnel endpoints. In *ACM Workshop on Scalable Trusted Computing*.
- [33] Google. 2026. BoringSSL. <https://boringssl.googlesource.com/boringssl/>. Accessed on 2026-05-07.
- [34] Google. 2026. Project Oak: Meaningful Control of Data in Distributed Systems. <https://github.com/project-oak/oak>. Accessed on 2026-09-15.
- [35] Google Cloud. 2026. Confidential Computing. <https://cloud.google.com/security/products/confidential-computing>. Accessed on 2026-06-10.
- [36] Google Cloud. 2026. Google Cloud Attestation. <https://cloud.google.com/confidential-computing/confidential-vm/docs/attestation>. Accessed on 2026-06-10.
- [37] Gramine Project. 2026. Gramine. <https://gramineproject.io/>. Accessed on 2026-06-10.
- [38] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference (ATC)*. 299–312.
- [39] Vivek Haldar, Deepak Chandra, and Michael Franz. 2004. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, Vol. 2004. USENIX Association.
- [40] HAProxy Technologies. 2026. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. <https://www.haproxy.org/>. Accessed on 2026-05-07.

- [41] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. 2020. Telekine: Secure computing with cloud GPUs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 817–833.
- [42] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient deep learning on multi-source private data. *arXiv preprint arXiv:1807.06689* (2018).
- [43] Intel Corporation. 2025. Affected Processors: Guidance for Security Issues on Intel Processors. <https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html>. Accessed on 2026-06-10.
- [44] Intel Corporation. 2026. Intel SGX Data Center Attestation Primitives (Intel SGX DCAP). <https://www.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-dcap-ecdsa-orientation.pdf>. Accessed on 2026-06-10.
- [45] Intel Corporation. 2026. Intel Software Guard Extensions (Intel SGX). Intel Product Page. <https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html>. Accessed on 2026-05-20.
- [46] Intel Corporation. 2026. Intel TDX DCAP: Quote Generation Library and Quote Verification Library. https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_TDX_DCAP_Quoting_Library_API.pdf. Accessed on 2026-06-10.
- [47] Intel Corporation. 2026. Intel® Trust Authority: Introduction. <https://docs.trustauthority.intel.com>. Accessed on 2026-06-10.
- [48] Intel Corporation. 2026. Intel Trust Domain Extensions (Intel TDX). Intel Product Page. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html>. Accessed on 2026-05-20.
- [49] Kevin Jacobs, J.C. Jones, and Thyra van der Merwe. 2019. Validating Delegated Credentials for TLS in Firefox. Mozilla Security Blog. <https://blog.mozilla.org/security/2019/11/01/validating-delegated-credentials-for-tls-in-firefox/>. Accessed on 2026-05-07.
- [50] Gordon King and Hans Wang. 2023. Httpa: Https attestable protocol. In *Future of Information and Communication Conference*.
- [51] Gordon King and Hans Wang. 2023. HTTP/2: A trusted end-to-end protocol for Web services. In *Future of Information and Communication Conference*.
- [52] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).
- [53] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.
- [54] B. Laurie, A. Langley, and E. Kasper. 2013. Certificate Transparency. RFC 6962. <https://www.rfc-editor.org/rfc/rfc6962.html>
- [55] B. Laurie, E. Messeri, and R. Stradling. 2021. Certificate Transparency Version 2.0. RFC 9162. <https://www.rfc-editor.org/rfc/rfc9162.html>
- [56] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip attack on hardware enclaves via the memory bus. In *USENIX Security Symposium*.
- [57] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*. 557–574.
- [58] Let’s Encrypt. 2025. Pebble: A miniature ACME test server. <https://github.com/letsencrypt/pebble>. Version 2.9.0; accessed on 2026-06-10.
- [59] Let’s Encrypt. 2026. Let’s Encrypt. <https://letsencrypt.org/>. Accessed on 2026-06-10.
- [60] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A systematic look at ciphertext side channels on AMD SEV-SNP. In *IEEE Symposium on Security and Privacy (S&P)*. 337–351.
- [61] Linux man-pages. 2026. tc-netem(8) — Linux manual page. <https://man7.org/linux/man-pages/man8/tc-netem.8.html>. Accessed on 2026-06-10.
- [62] Moxie Marlinspike. 2017. Technology Preview: Private Contact Discovery for Signal. Signal Blog. <https://signal.org/blog/private-contact-discovery/>. Accessed on 2026-05-20.
- [63] Meta. 2026. Fizz: A performant TLS 1.3 implementation. <https://github.com/facebookincubator/fizz>. Accessed on 2026-05-07.
- [64] Microsoft. 2026. Azure Confidential Computing. <https://learn.microsoft.com/en-us/azure/confidential-computing/overview>. Accessed on 2026-06-10.
- [65] Microsoft Corporation. 2026. Microsoft Azure Attestation. <https://learn.microsoft.com/azure/attestation/>. Accessed on 2026-06-10.
- [66] Mozilla. 2026. NSS Delegated Credentials Implementation. https://bugzilla.mozilla.org/show_bug.cgi?id=1562770. Accessed on 2026-05-07.
- [67] 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 (S&P)*. 1466–1482.
- [68] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [69] NGINX Inc. 2026. nginx: HTTP and reverse proxy server, mail proxy server, and generic TCP/UDP proxy server. <https://nginx.org/>. Accessed on 2026-05-07.
- [70] NVIDIA. 2025. NVIDIA Confidential Computing. <https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/>. Accessed on 2026-05-10.
- [71] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party machine learning on trusted processors. In *USENIX Security Symposium*. 619–636.
- [72] Mike Ounsworth, Hannes Tschofenig, Henk Birkholz, Monty Wiseman, and Ned Smith. 2026. *Use of Remote Attestation with Certification Signing Requests*. Internet-Draft. <https://datatracker.ietf.org/doc/draft-ietf-lamps-csr-attestation/>. Work in progress; accessed on 2026-06-10.
- [73] Mike Ounsworth, Monty Wiseman, Hannes Tschofenig, Tirumaleswar Reddy, and Ned Smith. 2025. *X.509 Certificate Extensions for Attestation Results*. Internet-Draft. <https://datatracker.ietf.org/doc/draft-ounsworth-lamps-x509-ar/>. Work in progress; accessed on 2026-06-10.
- [74] Pallets. 2026. Flask. <https://flask.palletsprojects.com/en/stable/>. Accessed on 2026-06-10.
- [75] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. 2011. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy (S&P)*. 379–394.
- [76] Luca Preibsch, Maxim Ritter von Onciul, and Rüdiger Kapitza. 2025. Site Attestation: Browser-based Remote Attestation. In *ACM European Workshop on Systems Security*.
- [77] E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. <https://www.rfc-editor.org/rfc/rfc8446.html>
- [78] Mark Russinovich. 2024. Azure AI Confidential Inferencing: Technical Deep-Dive. Microsoft Tech Community Blog. <https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/azure-ai-confidential-inferencing-technical-deep-dive/4253150>. Accessed on 2026-05-20.
- [79] Mark Russinovich, Edward Ashton, Christine Avanesians, Miguel Castro, Amaury Chamayou, Sylvain Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Magalhaes, Kartik Mehta, Sneha Nijhawan, Junji Nishihara, et al. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services*.

- Technical Report MSR-TR-2019-16. Microsoft Research.
- [80] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: using malicious #VC interrupts to break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy (S&P)*. 4220–4238.
 - [81] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*.
 - [82] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the perils of Security-Oblivious energy management. In *USENIX Security Symposium*. 1057–1074.
 - [83] Martin Thomson and Christopher A. Wood. 2024. Oblivious HTTP. RFC 9458. <https://www.rfc-editor.org/rfc/rfc9458>
 - [84] Anna Trikalinou and Dan Lake. 2017. Taking DMA attacks to the next level. *BlackHat USA (2017)*, 22–27.
 - [85] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. 2004. Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. RFC 3820. <https://www.rfc-editor.org/rfc/rfc3820.html>
 - [86] 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*.
 - [87] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*.
 - [88] Daan Vanoverloop, Andres Sanchez, Flavio Toffalini, Frank Piessens, Mathias Payer, and Jo Van Bulck. 2025. TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms. In *USENIX Security Symposium*.
 - [89] Robert Walther, Carsten Weinhold, and Michael Roitzsch. 2022. RATLS: Integrating transport layer security with remote attestation. In *International Conference on Applied Cryptography and Network Security*. Springer.
 - [90] Carsten Weinhold, Muhammad Usama Sardar, Ionuț Mihalcea, Yogesh Deshpande, Hannes Tschofenig, Yaron Sheffer, Thomas Fossati, and Michael Roitzsch. 2025. Separate but Together: Integrating Remote Attestation into TLS. In *USENIX Annual Technical Conference (ATC)*.
 - [91] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)*. 640–656.
 - [92] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. *arXiv preprint arXiv:2309.11998* (2023).