LAMBADA: Autoscaling Confidential Function Chains without Centralized Trust

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

Matthew Berthoud
mwberthoud@wm.edu
William & Mary
Williamsburg, Virginia, United States

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

Abstract

Function-as-a-Service (FaaS) platforms enable developers to build event-driven applications by composing lightweight functions into complex workflows. Many such workflows process sensitive data, raising confidentiality risks from untrusted cloud providers and compromised infrastructure. Existing confidential FaaS systems mitigate these risks with secure hardware enclaves (e.g., Intel SGX) but still depend on a trusted key-provisioning enclave for autoscaling, creating a single point of compromise.

We present LAMBADA, a confidential FaaS platform that eliminates centralized key management while preserving autoscaling. LAMBADA extends the Knative serverless framework and runs each function instance inside an Intel SGX enclave that independently generates its own keys. Using proxy re-encryption (PRE), the platform safely routes data to replicas without exposing plaintexts. LAMBADA treats attested enclaves as principals and PRE keys as capabilities, enforcing workflow policies that govern who may decrypt data, and in what order—even under an untrusted FaaS platform. Our prototype shows that LAMBADA adds only $1.55 - 1.83\times$ latency compared to a centralized confidential baseline with a shared RSA key, while retaining full autoscaling support.

CCS Concepts

• **Security and privacy** → **Key management**; • **Networks** → **Cloud computing**.

Keywords

Confidential Computing, Cloud Functions, Proxy Re-encryption

ACM Reference Format:
Aashutosh Poudel, Matthew Berthoud, and Stephen Herwig. 2026. LAMBADA: Autoscaling Confidential Function Chains without Centralized Trust. In *Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT '26), July 08–10, 2026, Waterloo, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3750555.3811894

1 Introduction

Function-as-a-Service (FaaS) platforms like AWS Lambda are popular choices for building event-driven, cloud-based solutions because of their autoscaling capabilities and pay-per-use billing. The stateless nature and loose coupling of functions allow organizations to create complex workflows simply by linking multiple functions in a sequence, or *function chain*. A recent study found that 46% of real-world FaaS applications use function chains, with some chains containing up to 10 functions [74]. Although functions can directly invoke each other to form chains, they more commonly rely on intermediary services—such as event queues, pub/sub channels, or object stores—to trigger and pass control to the next function.

Many use cases—especially in highly-regulated sectors like government, finance, and healthcare—require function chains to process sensitive data. For instance, a loan application chain might include: *(i)* a bank function to process the application; *(ii)* a function to verify the applicant's identity; and *(iii)* a function to retrieve the applicant's credit score. Even with trusted cloud providers, customers must remain wary of infrastructure bugs [39], insider threats [73], and data disclosures to law enforcement [1–3] that could expose this data. Such threats do not have to directly access the function data; prior research shows that attackers can simply manipulate data flows to steal sensitive data and conduct stealthy operations [5].

Efforts to secure function chains ensure either data confidentiality, or integrity of function order [32, 47]. For confidentiality, prior research systems [7, 25, 43, 49, 53, 80, 85] use hardware *trusted execution environments (TEEs)*, particularly Intel SGX enclaves [30, 64], to protect function workflows. However, to support autoscaling, these systems rely on a centralized key distribution enclave to provision identical keying material to each function replica. While this design simplifies key management, it introduces a single point of failure for undermining the security of the entire system. Similar risks appear in non-confidential microservice deployments; for instance, vulnerabilities in a cluster's local certificate authority have allowed the compromise of entire microservice applications [31].

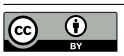
In this paper, we ask the following research question:

Is it possible for a FaaS platform to guarantee the confidentiality and integrity of function chains without resorting to centralized trusted services?

We present LAMBADA, the first FaaS platform to guarantee data confidentiality and execution path integrity without relying on the Fass platform's trustworthiness. LAMBADA combines non-interactive cryptographic protocols with TEEs to secure function chains without centralized trust. In LAMBADA, each function replica runs in a TEE and independently generates its own keys. Using proxy re-encryption, LAMBADA enables any replica to decrypt messages intended for its target function, eliminating the need for a trusted key escrow. For path integrity, LAMBADA cryptographically signs each function's provenance, removing the need for a global controller.

Contributions. We make the following contributions:

- We introduce LAMBADA, a FaaS platform for ensuring the confidentiality and integrity of FaaS pipelines. LAMBADA composes



SACMAT '26, Waterloo, ON, Canada

ACM ISBN 979-8-4007-2107-6/2026/07
https://doi.org/10.1145/3750555.3811894

Intel SGX enclaves with cryptographic protocols for proxy re-encryption to protect inter-function I/O while preserving horizontal scalability.
- We implement a prototype of Lambada in the popular Knative FaaS framework. For this prototype, we evaluate several proxy re-encryption and signature schemes, and demonstrate their transparent integration with FaaS workflows.
- We conduct an evaluation of Lambada in a Kubernetes cluster using a combination of stress tests and macrobenchmarks, and observe a modest 1.55 − 1.83× performance overhead compared to centralized approaches using a shared RSA key.

2 Background

In this section, we briefly discuss the trusted hardware concepts and cryptographic schemes we use in Lambada. Table 1 summarizes their Lambada-relevant APIs.

Intel SGX. Intel SGX [44, 64] extends the x86-64 instruction set to create TEEs, called *enclaves*, that allow user-space applications to run in isolation from potentially malicious privileged software, such as the operating system or hypervisor, and from certain physical attacks. SGX enforces memory isolation by encrypting and integrity-protecting enclave memory in DRAM and decrypting it only within the trusted CPU package, preventing privileged components from observing or modifying code and data—though side-channel attacks can weaken these guarantees [24, 26, 52, 54, 55, 82, 83]. By supporting enclaves at sub-process granularity, SGX reduces the trusted computing base but limits compatibility with legacy software, though several middleware solutions [10, 13, 42, 81] support running unmodified applications.

A fundamental feature of Intel SGX is attestation [9], which enables verification of an enclave's integrity and the hardware's authenticity. During attestation, the hardware root of trust signs the enclave's initial measurement (`MRENCLAVE`) and the hash of the enclave signer's public key (`MRSIGNER`), producing an *attestation report* (or *quote*). To bind runtime data—such as a public key—to the attestation, the enclave can embed a small, opaque *user-data* field that is also covered by the quote's signature. Clients or tenants can then verify the quote to ensure that the expected software is executing within a genuine enclave.

Proxy re-encryption. *Proxy re-encryption (PRE)* enables an untrusted proxy to transform a ciphertext encrypted under Alice's public key into one that Bob can decrypt with his private key—without revealing the underlying plaintext to the proxy. At a high level, Alice and Bob generate a public *re-encryption key* $\mathsf{rk}_{\text{Alice}\to\text{Bob}}$ which the proxy uses to perform this transformation from Alice (the delegator) to Bob (the delegatee).

In 1998, Blaze, Bleumer, and Strauss [17] introduced the first PRE scheme, which was based on the the ElGamal encryption system [37]. However, it required Bob to share his private key with Alice, and produced a *bidirectional* re-encryption key, allowing re-encryption in both directions ($\mathsf{rk}_{\text{Alice}\leftrightarrow\text{Bob}}$). To address these limitations, Dodis and Ivan [46] proposed a *unidirectional* variant, and later, Ateniese et al. [11] introduced *non-interactive* schemes using bilinear maps [21] that eliminated the need for interaction between delegator and delegatee. Subsequent research has explored advanced features such as *multi-use* re-encryption (where ciphertexts can be re-encrypted multiple times) [67], revocation [72], and stronger security guarantees, like *chosen-ciphertext security* [28, 59]. and ciphertext *unlinkability* [33].

Table 1: API summary for Intel SGX attestation, proxy re-encryption (PRE), and aggregate signatures.

Intel SGX
SGX.GetQuote(userData) → quote On input userData ∈ $\{0,1\}^{512}$, outputs a quote signed by the enclave platform for use in remote attestation.
SGX.VerifyQuote(quote) → true | false On input a quote, outputs true if the quote was signed by the enclave platform manufacturer. The caller must validate the actual content of the quote.
Proxy Re-Encryption†
PRE.KeyGen(1^λ) → pk, sk On input the security parameter 1^λ, outputs a public key pk and secret key sk.
PRE.ReKeyGen($\mathsf{sk}_1, \mathsf{pk}_2$) → $\mathsf{rk}_{1\to2}$ On input a secret key sk_1 and public key pk_2, outputs a re-encryption key $\mathsf{rk}_{1\to2}$.
PRE.Encrypt(pk, m) → c On input a public key pk and message $m \in \{0,1\}^*$, outputs a ciphertext c.
PRE.ReEncrypt($\mathsf{rk}_{1\to2}, c_1$) → c_2 On input a re-encryption key $\mathsf{rk}_{1\to2}$ and ciphertext c_1 encrypted with pk_1, outputs a second ciphertext c_2 decryptable by sk_2.
PRE.Decrypt(sk, c) → m On input a secret key sk and ciphertext c, outputs plaintext m.
Aggregate Signatures
AggSig.KeyGen(1^λ) → pk, sk On input the security parameter 1^λ, outputs a public (verification key) pk and secret (signing) key sk.
AggSig.Sign($sk, m, [\sigma^*]$) → σ^* On input a secret key sk, a message $m \in \{0,1\}^*$, and an optional aggregate signature σ^*, produces a new aggregate signature that includes the signature of sk on m.
AggSig.Agg($\sigma_1, \ldots, \sigma_n$) → σ^* On input a set of signatures σ_i, aggregates the signatures into a single signature σ^*.
AggSig.Verify($\mathsf{pk}_1, \ldots \mathsf{pk}_n, m_1, \ldots m_n, \sigma^*$) → true | false On input public keys pk_i, messages m_i, and an aggregate signature σ^*, returns true if each m_i was signed with its corresponding private key sk_i, and σ^* is the aggregation of the resultant signatures σ_i.

†This defines a generic PRE API; specific schemes may differ slightly. For example, single-use schemes may provide separate decryption algorithms for original and re-encrypted ciphertexts.

Aggregate signatures & multisignatures. In addition to protecting the confidentiality of function data, Lambada also ensures the integrity of the function chain's execution order, which we term *path integrity*. A straightforward approach is for the function chain author to sign and publish the correct function sequence(s), and embed the corresponding public key into each function. At runtime, the functions create a signature chain, where each function signs over its index as well as the prior signatures, and verifies the current sequence against the published reference. Unfortunately,

Table 2: Attacker Capabilities with respect to STRIDE. $\mathcal{A}_{\text{plat\&func}}$ has all capabilities of $\mathcal{A}_{\text{plat}}$.

Capability	$\mathcal{A}_{\text{plat}}$	$\mathcal{A}_{\text{plat\&func}}$
Spoof	N/A	Implied by InfoDisclose
Tamper	Inject and redirect messages; modify function order; modify `etcd` entries.	
Repudiate	Replay messages	
InfoDisclose	N/A	(non-persistently) leak secrets from a function instance
DenyService	Out of scope	
ElevatePriv	Implies all other capabilities not N/A or out of scope	

this leads to linear overhead in signature size and verification time (this approach is also incomplete with respect to replay attacks; see §4.4). Instead, LAMBADA explores sublinear alternatives by implementing and evaluating two signature compression schemes: *aggregate signatures* and *multisignatures*.

In an aggregate signature scheme [22, 61, 62], each private key sk_i signs a distinct message m_i, and anyone can compress the resulting signatures σ_i into a single compact signature σ^*. This aggregate signature allows a verifier to confirm that each signer signed their respective message. In contrast, a multisignature scheme [15, 18, 20, 23, 35, 63, 66] requires all signers to jointly sign the same message, producing a single signature of constant size. While multisignatures typically attest only to the set of signers, some also preserve signer order [19], making them suitable for verifying execution paths.

3 Goals and Assumptions

We present our threat model using the STRIDE threat modeling framework [50] that Microsoft developed to guide their product security, and which cloud native projects like the Istio service mesh use in their own threat modeling documentation [56]. STRIDE is an acronym for **S**poofing, **T**ampering, **R**epudiation, **I**nformation disclosure, **D**enial of service, and **E**levation of privileges, and emphasizes modeling in terms of these specific risks (see Table 2).

System principals. A FaaS platform is composed of: *(1)* a *control plane*—controllers, webhooks, and the autoscaler—that manages deployments, and *(2)* a *data plane*—ingress gateway and message broker—that carries user traffic and events. The platform is hosted on an orchestrator such as Kubernetes, whose control plane includes the API server and `etcd` key-value store. We consider the entire FaaS platform untrusted—including both control plane and data plane—except for the attested enclaves that the tenant deploys as part of LAMBADA (function instances and optional ingress gateway).

Security goals. LAMBADA provides strong security even when the FaaS platform is persistently compromised. Specifically, it guarantees:

S1 *End-to-end payload confidentiality and integrity:* An attacker cannot learn the underlying plaintext of messages between functions and cannot modify a message without detection.

S2 *Path integrity:* An attacker cannot cause the flow of data to follow an unauthorized sequence of functions without detection.

If one or more function instances is compromised, LAMBADA cannot uphold S1. Our objective is then *rapid recovery*: once the compromise ends, key rotation restores confidentiality so leaked keys cannot decrypt future payloads. Additionally, although such an attack can exfiltrate data, LAMBADA still seeks to preserve S2—preventing unauthorized flows *among* deployed functions.

Adversary classes. We distinguish two attacker models, both assuming a persistently compromised FaaS platform:

$\mathcal{A}_{\text{plat}}$ *Persistent FaaS platform compromise:* The attacker $\mathcal{A}_{\text{plat}}$ can observe or alter network traffic, insert/remove/reorder functions, control the message broker, and influence function scheduling and placement. Additionally, $\mathcal{A}_{\text{plat}}$ can read, modify, delete, or rollback values in the `etcd` key-value store. LAMBADA must maintain S1 and S2 against $\mathcal{A}_{\text{plat}}$.

$\mathcal{A}_{\text{plat\&func}}$ *Persistent FaaS platform compromise, occasional function leak:* $\mathcal{A}_{\text{plat\&func}}$ extends $\mathcal{A}_{\text{plat}}$ with the ability to *non-persistently* compromise a function instance (e.g., via a bug or side channel) and exfiltrate its key material. Continuous compromise of every instance is out of scope. During such a leak, S1 may be violated, but upon key rotation all old keys are revoked and erased, preventing the attacker from decrypting future traffic. We assume that functions are approximately time-synchronized and can securely delete their old keys.

Out of scope. We exclude: *(i)* implicit data-flow leaks from input-dependent routing, *(ii)* metadata leaks such as size or timing, whose defense requires anonymous-communication techniques [29, 34], and *(iii)* availability attacks (an attacker who breaches the FaaS platform can always drop or delay requests).

4 Design

We now incrementally develop LAMBADA's protocol, starting from a basic proxy re-encryption model and extending it to support path integrity, replay detection, and key rotation. Algorithms 1–3 present pseudocode for the complete protocol under $\mathcal{A}_{\text{plat\&func}}$. The pseudocode uses a small set of helper functions listed in Table 3 for *authenticated encryption with associated data (AEAD)* (e.g., AES-GCM) [14, 16, 70], as well as the message broker and registry service.

4.1 Overview

Figure 1 presents a high-level diagram of LAMBADA's architecture. At startup, each function instance generates a PRE key pair, and binds the corresponding public key to the instance's *attestation quote* as *user-data*. Each instance is identified by a *function ID* (f_{id})—its `MRENCLAVE` value, shared by all replicas of the same function—and a *replica ID* (r_{id}), a unique randomly generated identifier for that specific instance. The instance then publishes its quote and public key to the registry as a record (rec).

We apply a proxy re-encryption scheme among a group of function replicas within a function chain. We elect one replica as the *leader*, while the other replicas join as *followers*. A leader serves

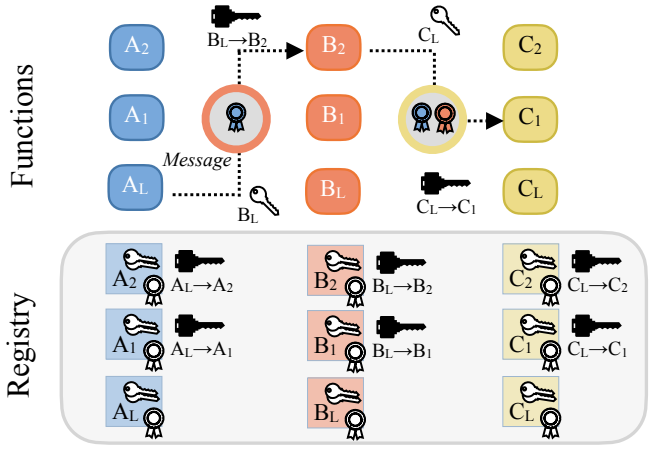


Figure 1: Lambada architecture. The registry holds attested signature verification and encryption keys, as well as re-encryption keys. Each sending function encrypts its message to the next hop's leader; the re-encryption keys allow any replica for the next hop to decrypt the data.

Table 3: API summary for authenticated encryption with associated data (AEAD), and select cloud services.

Authenticated Encryption with Associated Data
AEAD.KeyGen(1^λ) → k
On input the security parameter 1^λ, outputs a secret key k.
AEAD.Encrypt(k, nonce, m, aad) → c
On input a secret key k, nonce $\in \{0,1\}^\ell$, message $m \in \{0,1\}^*$, and additional authenticated data aad $\in \{0,1\}^*$, outputs ciphertext c.
AEAD.Decrypt(k, nonce, c, aad) → m | ⊥
On input a secret key k, nonce, ciphertext c, and additional authenticated data aad, outputs the plaintext m, or ⊥ on error.
FaaS Platform Services
Broker.Post(id_{dst}, m)
On input the destination function ID id_{dst} and message m, posts a message to the broker to deliver the message.
Orchestrator Services
Registry.Put(key, value)
On input a key $\in \{0,1\}^*$ and value $\in \{0,1\}^*$, publishes the value in the registry under the given key.
Registry.Get(key) → value | ⊥
On input a key $\in \{0,1\}^*$, returns the associated value from the registry, or ⊥ if the key does not exist.
Registry.List(keyPath) → subkeys | ⊥
On input a keyPath $\in \{0,1\}^*$, returns the set of subkeys under that path, or ⊥ if keyPath does not exist.

as a delegator and creates the re-encryption keys for the follower replicas, which it publishes to the registry. To ensure that a leader A_L only creates re-encryption keys for its replicas, A_L consults the registry for the attestation of each A_i replica, and checks that the `MRENCLAVE` value in A_i's attestation matches its own.

A message intended for a function is encrypted with the leader's public key. For efficiency, Lambada applies a hybrid-encryption scheme through a KEM-DEM construction: as a *key encapsulation mechanism (KEM)*, the public PRE key encrypts a symmetric key for an AEAD scheme. As the *data encapsulation mechanism (DEM)*, the AEAD key encrypts the actual message.

To decouple sending from receiving, we assume the instance posts the message to a broker—which eventually delivers it to the downstream function—rather than directly invoking the downstream function. When posting the message, the sender addresses it to the f_{id} of the downstream function. If the downstream leader receives the message, it can trivially read the message with its own key. If the FaaS platform instead delivers the message to a follower replica, the follower uses its re-encryption key to re-encrypt the message (intended for the leader), decrypts it with its own key, and processes the message. This allows all the replicas of a function to be represented by a single public key (the leader's), while allowing the platform to deliver a message to any replica of the function.

4.2 Transitive Attestation

The client sends its request over an attested TLS connection to a Lambada ingress gateway running in an enclave. Upon receipt, the gateway looks up the attested public key of the initial leader replica (A_L) in the registry, verifies the attestation, and encrypts the request to the bound public key. When a replica of A encrypts its output, it consults the registry to verify the downstream leader's (B_L) attestation. A then encrypts its output to B_L's attested public key. Thus, *attestation is transitive*: if the client trusts the gateway's attestation, it trusts that the gateway delivers the request to a trusted function A, which delivers the request to a trusted B, and so on.

4.3 Path Integrity

To enforce correct function sequences, Lambada extends the basic protocol with techniques that vary by threat model. In both models, we assume the function owner (the customer deploying the function chain) publishes the signed set of valid call graphs (each node is the `MRENCLAVE` for that function), and embeds the verification key in each enclave image. The function instance (or its sidecar; see §6) downloads and verifies the call graphs at runtime.

Path integrity under $\mathcal{A}_{\text{plat}}$. Under $\mathcal{A}_{\text{plat}}$, where function instances are assumed invulnerable, the attacker can only disrupt path integrity by injecting new messages or redirecting messages.

To prevent message injection, each function instance self-generates a conventional signing key (e.g., RSA or Ed25519) and includes the corresponding public key in its attestation quote as additional user-data. When publishing its quote and PRE public key to the registry, the function also publishes this verification key. Each function then signs its message and includes the signature in the encrypted payload. On receipt of a message, the destination function verifies the message was signed by one of its predecessor functions with respect to the valid call graphs.

To prevent message redirection (perhaps to another function in conformance with the signed call graphs), the sender also includes the destination enclave's f_{id} as *additional authenticated data* (AAD) in the AEAD encryption. If $\mathcal{A}_{\text{plat}}$ redirects the message to a different function, that destination function will use its own f_{id} for the AAD, and the message will fail authentication.

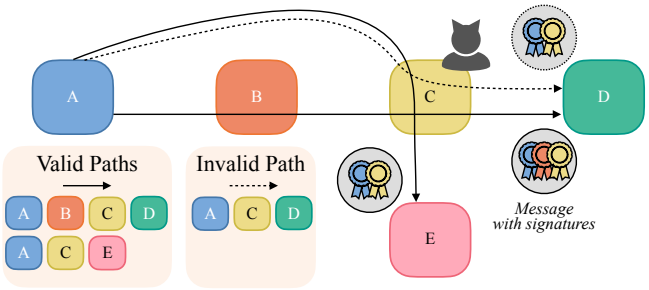


Figure 2: Path integrity. An attacker $\mathcal{A}_{\text{plat\&func}}$ that has captured a message from A to C that is en route to E can instead encrypt the message to D, which would be an invalid path ($A \to C \to D$). D detects the invalid path by verifying the signature of each function that processed the message.

Path integrity under $\mathcal{A}_{\text{plat\&func}}$. Under $\mathcal{A}_{\text{plat\&func}}$, where the attacker can leak a function's keying material, the defense used for $\mathcal{A}_{\text{plat}}$ is insufficient. As Figure 2 shows, if an attacker compromises a function, it can sign and send a message to a destination function outside the valid *end-to-end* call graph. To enable the destination to detect such invalid flows, each message must carry signatures from all preceding functions in the encrypted payload.

With traditional schemes such as Ed25519 or RSA, each function includes all prior signatures in its message. With aggregate signatures, each function signs its position together with a shared nonce and aggregates its signature with the existing ones. In an ordered multisignature scheme, the first function signs a nonce, and subsequent functions co-sign it, producing a single compact joint signature.

4.4 Replay Detection

The path-integrity design does not yet address replay detection. To prevent replays, the LAMBADA ingress gateway assigns each incoming message a unique identifier, flow_{id}, which—together with epoch e (§4.5)—serves as the nonce referenced in §4.3. As each instance processes the flow, it writes `/messages/`flow_{id}`/`f_{id} to the registry with value true. Each hop performs this write as a conditional transaction (failing if the key already exists), ensuring concurrent replays cannot both succeed.

Under $\mathcal{A}_{\text{plat}}$, the destination function detects replays by checking whether `/messages/`flow_{id}`/`f_{dst} already exists in the registry. Under $\mathcal{A}_{\text{plat\&func}}$, the destination function lists all entries prefixed with `/messages/`flow_{id}`/` and checks that this set of f_{id}s precisely matches the set for the received message's signature. This set validation is necessary because a compromised function may not only replay a message that it previously delivered to the destination, but it may also duplicate a message and send it to multiple functions.

Tamper-evident log. The replay-detection mechanism above relies on registry entries remaining intact, but an adversary ($\mathcal{A}_{\text{plat}}$ or $\mathcal{A}_{\text{plat\&func}}$) could tamper or delete entries under `/messages/`flow_{id}. To prevent this, we introduce a tamper-evident log realized as a hash chain whose entries are persisted in the registry: the registry (`etcd`) provides availability, signatures provide integrity, and the chain enforces a write-once, append-only property.

Head	`lambada/audit/head`
$H_{idx} = (idx,\ D_{idx})_\sigma$	
Entry	`lambada/audit/entry/{idx}`
$E_{idx} = (idx,\ D_{idx-1},\ key,\ H(value))_\sigma$	
Record	`{key}`
$key \mapsto (idx,\ key,\ value)_\sigma$	

D_{idx}: digest at index idx; $(\cdot)_\sigma$: signed by the function's private key.

Figure 3: Hash-chain record types for the tamper-evident log.

The function owner initializes the log by generating and signing a unique genesis hash; each function embeds the signed hash and corresponding public key at deployment. On startup, a function verifies the genesis hash and uses it as the root to validate the chain.

To append a key-value pair, a function *(i)* fetches and verifies the current head, *(ii)* signs a new entry, and *(iii)* atomically commits the entry, updated head, and key-value pair in one registry transaction. The log covers entries under `/messages/` and, for forward secrecy, epoch-pointer updates (§4.5). Figure 3 summarizes the three records.

To avoid redundant work, instances use SGX to seal the verified head to their function ID, allowing new replicas to resume from that state rather than re-validating the entire chain.

4.5 Key Rotation

Under the $\mathcal{A}_{\text{plat\&func}}$ threat model, an attacker may transiently leak data from a function instance. To limit the impact, LAMBADA provides forward secrecy for new traffic: once keys rotate, previously leaked keys cannot decrypt future messages. LAMBADA enforces this with a unified epoch-based rotation mechanism in which every ciphertext, re-encryption key, and path signature is bound to an epoch value e. Specifically:

(1) Each attestation embeds e in its user data.
(2) The AEAD (DEM) layer includes e in its associated data.
(3) Each signature covers both the flow ID flow_{id} and e.

When publishing attestation quotes and public keys, instances store them in the registry under a hierarchical path that includes the epoch. The leader publishes the current epoch under a well-known *epoch pointer* path in the registry. Periodically—such as daily or upon leader re-election—all instances generate fresh keys for epoch $e+1$, and the leader updates the pointer accordingly. From then on, all encryption targets the $e+1$ keys.

During a bounded grace window Δ (e.g., a few minutes), functions accept messages from either epoch e or $e+1$, allowing in-flight messages to drain from the event mesh. After Δ, instances reject old-epoch messages and securely erase previous keys from enclaves.

5 Security Analysis

We analyze LAMBADA under $\mathcal{A}_{\text{plat}}$ and $\mathcal{A}_{\text{plat\&func}}$. We assume standard cryptographic hardness: AEAD is IND-CPA and INT-CTXT secure; the PRE scheme is (at least) IND-CPA; aggregate signatures are UF-CMA secure; and SGX attestation binds enclave identity and user data (including public keys and epoch e). The registry (`etcd`) is untrusted for integrity: it provides only record availability and supports atomic transactions. LAMBADA derives integrity, freshness,

Algorithm 1 Initializing a function instance for the $\mathcal{A}_{\text{plat\&func}}$ model. (For brevity, we omit leader re-election.)

function Init
- $\text{self.r}_{\text{id}} \leftarrow$ Rand ▷ *replica ID*
- $\text{pk}_\sigma, \text{sk}_\sigma \leftarrow \text{AggSig.KeyGen}(1^{\lambda_\sigma})$
- $\text{pk}_{pre}, \text{sk}_{pre} \leftarrow \text{PRE.KeyGen}(1^{\lambda_{\text{pre}}})$
- $\text{quote} \leftarrow \text{SGX.GetQuote}(H(e \parallel \text{self.r}_{\text{id}} \parallel \text{pk}_\sigma \parallel \text{pk}_{pre}))$
- $\text{self.f}_{\text{id}} \leftarrow \text{quote.MRENCLAVE}$
- $\text{name} \leftarrow /e/\text{self.f}_{\text{id}}/\text{self.r}_{\text{id}}/\texttt{keys}$
- $\text{rec} \leftarrow \{\text{quote}, e, \text{self.r}_{\text{id}}, \text{pk}_\sigma, \text{pk}_{pre}\}$ ▷ *Instance's record*
- Registry.Put(name, rec)

Algorithm 2 Sending in the $\mathcal{A}_{\text{plat\&func}}$ model. (For brevity, we omit appending to the tamper-evident log.)

function $\text{Send}(\text{f}_{\text{id}_{\text{next}}}, \text{msg}, \text{flow}_{\text{id}}, \{\text{rec}_i\}_1^n)$
- $\text{name} \leftarrow /e/\text{f}_{\text{id}_{\text{next}}}/\texttt{leader}$
- $\text{ldr} \leftarrow \text{Registry.Get(name)}$
- ▷ *Verify & validate quote of destination's leader* ◁
- **if** ¬ SGX.VerifyQuote(ldr.quote) **then**
  - **return** ⊥
- **if** $\text{ldr.quote.MRENCLAVE} \neq \text{f}_{\text{id}_{\text{next}}}$ **then**
  - **return** ⊥
- **if** $\text{ldr.quote.userData} \neq H(e \parallel \text{ldr.r}_{\text{id}} \parallel \text{ldr.pk}_\sigma \parallel \text{ldr.pk}_{\text{pre}})$ **then**
  - **return** ⊥
- ▷ *Sign position & update registry* ◁
- $\text{pos} \leftarrow n + 1$
- $\sigma^* \leftarrow \text{AggSig.Sign}(\text{self.sk}_\sigma, \{e, \text{flow}_{\text{id}}, \text{pos}\}, \sigma^*)$
- $\text{name} \leftarrow /\texttt{messages}/\text{flow}_{\text{id}}/\text{self.f}_{\text{id}}$
- Registry.Put(name, true)
- ▷ *KEM-DEM construction* ◁
- $\text{k}_{\text{aead}} \leftarrow \text{AEAD.KeyGen}(1^{\lambda_{\text{aead}}})$
- $\text{kem} \leftarrow \text{PRE.Encrypt}(\text{ldr.pk}_{\text{pre}}, \text{k}_{\text{aead}})$
- $\text{nonce} \leftarrow$ Rand
- $\text{aad} \leftarrow e \parallel \text{f}_{\text{id}_{\text{next}}} \parallel \text{flow}_{\text{id}}$
- $\text{dem} \leftarrow \text{AEAD.Encrypt}(\text{k}_{\text{aead}}, \text{nonce}, \sigma^* \parallel \text{msg}, \text{aad})$
- ▷ *Post message for the next function* ◁
- $\text{posting} \leftarrow \{\text{ldr.r}_{\text{id}}, \text{flow}_{\text{id}}, \text{nonce}, \{\text{rec}_i\}_1^n \cup \{\text{self.rec}\}, \text{kem}, \text{dem}\}$
- $\text{Broker.Post}(\text{f}_{\text{id}_{\text{next}}}, \text{posting})$

Algorithm 3 Receiving in the $\mathcal{A}_{\text{plat\&func}}$ model. (For brevity, we omit handling of the epoch grace window Δ and consulting the tamper-evident log.)

function Receive(posting)
- ▷ *Unpack posted message* ◁
- $\text{r}_{\text{id}_{\text{target}}}, \text{flow}_{\text{id}}, \text{nonce}, \{\text{rec}_i\}_1^n, \text{kem}, \text{dem} \leftarrow \text{posting}$
- **if** $\text{r}_{\text{id}_{\text{target}}} \neq \text{self.r}_{\text{id}}$ **then**
  - ▷ *Fetch re-encryption key* ◁
  - $\text{name} \leftarrow /e/\text{self.f}_{\text{id}}/\text{self.r}_{\text{id}}/\texttt{rk}$
  - $rk_{\text{ldr}\to\text{self}} \leftarrow \text{Registry.Get(name)}$
  - $\text{kem} \leftarrow \text{PRE.ReEncrypt}(\text{rk}_{\text{ldr}\to\text{self}}, \text{kem})$
- ▷ *KEM-DEM construction* ◁
- $\text{k}_{\text{aead}} \leftarrow \text{PRE.Decrypt}(\text{self.sk}_{\text{pre}}, \text{kem})$
- $\text{aad} \leftarrow e \parallel \text{self.f}_{\text{id}} \parallel \text{flow}_{\text{id}}$
- $\text{tmp} \leftarrow \text{AEAD.Decrypt}(\text{k}_{\text{aead}}, \text{nonce}, \text{dem}, \text{aad})$
- **if** tmp = ⊥ **then** ▷ *Decryption failed authentication*
  - **return** ⊥
- $\sigma^*, \text{msg} \leftarrow \text{tmp}$
- ▷ *Verify/Validate enclaves in path. π is the valid call graphs.* ◁
- **if** $\{\text{rec}_i.\text{quote.MRENCLAVE}\}_i^n$ is not a valid execution in π **then**
  - **return** ⊥
- **for all** $\text{rec} \in \{\text{rec}_i\}_1^n$ **do**
  - **if** ¬ SGX.VerifyQuote(rec.quote) **then**
    - **return** ⊥
  - **if** $\text{rec.quote.userData} \neq H(e \| \text{rec.r}_{\text{id}} \| \text{rec.pk}_\sigma \| \text{rec.pk}_{\text{pre}})$ **then**
    - **return** ⊥
- ▷ *Verify path signature* ◁
- $\text{valid} \leftarrow \text{AggSig.Verify}(\text{rec}_1.\text{pk}_\sigma, \ldots, \text{ref}_n.\text{pk}_\sigma, \{e, \text{flow}_{\text{id}}, 1\}, \ldots, \{e, \text{flow}_{\text{id}}, n\}, \sigma^*)$
- **if** ¬valid **then**
  - **return** ⊥
- ▷ *Check for replay* ◁
- $\{\text{entNames}\} \leftarrow \text{Registry.List}(/\texttt{messages}/\text{flow}_{\text{id}}/)$
- **if** $\{\text{entNames}\} \neq \{\text{rec}_i.\text{f}_{\text{id}}\}_1^n$ **then**
  - **return** ⊥
- **return** $\text{msg}, \text{flow}_{\text{id}}, \{\text{rec}_i\}_1^n$

and write-once semantics for replay state from a signed hash-chain log persisted in the registry.

Definition 5.1 (Valid path)**.** Let $G = (V, E)$ be the published signed call graph (nodes are f_{id}s). A flow with ID flow_{id} follows a *valid path* $P = (v_1, \ldots, v_\ell)$ if $(v_i, v_i + 1) \in E$ for all i, and the message's path attestation[1] encodes each hop index i bound to v_i.

Theorem 5.2 (S1 under $\mathcal{A}_{\text{plat}}$: confidentiality and integrity)**.** *In the $\mathcal{A}_{\text{plat}}$ model, inter-function messages in* Lambada *achieve end-to-end confidentiality and ciphertext integrity.*

[1]We mean attestation in the generic sense, not an Intel SGX attestation.

Sketch. The sender KEM-encapsulates an AEAD key to the leader's PRE public key and AEAD-encrypts the payload, using AAD that includes the destination enclave identity and epoch e. As PRE protects the encapsulated key and AEAD is IND-CPA, an adversary controlling the platform learns nothing about the plaintext. As AEAD is INT-CTXT and its AAD binds the destination identity and epoch, any tampering or redirection causes decryption failure at the receiver. Attestation pins keys to enclave code; transitive attestation ensures only attested downstream leaders are targeted (§4.2). □

Theorem 5.3 (S2 under $\mathcal{A}_{\text{plat}}$: path integrity)**.** *Under $\mathcal{A}_{\text{plat}}$, Lambada detects message injection and redirection that would violate the published call graph.*

SKETCH. Each hop signs its position with a conventional signing key; the receiver verifies that the immediate predecessor is valid per the call graph. UF-CMA security prevents forgery; AAD over the destination breaks redirection because the AEAD tag is bound to that identity. A function rejects any injected message that does not have a valid predecessor signature. □

Theorem 5.4 (S2 under $\mathcal{A}_{\text{plat\&func}}$: path integrity with leakage)**.** *In $\mathcal{A}_{\text{plat\&func}}$, even if an attacker transiently compromises some function instances and exfiltrates their keys, the destination detects any path that is not valid with respect to G.*

SKETCH. A compromised hop can produce its signature on arbitrary messages, but cannot forge other hops or re-order indices. The receiver verifies *(a)* the aggregate signature captures a contiguous path with strictly increasing indices, *(b)* all identities appear on an allowed path in G, and *(c)* the registry entries for this flow_{id} reflect the same path prefix (replay defense in §4.4). Thus, an off-graph detour (e.g., $A \to C \to D$ in Figure 2) is detected because either the aggregate signature does not validate against the published keys/indices or the path is not in G. □

Lemma 5.5 (Tamper-evident log soundness)**.** *Let* Log *be the hash-chain log defined in §4.4, rooted at a signed genesis hash and consisting of signed head and entry records where each entry commits to the previous digest and the hash of the appended* (key, value)*. Assume collision resistance of H and UF-CMA security of the signing scheme. Then, except with negligible probability, an adversary controlling the registry cannot* (i) *modify an existing logged record,* (ii) *reorder logged records, or* (iii) *splice together records from different valid histories without causing verification failure for any verifier that checks the chain from genesis to the advertised head.*

SKETCH. Any modification or reordering changes either the signed $(\text{idx}, \text{D}_{\text{idx}})$ head or some entry's committed $(\text{D}_{\text{idx}-1}, H(\text{value}))$, which would require either forging a signature or finding a hash collision to preserve verification. Splicing requires the adversary to produce a consistent sequence of digests and signatures across the splice boundary, which again implies forgery or collision. To detect truncation/rollback, functions retain the last verified head digest and reject any registry view whose head does not extend it. □

Theorem 5.6 (Replay detection)**.** *For any flow_{id}, LAMBADA detects duplicates and forks.*

SKETCH. The ingress assigns each incoming message a unique flow_{id} and each hop records its participation under `/messages/`flow_{id}`/`f_{id}. By Lemma 5.5, an adversary cannot remove, modify, or reorder these records without detection by a verifier that checks the log (and rejects rollback using a last-seen head checkpoint). Under $\mathcal{A}_{\text{plat}}$, a replay to the same destination is rejected because the destination observes a prior record for that flow_{id}. Under $\mathcal{A}_{\text{plat\&func}}$, the receiver lists all `/messages/`flow_{id}`/*` records and checks that the recorded set of hops matches the set authenticated by the path signature, so duplicates or divergent forks create a mismatch. □

Our implementation batches log appends via a tenant-side auditor (§6.3); this changes performance but not safety because functions verify the committed batch contents and treat missing batches as a liveness failure.

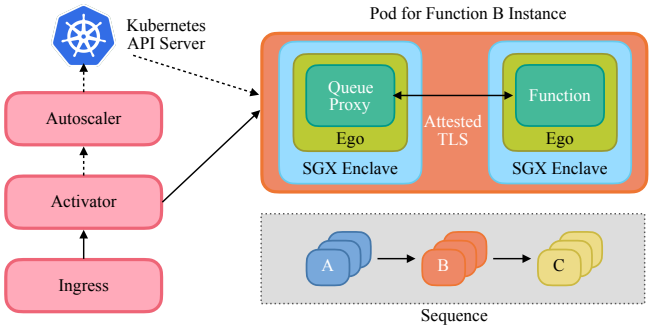


Figure 4: Knative architecture, augmented with LAMBADA's use of SGX enclaves and the Ego runtime. The solid arrows are the request flow, and the dashed arrows the autoscaling flow. Some use cases may also run the Ingress in an enclave.

6 Implementation

6.1 Knative

We implement LAMBADA within Knative, a platform-agnostic framework that brings serverless capabilities to Kubernetes. Per Figure 4, the Knative control plane consists of an HTTP ingress router, an activator, and an autoscaler. Under normal traffic, the ingress router routes requests to the *activator*, which buffers incoming requests and notifies the *autoscaler* if additional capacity is needed; the autoscaler, in turn, requests more pods from Kubernetes. Once new pods are ready—or existing ones become available—the activator forwards the buffered requests. Under high traffic, the ingress router bypasses the activator and routes requests directly to the active function pods. Regardless of the traffic pattern, all requests pass through the *queue-proxy*, a sidecar container in each function pod. The queue-proxy collects metrics, enforces concurrency limits, and can also buffer requests when needed, similar to the activator.

To support function chaining, Knative provides an event mesh—an abstraction of a publish-subscribe system—that enables asynchronous, store-and-forward message delivery, decoupling senders and recipients in time. As the queue-proxy sidecar interposes on the function's I/O, we augment the sidecar with the functionality for proxy re-encryption and path enforcement.

6.2 Ego

We use Edgeless Systems' *Ego* framework to run Knative functions inside SGX enclaves. *Ego* compiles Go applications with a modified compiler (`ego-go`) into enclave-compatible binaries that are signed and launched with the `ego` tool [77, 78], and provides a high-level Go API for attestation and sealing.

We use *Ego* to build, sign, and run both the function and queue-proxy binaries, each in its own enclave. As a result, proxied requests cross enclave boundaries. To secure this channel, we establish an attested TLS connection between the function and the queue-proxy. Since the queue-proxy already acts as a reverse proxy for incoming requests, adding attested TLS is straightforward. However, to support attested TLS on the function side, we introduce a reverse proxy within the function's container that terminates the attested TLS connection on behalf of the function.

Additionally, *Ego* relies on external SGX services for attestation. To facilitate this, we set up the *Intel SGX Data Center Attestation Primitives (DCAP)* locally (which uses ECDSA-based quotes), thus avoiding reliance on the Intel Attestation Service (IAS) during steady-sate validation.

6.3 Kubernetes Components

We implement a public registry using the *etcd* distributed key-value store, allowing functions to publish their keying material. The leader replicas use *etcd*'s *Watch API* to receive public keys, while follower replicas use it to obtain their re-encryption keys. All replicas use the *Watch API* to discover the public key of the next function's leader in the chain. The replicas use the Kubernetes Lease API to handle leader election and discovery.

As described in §4.4, the ingress gateway assigns each incoming message a unique identifier, $\mathsf{flow}_{\mathsf{id}}$, which is logged in the `etcd`-backed hash chain, allowing functions to verify whether a $\mathsf{flow}_{\mathsf{id}}$ has already been processed. A naïve design would have each function f_{id} directly append a processed message (keyed at $\mathsf{flow}_{\mathsf{id}}/\mathsf{f}_{\mathsf{id}}$) to the chain, but concurrent updates from multiple functions cause severe write contention and limit throughput.

As an implementation optimization, we therefore introduce a tenant-side auditor[2] (batching service) that decouples $\mathsf{flow}_{\mathsf{id}}$ tracking from the critical request path. Rather than each function writing individually to the hash chain, the auditor tracks outgoing requests and collects the completed responses, verifies the aggregate signatures to confirm path integrity, and commits $\mathsf{flow}_{\mathsf{id}}$s in batches. Functions then only need to check these committed batches before processing a new request. Functions independently verify the auditor batch records; a faulty auditor can delay confirmation (availability) but cannot forge path evidence.

Both the auditor and functions maintain local $\mathsf{flow}_{\mathsf{id}}$ state and enforce time-bounded windows to handle edge cases. If the auditor fails to receive a response—due to non-Byzantine faults or the platform ($\mathcal{A}_{\mathsf{plat}}$) dropping messages—it waits for a configurable window (e.g., a multiple of the estimated chain traversal time) before committing the missing $\mathsf{flow}_{\mathsf{id}}$, preventing future replays. Conversely, if the platform suppresses batch updates destined for functions, each function continues processing a bounded number of new requests while verifying that incoming hash-chain updates include its recently processed $\mathsf{flow}_{\mathsf{id}}$s; if they do not arrive within the window, the function infers active suppression and halts. Although enclaves lack a trusted clock, they approximate the timeout by counting processed requests.

6.4 Cryptographic Schemes

To guide our choice of PRE and signature schemes, we implement several from the literature using Cloudflare's Circl [38] cryptographic library.

Proxy re-encryption schemes. We implement four PRE schemes: BBS98 [17], AFGH05 [11], CH07 [28], and LV08 [59]. BBS98 and CH07 are interactive schemes that require the delegatee to share their private key with the delegator. We can support this in our *etcd*-based registry by having the delegatee encrypt their private key using the delegator's public key, at the expense of additional cryptographic operations during registration. In bidirectional schemes (BBS98, CH07), ciphertexts have the same structure for both encryption and re-encryption. In contrast, unidirectional schemes (AFGH05, LV08) use different ciphertext formats: first-level ciphertexts for encryption, and second-level ciphertexts for re-encryption.

[2]An enclaved Lambada ingress gateway could also serve as an auditor.

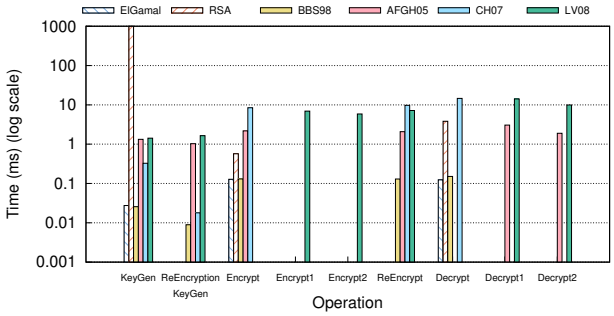


Figure 5: The time for each proxy re-encryption operation across schemes. For comparison, the figure plots the ElGamal (using the P-256 curve) and RSA (using a 3072-bit key) encryption scheme. All schemes exhibit ~128-bits of security.

Figure 5 presents Go benchmark results for each scheme's operations, alongside RSA and ElGamal for comparison. Among the unidirectional schemes, AFGH05 performs better overall. Its re-encryption and decryption times are comparable to—or faster than—RSA, though its encryption cost is roughly an order of magnitude higher. Thus, we use AFGH05 in our Lambada proof-of-concept.

Signature schemes. We implement two multisignature schemes (B03 [18] and BGOY07 [19]) and one aggregate signature scheme, (BGLS03 [22]), comparing their performance to Ed25519 and RSA. The B03 construction is non–order-preserving and is therefore unsuitable for Lambada's ordered path verification, but it serves as a useful baseline. Figure 5 reports Go benchmark results for signing and verification as the number of signers increases. For all schemes except BGOY07, signing time remains $O(1)$, while verification time is sublinear for B03 and BGOY07 and $O(n)$ for BGLS03 in the number of signers n. Empirically, BGLS03 exhibits faster signing performance than BGOY07, with comparable verification latency for small signer sets ($n \leq 10$), which aligns with Lambada's function-chain scale. Accordingly, our proof-of-concept implementation adopts BGLS03 as its aggregate signature scheme.

7 Evaluation

We build on Knative Serving `v1.17.0`. To run enclave-compatible binaries, we use the *Intel SGX device plugin for Kubernetes* [45], which grants workloads access to SGX enclaves and enables the queue-proxy and function containers to generate attestation reports.

7.1 Stress Tests

To stress test the performance of Lambada, we use an *appender* function as our sample function. The function creates a response appending a constant string to the request message. We run our experiments on a single-node local Kubernetes cluster with 16

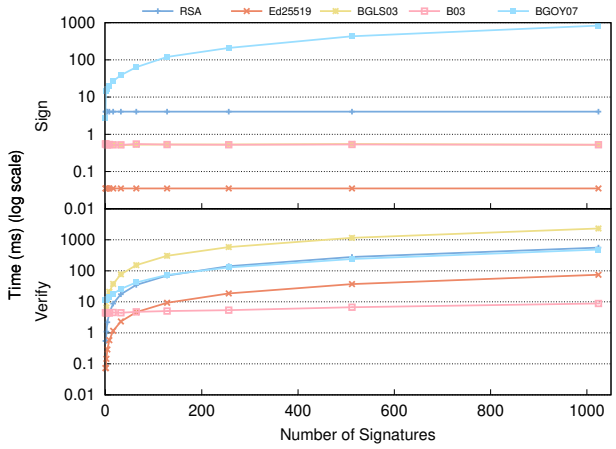


Figure 6: The time for sign and verify operations for aggregate and multisignature schemes. The sign latencies for BGLS03 are nearly identical to B03. The figure also plots the Ed25519 and RSA (using a 3072-bit key) signature schemes.

Table 4: Function variants used in evaluation

Function Configuration	Features
LAMBADA **Follower** EFunction	`(PRE + 1) + SGX`
LAMBADA **Leader** EFunction	`PRE + SGX`
RSA EFunction	`RSA + SGX`
EFunction	`SGX`
Knative Function	–

CPUs and 32 GB of memory, created using Minikube. The node runs as a Docker container on an Intel Xeon Gold 5512U server with SGXv2. To gather metrics for benchmarking, we deploy *Prometheus*, *Grafana*, and *kube-state-metrics* [4] into the cluster.

Table 4 summarizes our function configurations. `SGX` indicates the *function* and *queue-proxy* binaries (built with *Ego*) are run inside SGX enclaves. `RSA` denotes functions that use a shared RSA-3072 key, provisioned by a trusted registry, to hybrid-encrypt responses. `PRE` denotes functions using proxy re-encryption (AFGH05), and `(PRE+1)` indicates the additional re-encryption performed by follower replicas. Finally, `"-"` indicates unmodified functions running without SGX enclaves.

Function deployment. We deploy a single *appender* instance as a Knative service and measure its time to readiness. Since a Knative service runs in a Kubernetes pod, we define deployment time as the interval from the `PodScheduled` state to the `Ready` state. A function is ready once it can serve requests and its *queue-proxy* sidecar container is initialized and ready to proxy them.

We repeat each measurement 50 times and compute the 10% trimmed mean. We measure the deployment time for four different function configurations: *(1)* **Knative** serves as the baseline, *(2)* **EFunction** represents the baseline **Knative** function deployed within an enclave, *(3)* **Leader** assumes the role of a delegator and

Table 5: Average time (sec) to deploy function configurations

Function Configuration	Duration
Knative Function	2.05
EFunction	6.225
LAMBADA Leader EFunction	7.45
LAMBADA Follower EFunction	8.325

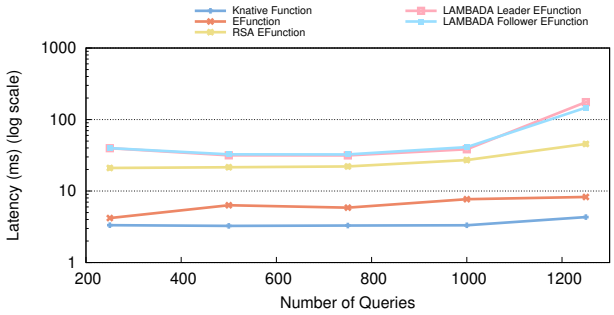


Figure 7: p95 latency for function invocation

creates the re-encryption keys for the other replicas; and *(4)* **Follower** acts as a delegatee, waiting for the leader function before creating its own keys and receiving its re-encryption key.

Per Table 5, enclaved function variants incur higher deployment times than the baseline: 3× (*EFunction*), 3.6× (*Leader*), and 4.1× (*Follower*). This is because both the function and the *queue-proxy* sidecar binaries run in separate enclaves, and it takes 4–6 s for each application to be loaded into the enclave. This latency is primarily due to the static heap initialization cost in EGo, which currently lacks support for Enclave Dynamic Memory Management (EDMM). Adopting a runtime with EDMM support would significantly mitigate this initialization overhead [80]. The additional delay for the *Follower* function is due to its dependence on the *Leader* to create its re-encryption key. To improve results, we cache the deplomentimages locally and set the probe frequency to `1s` to speed up readiness checks. Finally, we note deployment time can be further reduced by optimizing enclave startup using alternative techniques [49, 53, 85].

Function invocation. We deploy a single *appender* instance and measure its response time, which includes network transit as well as request decryption, processing, and response encryption. Using the *vegeta* [84] load-testing tool, we generate requests at constant rates ranging from 250 to 1250 (in increments of 250) and run each test for five minutes.

We compare the `p95` latency of the *appender* function in five different configurations (Figure 7). The **EFunction** variants using either RSA or proxy encryption incur higher latencies due to request decryption and response encryption being on the request path. **Leader** incurs 1.55× the latency observed for the **RSA**, which is consistent with the cryptographic microbenchmarks shown in Figure 5 in §6. In addition, follower replicas (**Follower**) incur a 3.6% overhead due to the extra re-encryption operation, compared to leaders.

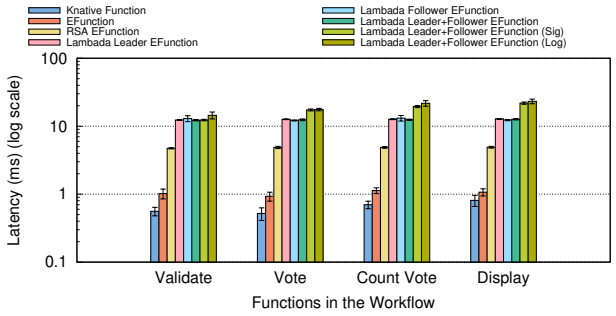


Figure 8: Mean latency observed for each function in the *emojivoto* function workflow

7.2 Application Macrobenchmark

We use a managed, four-node Azure Kubernetes Service (AKS) cluster consisting of DC4s_v3 nodes to evaluate Lambada's performance in a real-world application. Specifically, we evaluate the *emojivoto* [27] microservice application, which allows users to *(1)* vote for their favorite emoji and *(2)* keep track of the total votes. We extract the core functionalities of *emojivoto* into four separate serverless functions, which we then compose into a *function workflow* (sequence). The four functions are:

- **Validate:** Validates incoming votes.
- **Vote:** Records each vote.
- **Count Vote:** Takes a snapshot of all votes.
- **Display:** Displays the top 10 most-voted emojis to the user.

We instrument the cluster with *Zipkin* [12] for distributed tracing. A user initiates the *emojivoto* function workflow by sending an event specifying the emoji they wish to vote for. To simulate user activity, we send requests to the workflow at a rate of 100 rps for 5 minutes, sampling 1% of the requests (300 traces).

Figure 8 shows the average time (10% trimmed mean) each request spends on a particular function. We measure workflow duration by executing the individual functions under the configurations in Table 4. We focus on the Display function since, as the final stage, it incurs the highest verification cost.

We first compare **Knative**, **EFunction**, and **RSA**, which do not use proxy re-encryption. RSA serves as the baseline and exhibits 6× higher latency than Knative due to hybrid encryption and enclave overhead. Next, we evaluate configurations with proxy re-encryption—**Leader**, **Follower**, and **Leader+Follower**—which add about 2.5× latency over RSA. Finally, enabling signature verification (**Sig**) for path integrity and a tamper-evident log (**Log**) for replay detection atop **Leader+Follower** raises latency to roughly 4× RSA, as each function verifies all prior signatures in the flow and checks whether the request's flow_{id} appears in the log.

8 Related Work

Achieving confidential FaaS. S-FaaS [7] was the first system to integrate Intel SGX enclaves into the FaaS model, while also addressing the challenge of trustworthy resource metering. Several similar systems [25, 41, 80] followed closely, adopting libc shims and library OS approaches like SCONE [10] and SGX-LKL [51, 68] to ensure the confidentiality and integrity of function code and data. These systems all rely on a key distribution enclave to securely generate and distribute keys to dynamically created function enclaves. In contrast, the core contribution of Lambada is the elimination of this centralized trust component, allowing each enclave to locally generate their keys, and the untrusted cloud provider to safely re-encrypt data between instances. Among these systems, Clemmys [80] is unique in cryptographically enforcing function execution order: the key distribution enclave stores the expected sequence, and each function signs its position in the chain. Lambada adopts a similar approach, but further demonstrates the use of aggregate signatures to optimize signature size and overhead.

Flow control in serverless. Kalium [47] enforces control-flow integrity within and across function instances to prevent data-integrity violations and stealthy operations. AutoArmor [57] automates inter-service access control policy generation using static-analysis and graph techniques. Trapeze [8] and Growlithe [40] use language- and compiler-level defenses to prevent buggy or malicious third-party functions from leaking data. Valve [32] enforces fine-grained *information-flow control (IFC)* by interposing on function I/O and propagating taint labels across invocations. A natural extension of Lambada would integrate IFC to track data movement between enclaves, complementing its guarantees on permitted execution paths.

Sandboxing untrusted functions. While Lambada's threat model either assumes function invulnerability or tolerates limited data leakage, prior work has focused on building secure runtimes for safely executing untrusted or buggy functions. Several studies [48, 60, 69] analyze Docker escape techniques and propose lightweight countermeasures based on enhanced namespace isolation and credential management. Others pursue more substantial design changes, such as stacking namespace policies [76], achieving strong isolation through library operating systems [75], or introducing container-aware network stacks to sandbox communication [65]. Industry efforts include secure container runtimes such as Alibaba's RunD [58] and Google's gVisor [79]. Another line of research targets runtime anomaly detection to identify misbehavior during execution [6, 36]. These approaches are complementary to Lambada and provide additional defense-in-depth against compromised functions.

9 Conclusion

As Mark Russinovich, CTO of Microsoft Azure, famously stated, in the future, "confidential computing will just be computing" [71]. This paper presents Lambada, a step to bringing confidentiality to the widely used FaaS model while maintaining compatibility with existing applications. To support future work, we have made the code for Lambada available at https://github.com/etclab/serving.

Acknowledgments

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

References

[1] 2014. US court forces Microsoft to hand over personal data from Irish server. https://www.theguardian.com/technology/2014/apr/29/us-court-microsoft-personal-data-emails-irish-server.

[2] 2020. The police want your phone data. Here's what they can get — and what they can't. https://www.vox.com/recode/2020/2/24/21133600/police-fbi-phone-search-protests-password-rights.

[3] 2022. Amazon gave Ring videos to police without owners' permission. https://www.politico.com/news/2022/07/13/amazon-gave-ring-videos-to-police-without-owners-permission-00045513.

[4] 2025. Prometheus Operator. https://prometheus-operator.dev/.

[5] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. 2021. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium*.

[6] Rami Alboqmi and Rose F. Gamble. 2025. Enhancing Microservice Security Through Vulnerability-Driven Trust in the Service Mesh Architecture. *Sensors* 25, 3 (Feb. 2025).

[7] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *ACM Workshop on Cloud Computing Security (CCSW)*.

[8] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[9] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. 2005. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In *Network and Distributed System Security Symposium (NDSS)*.

[12] Zipkin Authors. 2025. Zipkin. https://zipkin.io/.

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

[14] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

[15] Mihir Bellare and Gregory Neven. 2006. Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma. In *ACM Conference on Computer and Communications Security (CCS)*.

[16] Mihir Bellare, Phillip Rogaway, and David Wagner. 2003. EAX: A Conventional Authenticated-Encryption Mode. Cryptology ePrint Archive, Paper 2003/069. https://eprint.iacr.org/2003/069

[17] Matt Blaze, Gerrit Bleumer, and Martin Strauss. 1998. Divertible Protocols and Atomic Proxy Cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[18] Alexandra Boldyreva. 2003. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *International Conference on Practice and Theory in Public Key Cryptography (PKC)*.

[19] Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. 2007. Ordered Multisignatures and Identity-Based Sequential Aggregate Signatures, with Applications to Secure Routing. In *ACM Conference on Computer and Communications Security (CCS)*.

[20] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact Multi-Signatures for Smaller Blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

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

[22] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[23] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

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

[25] Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In *ACM International Systems and Storage Conference (SYSTOR)*.

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

[27] BuoyantIO. [n. d.]. emojivoto | Example application to help demonstrate the Linkerd service mesh. https://github.com/BuoyantIO/emojivoto.

[28] Ran Canetti and Susan Hohenberger. 2007. Chosen-ciphertext secure proxy re-encryption. In *ACM Conference on Computer and Communications Security (CCS)*.

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

[30] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. Technical Report 2016/086. Cryptology ePrint Archive.

[31] cve-2022-39388 2022. Istio May Allow Identity Impersonation If User Has Localhost Access. CVE-2022-39388. https://www.cve.org/CVERecord?id=CVE-2022-39388.

[32] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *The Web Conference (WWW)*.

[33] Alex Davidson, Amit Deo, Ela Lee, and Keith Martin. 2019. Strong Post-Compromise Secure Proxy Re-Encryption. In *Australasian Conference on Information Security and Privacy (ACISP)*.

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*.

[35] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. 2020. Pixel: Multi-signatures for Consensus. In *USENIX Security Symposium*.

[36] Asbat El Khairi, Marco Caselli, Christian Knierim, Andreas Peter, and Andrea Continella. 2022. Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection. In *ACM Workshop on Cloud Computing Security (CCSW)*.

[37] T. Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (July 1985).

[38] Armando Faz-Hernandez and Kris Kwiatkowski. 2019. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare. Available at https://github.com/cloudflare/circl. v1.6.0 Accessed Jan, 2025.

[39] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *ACM Symposium on Cloud Computing (SoCC)*.

[40] Praveen Gupta, Arshia Moghimi, Devam Sisodraker, Mohammad Shahrad, and Aastha Mehta. 2025. Growlithe: A Developer-Centric Compliance Tool for Serverless Applications. In *IEEE Symposium on Security and Privacy*.

[41] Changhee Han, Taehun Kim, Woomin Lee, and Youngjoo Shin. 2024. S-ZAC: Hardening Access Control of Service Mesh Using Intel SGX for Zero Trust in Cloud. *Electronics* 13, 16 (Aug. 2024).

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

[43] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

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

[45] Intel. 2020. Intel SGX Device. https://intel.github.io/intel-device-plugins-for-kubernetes/cmd/sgx_plugin/README.html.

[46] Anca-Andreea Ivan and Yevgeniy Dodis. 2003. Proxy Cryptography Revisited. In *Network and Distributed System Security Symposium (NDSS)*.

[47] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. 2023. Guarding Serverless Applications with Kalium. In *USENIX Security Symposium*.

[48] Zhiqiang Jian and Long Chen. 2017. A Defense Method against Docker Escape Attack. In *International Conference on Cryptography, Security and Privacy (CSP)*.

[49] Seong-Joong Kim, Myoungsung You, Byung Joon Kim, and Seungwon Shin. 2023. Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves. In *ACM Symposium on Cloud Computing (SoCC)*.

[50] Loren Kohnfelder and Praerit Garg. 1999. *The Threats to Our Products*. Technical Report. Microsoft.

[51] Large-Scale Data & Systems (LSDS) Group. [n. d.]. SGX-LXL. https://github.com/lsds/sgx-lkl. Accessed: 2025-07-13.

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

[53] Mingyu Li, Yubin Xia, and Haibo Chen. 2021. Confidential Serverless Made Efficient with Plug-In Enclaves. In *International Symposium on Computer Architecture (ISCA)*.

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

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

[56] Tao Li, Wencheng Lu, Spike Curtis, Oliver Liu, Evan Gilman, and Justin Burke. 2018. *Istio Security Threat Model*. Technical Report. Istio.

[57] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. 2021. Automatic Policy Generation for Inter-Service Access Control of Microservices. In *USENIX Security Symposium*.

[58] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup

in Serverless Computing. In *USENIX Annual Technical Conference (ATC)*.

[59] Benoît Libert and Damien Vergnaud. 2008. Unidirectional Chosen-Ciphertext Secure Proxy Re-Encryption. In *International Conference on Practice and Theory in Public Key Cryptography (PKC)*.

[60] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Annual Computer Security Applications Conference (ACSAC)*.

[61] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. 2006. Sequential aggregate signatures and multisignatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[62] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. 2004. Sequential Aggregate Signatures from Trapdoor Permutations. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[63] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. 2019. Simple Schnorr multi-signatures with applications to Bitcoin. *Designs, Codes, and Cryptography* 87 (Sept. 2019).

[64] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[65] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *USENIX Annual Technical Conference (ATC)*.

[66] Jonas Nick, Tim Ruffing, and Yannick Seurin. 2021. MuSig2: Simple Two-Round Schnorr Multi-Signatures. In *International Cryptology Conference (CRYPTO)*.

[67] Yuriy Polyakov, Kurt Rohloff, Gyana Sahu, and Vinod Vaikuntanathan. 2017. Fast Proxy Re-Encryption for Publish/Subscribe Systems. *ACM Transactions on Privacy and Security* 20, 4 (Sept. 2017).

[68] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. 2020. SGX-LKL: Securing the Host OS Interface for Trusted Execution. arXiv:1908.11143 https://arxiv.org/abs/1908.11143

[69] Michael Reeves, Dave (Jing) Tian, Antonio Bianchi, and Z. Berkay Celik. 2021. Towards Improving Container Security by Preventing Runtime Escapes. In *IEEE Secure Development Conference (SECDEV)*.

[70] Phillip Rogaway. 2002. Authenticated-Encryption with Associated-Data. In *ACM Conference on Computer and Communications Security (CCS)*.

[71] Mark Russinovich. 2023. Confidential Computing: Elevating Cloud Security and Privacy. *ACM Queue* 21, 4 (Sept. 2023).

[72] Amit Sahai, Hakan Seyalioglu, and Brent Waters. 2012. Dynamic Credentials and Ciphertext Delegation for Attribute-Based Encryption. In *International Cryptology Conference (CRYPTO)*.

[73] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. 2009. Towards Trusted Cloud Computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[74] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX Annual Technical Conference (ATC)*.

[75] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[76] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security Symposium*.

[77] Edgeless Systems. 2025. Programming Model | EGo. https://docs.edgeless.systems/ego/knowledge/model.

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

[79] The gVisor Authors. [n. d.]. gVisor - The Container Security Platform. https://gvisor.dev. Accessed: 2025-07-25.

[80] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. 2019. Clemmys: Towards Secure Remote Execution in FaaS. In *ACM International Systems and Storage Conference (SYSTOR)*.

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

[82] 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*.

[83] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*.

[84] vegeta's authors. [n. d.]. tsenart/vegeta: HTTP load testing tool and library. It's over 9000! https://github.com/tsenart/vegeta.

[85] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable Enclaves for Confidential Serverless Computing. In *USENIX Security Symposium*.