Privacy-First REST API Gateway Using Partially Homomorphic Encryption: An Empirical Performance Evaluation at Scale

Author Name* *Department of Computer Science,
University Name, City, Country
Email: author@university.edu

Abstract

The proliferation of cloud-based APIs has raised critical concerns about data privacy, particularly when sensitive numerical data traverses untrusted network intermediaries such as proxies, load balancers, and API gateways. While Fully Homomorphic Encryption (FHE) offers theoretical completeness, its computational overhead renders it impractical for latency-sensitive REST API workloads. This paper presents PHE-GATEWAY, a privacy-first REST API gateway architecture that leverages Paillier Partially Homomorphic Encryption (PHE) to perform meaningful computations—specifically, homomorphic summation and counting—on encrypted payloads without ever accessing plaintext data. We implement a complete experimental environment in Go and conduct a large-scale empirical evaluation comprising up to **[N total]** request records across five systematic experiment suites: concurrency scaling (1–1,000 users), batch size impact (1–1,000 items), proxy-induced latency (0–100 ms), throughput stress testing, and fault injection (0–5% drop rates). Our results demonstrate that PHE-GATEWAY achieves **[X]** requests per second under burst load with a median end-to-end latency of **[Y ms]** at batch size 10, while ciphertext expansion remains bounded at **[Z]**× relative to plaintext. We further characterize the cost decomposition of PHE operations in production-like conditions, quantify the impact of real-world proxy diversity (1,025 HTTP/SOCKS5/SOCKS4 proxies across **[N]** countries), and identify operational boundaries where PHE-based API privacy becomes practical. To the best of our knowledge, this is the first large-scale empirical study evaluating PHE-based REST API gateways under realistic, proxy-mediated network conditions.

Index Terms

Partially Homomorphic Encryption, Paillier Cryptosystem, REST API Gateway, Privacy-Preserving Computation, Proxy Networks, Performance Evaluation, Cloud Security

I. INTRODUCTION

The modern internet economy is built upon REST APIs that transmit, aggregate, and process sensitive numerical data—financial transactions, health metrics, sensor telemetry, and user analytics—across networks of untrusted intermediaries. Load balancers, API gateways, reverse proxies, and content delivery networks routinely inspect, route, and sometimes cache this data in plaintext, creating a broad attack surface for data exfiltration, insider threats, and regulatory non-compliance [16], [17].

Transport Layer Security (TLS) protects data in transit between hops but provides no protection at the application layer: once a request reaches an API gateway, its payload is fully accessible in plaintext. This architectural limitation fundamentally conflicts with privacy-by-design principles mandated by GDPR [18], HIPAA, and emerging data sovereignty regulations.

Homomorphic encryption offers a compelling theoretical solution: perform computations on encrypted data without decryption. Fully Homomorphic Encryption (FHE) supports arbitrary computations but imposes orders-of-magnitude overhead in both computation and ciphertext size [2], [6], making it impractical for latency-sensitive API workloads. Partially Homomorphic Encryption (PHE), in contrast, supports a restricted but practically useful set of operations—notably additive homomorphism in the Paillier cryptosystem [1]—with significantly lower overhead.

Despite the theoretical promise, **no prior work has empirically evaluated the viability of PHE-based REST API gateways under realistic, large-scale, proxy-mediated network conditions**. Existing evaluations of homomorphic encryption focus on raw cryptographic benchmarks [7] or specialized applications such as machine learning [12], neglecting the systems-level concerns of API gateway deployments: concurrent user scaling, proxy-induced latency and jitter, fault tolerance, and ciphertext expansion impact on bandwidth.

A. Contributions

This paper makes the following contributions:

(1) **Architecture.** We design PHE-GATEWAY, a privacy-first REST API gateway that performs homomorphic summation and counting on Paillier-encrypted payloads, requiring zero plaintext access at the gateway layer.
(2) **Experimental Environment.** We implement a complete, reproducible experimental framework in Go comprising a concurrent user simulation engine, an untrusted proxy simulator with configurable fault injection, the PHE-enabled gateway, and a worker pool—all executable with a single command.
(3) **Large-Scale Evaluation.** We conduct systematic experiments generating up to **[1B]** request records across five dimensions: concurrency scaling, batch size impact, proxy latency, throughput stress, and failure injection, using a pool of 1,025 real-world HTTP/SOCKS5/SOCKS4 proxies.
(4) **Cost Decomposition.** We decompose end-to-end latency into its constituent components—client-side encryption, network transit, gateway processing, worker queue wait, and PHE computation—enabling precise identification of performance bottlenecks.
(5) **Practical Boundaries.** We identify the operational regimes where PHE-based API privacy is viable and characterize the conditions under which it degrades beyond acceptable thresholds.

B. Paper Organization

Section II reviews relevant background on Paillier PHE and API gateway architectures. Section III surveys related work. Section IV presents the PHE-GATEWAY architecture. Section V describes our experimental methodology. Section VI presents and analyzes experimental results. Section VII discusses implications, limitations, and practical considerations. Section VIII concludes with future directions.

II. BACKGROUND

A. Paillier Partially Homomorphic Encryption

The Paillier cryptosystem [1] is a probabilistic, additively homomorphic public-key encryption scheme based on the decisional composite residuosity assumption. Given a public key (n, g) where $n = pq$ is the product of two large primes and $g = n + 1$, encryption of a message $m \in \mathbb{Z}_n$ is:

$$E(m) = g^m \cdot r^n \bmod n^2 \tag{1}$$

where r is a random value in $\mathbb{Z}_n^*$. The scheme satisfies the following homomorphic properties:

Additive Homomorphism:

$$E(m_1) \cdot E(m_2) \bmod n^2 = E(m_1 + m_2) \tag{2}$$

Scalar Multiplication:

$$E(m)^k \bmod n^2 = E(m \cdot k) \tag{3}$$

These properties enable summation and counting of encrypted values without decryption, which are the fundamental operations required by our API gateway.

B. Key Properties

- **Semantic Security:** Paillier encryption is probabilistic (due to random r), ensuring that encryptions of the same plaintext produce different ciphertexts.
- **Ciphertext Expansion:** For an n-bit key, each ciphertext is approximately $2n$ bits, independent of plaintext size. For a 1024-bit key, each ciphertext is $\sim$256 bytes.
- **Computational Cost:** Encryption requires one modular exponentiation of g^m and one of r^n modulo n^2, both $O(\log^2 n)$ multiplications.

C. REST API Gateway Architecture

A REST API gateway is a reverse proxy that sits between clients and backend services, providing routing, load balancing, authentication, and request transformation. In conventional architectures, the gateway has full access to request payloads in plaintext. Our work challenges this assumption by demonstrating that meaningful aggregation operations can be performed on encrypted payloads.

D. Threat Model: Honest-but-Curious Proxy

We adopt the standard *honest-but-curious* adversary model for network intermediaries:

- The proxy faithfully forwards messages between clients and the gateway.
- The proxy may observe metadata: timing, payload sizes, HTTP headers, and request frequency.
- The proxy **cannot** decrypt or modify ciphertext payloads.
- The gateway has access to the public key only; the private key remains exclusively with clients.

This model reflects real-world deployment conditions where proxies, CDNs, and cloud load balancers process traffic from mutually distrustful parties.

III. Related Work

A. Homomorphic Encryption Systems

Gentry's seminal work on fully homomorphic encryption (FHE) [2] demonstrated the theoretical possibility of arbitrary computation on encrypted data. Subsequent systems—BGV [3], BFV [4], CKKS [5], and TFHE [6]—have reduced the practical overhead of FHE, but computational costs remain 3–6 orders of magnitude higher than plaintext operations [7].

Libraries such as Microsoft SEAL [9], HElib [8], OpenFHE [10], and Lattigo [11] provide production-grade implementations. However, these target batch computation workloads (e.g., machine learning inference) rather than per-request API processing.

B. PHE in Applied Systems

Paillier encryption has been applied to privacy-preserving aggregation in smart grids [13], electronic voting [14], and federated learning [15]. These applications share the common pattern of aggregating encrypted values from multiple sources—precisely the pattern our API gateway exploits.

C. Privacy-Preserving API Design

Prior work on privacy-preserving APIs has focused on differential privacy [19], secure multi-party computation [20], and trusted execution environments [21]. To our knowledge, no prior work has evaluated PHE as a mechanism for privacy-preserving REST API gateways, nor conducted large-scale performance studies under realistic proxy-mediated network conditions.

D. Proxy and Network Simulation

Network simulation tools such as tc/netem, Toxiproxy, and Chaos Monkey provide fault injection capabilities. Our approach embeds a configurable proxy simulator directly into the experimental framework, enabling reproducible, parameterized experiments without external infrastructure dependencies. Additionally, we incorporate 1,025 real-world public proxies to introduce authentic network diversity.

IV. System Architecture

PHE-Gateway comprises four components operating in a pipeline architecture:

A. Concurrent User Simulation Engine

The simulation engine generates realistic encrypted API traffic using Go goroutines. Each simulated user independently:

1) Generates a batch of random numeric payloads $\{m_1, m_2, \ldots, m_k\}$ where $m_i \in [1, 1000]$.
2) Encrypts each value using Paillier encryption (Equation (1)) with the shared public key.
3) Serializes the encrypted batch as a JSON array of hexadecimal-encoded ciphertexts.
4) Transmits the request through the proxy chain to the gateway.
5) Records per-request metrics: encryption time, end-to-end latency, ciphertext size, success/failure.

Each user goroutine maintains a deterministic pseudo-random number generator seeded with seed + userID, ensuring reproducibility.

B. Untrusted Proxy Simulator

The proxy simulator operates as an L7 reverse proxy between clients and the gateway. It provides configurable network condition simulation:

- **Latency injection**: Fixed delay d_f plus uniform jitter $d_j \sim U(0, d_{\max})$.
- **Fault injection**: Random request dropping with probability p_{drop}.
- **Metadata observation**: Logs request timing, payload sizes, and headers without inspecting ciphertext.

Critically, the proxy **never decrypts or inspects** encrypted payloads, maintaining the honest-but-curious model.

1) Real-World Proxy Pool: To introduce authentic network diversity, each simulated user routes traffic through a real external proxy selected via round-robin from a pool of 1,025 verified proxies spanning HTTP (410), SOCKS5 (546), and SOCKS4 (69) protocols. The pool is sourced from the IPLocate free proxy list [22] and auto-refreshed every 30 minutes.

C. Privacy-First REST API Gateway

The gateway exposes a single REST endpoint:

```
POST /compute
Content-Type: application/json
{
  "request_id": "...",
  "operation": "sum" | "count",
  "payloads": ["hex_ciphertext_1", ...]
}
```

It supports two homomorphic operations:

- **Sum**: $E\left(\sum_{i=1}^{k} m_i\right) = \prod_{i=1}^{k} E(m_i) \bmod n^2$ using Equation (2).
- **Count**: $E(k) = E(1)^k \bmod n^2$ using Equation (3).

The gateway is **stateless** and has access only to the public key. It **cannot** decrypt any payload.

D. PHE Worker Pool

Homomorphic operations are dispatched to a pool of W worker goroutines via a buffered Go channel. Each worker:

1) Dequeues a job from the channel (measuring queue wait time).
2) Executes the requested homomorphic operation.
3) Returns the encrypted result with timing metadata.

This design enables controlled measurement of both computation time and queuing delays under varying concurrency.

E. Streaming Metrics Recorder

All per-request metrics are streamed to disk via a dedicated goroutine, writing simultaneously to CSV (for statistical analysis) and JSON Lines (for structured inspection) formats. Files are automatically chunked at configurable boundaries (default: 1,000,000 records per file), enabling the system to record up to 10^9 requests without memory exhaustion.

V. Experimental Methodology

A. Implementation

The entire system is implemented in Go (version 1.25.6) using only the standard library, with no third-party dependencies. The Paillier cryptosystem is implemented from first principles using Go's `math/big` arbitrary-precision arithmetic and `crypto/rand` cryptographic random number generator. Key sizes follow standard practice at 1024 bits.

B. Hardware and Environment

[Describe your hardware: CPU model, cores, RAM, disk type, OS version, Go version.]

C. Experiment Design

We conduct five systematic experiment suites, each isolating one variable while controlling others:

1) Experiment 1: Concurrency Scaling: Measures system behavior as concurrent users increase from 1 to 1,000. Fixed parameters: batch size = 10, proxy latency = 0 ms, drop rate = 0%.

TABLE I: Concurrency Scaling Parameters

Parameter	Values
Concurrent users	1, 10, 100, 500, 1,000
Batch size	10
Requests per user	100
Proxy latency	0 ms
Drop rate	0%

2) Experiment 2: Batch Size Impact: Measures the effect of increasing batch size (number of ciphertexts per request) from 1 to 1,000. Fixed: 10 concurrent users, 0 ms latency.

3) Experiment 3: Proxy Latency Impact: Measures the effect of network-induced latency injected by the proxy layer. Fixed: 10 users, batch size 10.

4) Experiment 4: Throughput Stress Test: Measures maximum sustained throughput under burst load. 1,000 concurrent users send requests as fast as possible for 30 seconds.

5) Experiment 5: Failure Injection: Measures system resilience under random request dropping at rates of 0%, 1%, and 5%. Fixed: 50 users, batch size 10.

TABLE II: Batch Size Scaling Parameters

Parameter	Values
Batch sizes	1, 10, 100, 1,000
Concurrent users	10
Requests per user	100
Proxy latency	0 ms

TABLE III: Proxy Latency Parameters

Parameter	Values
Proxy latency	0, 10, 50, 100 ms
Concurrent users	10
Batch size	10
Requests per user	100

D. Metrics Collected

For each request, we record 14 metrics:

E. Reproducibility

All experiments use deterministic random seeds. Configuration is persisted in `meta.json` per experiment. The entire system executes with a single command: `go run cmd/usersim/main.go`. Source code and data will be made publicly available.

VI. Results

A. Experiment 1: Concurrency Scaling

[Insert Figure: Line chart showing median end-to-end latency vs. number of concurrent users. Include error bars for P25/P75.]

[Insert Figure: Stacked bar chart decomposing end-to-end latency into encryption, network, queue wait, and PHE compute for each concurrency level.]

Key Observations:

- **[Describe how latency scales with concurrency. Linear? Sublinear? Where does it degrade?]**
- **[Describe throughput ceiling and where worker pool saturation occurs.]**
- **[Describe queue wait time growth pattern.]**

B. Experiment 2: Batch Size Impact

[Insert Figure: Dual-axis chart showing encryption time (left axis) and ciphertext size (right axis) vs. batch size on log scale.]

Key Observations:

- **[Describe linear vs. sublinear scaling of encryption time with batch size.]**
- **[Quantify ciphertext expansion ratio (ciphertext bytes / plaintext bytes).]**
- **[Identify practical batch size limits for latency-sensitive applications.]**

C. Experiment 3: Proxy Latency Impact

[Insert Figure: Comparison of PHE overhead as a percentage of total latency across different proxy latencies. Show that PHE cost becomes proportionally smaller as network latency increases.]

Key Observations:

- **[Show that at higher network latencies, PHE overhead becomes a smaller fraction of total cost.]**
- **[Quantify the "amortization effect" of network latency on encryption cost.]**

TABLE IV: Per-Request Metrics

Metric	Description
`encryption_time_ms`	Client-side Paillier encryption time for the full batch
`end_to_end_latency_ms`	Total client-observed time (encrypt + network + compute)
`gateway_processing_ms`	Gateway processing time (parse + dispatch + response)
`phe_compute_ms`	PHE worker computation time
`queue_wait_ms`	Time job spent waiting in the worker queue
`proxy_delay_ms`	Configured proxy latency
`ciphertext_size_bytes`	Encrypted JSON payload size
`plaintext_size_bytes`	Equivalent plaintext payload size
`success`	Request success/failure indicator

TABLE V: Concurrency Scaling Results

Users	Median E2E (ms)	P99 E2E (ms)	Throughput (req/s)	Encrypt (ms)	Success (%)
1	[–]	[–]	[–]	[–]	[–]
10	[–]	[–]	[–]	[–]	[–]
100	[–]	[–]	[–]	[–]	[–]
500	[–]	[–]	[–]	[–]	[–]
1000	[–]	[–]	[–]	[–]	[–]

D. Experiment 4: Throughput Stress Test

[Insert Figure: Time-series plot of instantaneous throughput (req/s) over the 30-second burst window.]

E. Experiment 5: Failure Injection

Key Observations:

- **[Compare configured drop rate vs. observed failure rate.]**
- **[Describe whether failures affect latency of successful requests.]**

F. Cross-Experiment Analysis: Cost Decomposition

[Insert Figure: Pie chart or stacked bar showing the average decomposition of end-to-end latency into: (1) client encryption, (2) network/proxy transit, (3) gateway parsing, (4) queue wait, (5) PHE computation, for the baseline configuration (10 users, batch 10, 0ms latency).]

G. Ciphertext Expansion Analysis

VII. Discussion

A. Practical Viability

[Based on results, discuss: At what concurrency, batch size, and latency does PHE remain viable for production use? What SLA targets (e.g., P99 < 500ms) can be met?]

B. When PHE Makes Sense

The results suggest that Paillier-based API encryption is most practical when:

1) **Operations are limited to addition and counting.** These cover a significant fraction of real-world aggregation APIs (financial totals, sensor sums, vote tallies, analytics counters).
2) **Batch sizes are moderate.** **[Identify the practical batch size range from results.]**

TABLE VI: Batch Size Scaling Results

Batch	Median E2E (ms)	Encrypt (ms)	PHE Compute (ms)	Ciphertext (KB)	Expansion ×
1	[–]	[–]	[–]	[–]	[–]
10	[–]	[–]	[–]	[–]	[–]
100	[–]	[–]	[–]	[–]	[–]
1000	[–]	[–]	[–]	[–]	[–]

TABLE VII: Proxy Latency Impact Results

Latency (ms)	Median E2E (ms)	PHE Overhead (%)	Net Overhead (%)	Success (%)
0	[–]	[–]	[–]	[–]
10	[–]	[–]	[–]	[–]
50	[–]	[–]	[–]	[–]
100	[–]	[–]	[–]	[–]

3) **Network latency dominates.** In WAN deployments where network round-trip times exceed 50 ms, the relative overhead of PHE becomes a smaller fraction of total latency.
4) **Privacy requirements are strict.** When regulatory or contractual obligations prohibit plaintext exposure at any intermediary.

C. Comparison with Alternative Approaches

D. Impact of Real-World Proxy Diversity

[Discuss the impact of routing through 1,025 real proxies: HTTP vs. SOCKS5 vs. SOCKS4 reliability, geographic latency variation, connection failure patterns. This data is unique to this study.]

E. Limitations

- **Operation set:** Paillier supports only addition and scalar multiplication. Applications requiring comparisons, divisions, or non-linear operations are not supported.
- **Key size:** 1024-bit keys are used for experimental tractability. Production deployments would require 2048-bit or larger keys, approximately doubling computation time.
- **Single key pair:** Our experiments use a single key pair. Multi-tenant deployments would require key management infrastructure.
- **Simulated users:** All users run on the same machine, which does not capture true geographic distribution of clients.
- **No persistent state:** The gateway is stateless; time-series aggregation across requests would require additional architectural components.

F. Threats to Validity

Internal validity: All experiments use deterministic seeds and reproducible configurations. Metrics are collected independently at the client, gateway, and worker layers to enable cross-validation.

External validity: While our proxy pool provides real-world network diversity, all components run on a single machine. True distributed deployment would introduce additional variability.

Construct validity: Our latency measurements use `time.Now()` at nanosecond resolution. Clock skew is not a factor since all components share the same system clock.

TABLE VIII: Throughput Stress Test Results

Metric	Value
Duration	30 s
Concurrent users	1,000
Total requests completed	[–]
Peak throughput (req/s)	[–]
Mean throughput (req/s)	[–]
Median E2E latency (ms)	[–]
P99 E2E latency (ms)	[–]
Success rate (%)	[–]

TABLE IX: Failure Injection Results

Drop Rate	Observed Failures	Success Rate (%)	Median E2E (ms)
0%	[–]	[–]	[–]
1%	[–]	[–]	[–]
5%	[–]	[–]	[–]

VIII. Conclusion and Future Work

We presented PHE-Gateway, a privacy-first REST API gateway architecture using Paillier Partially Homomorphic Encryption, and conducted the first large-scale empirical evaluation of PHE-based API gateways under realistic, proxy-mediated network conditions.

Our experimental results, comprising **[N total]** request records across five experiment suites, demonstrate that:

1) PHE-based REST API gateways are **[viable/challenging]** for production workloads at batch sizes up to **[N]**, achieving **[X]** requests per second with median latency of **[Y ms]**.
2) Ciphertext expansion is bounded at **[Z]**×, which is **[acceptable/challenging]** for bandwidth-constrained deployments.
3) Client-side encryption dominates end-to-end latency at **[X%]**, while gateway-side PHE computation accounts for only **[Y%]**.
4) Under WAN-like conditions (50–100 ms proxy latency), PHE overhead becomes **[a minor/the dominant]** fraction of total latency.
5) The system degrades gracefully under 5% fault injection, with **[describe behavior]**.

A. Future Work

- **Distributed deployment:** Evaluate PHE-Gateway across multiple geographic regions with true network separation.
- **Multi-key support:** Extend the gateway to support per-tenant key management for multi-user scenarios.
- **Hybrid PHE/FHE:** Investigate using PHE for latency-sensitive operations and FHE for complex batch operations.
- **Hardware acceleration:** Explore GPU-accelerated modular arithmetic for Paillier operations.
- **Larger key sizes:** Evaluate 2048-bit and 4096-bit keys for production security levels.
- **Real-world case study:** Deploy PHE-Gateway in a production environment (e.g., financial API aggregation) and compare with baseline TLS-only architecture.

References

[1] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT '99*, ser. LNCS, vol. 1592. Springer, 1999, pp. 223–238.

TABLE X: Latency Cost Decomposition (Baseline: 10 Users, Batch 10)

Component	Mean (ms)	Fraction (%)
Client encryption	[–]	[–]
Network transit	[–]	[–]
Gateway processing	[–]	[–]
Queue wait	[–]	[–]
PHE computation	[–]	[–]
Total E2E	[–]	100%

TABLE XI: Ciphertext vs. Plaintext Size

Batch Size	Plaintext (bytes)	Ciphertext (bytes)	Expansion
1	8	[–]	[–]×
10	80	[–]	[–]×
100	800	[–]	[–]×
1000	8,000	[–]	[–]×

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st ACM Symposium on Theory of Computing (STOC)*, 2009, pp. 169–178.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Computation Theory*, vol. 6, no. 3, pp. 1–36, 2014.

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT 2017*, ser. LNCS, vol. 10624. Springer, 2017, pp. 409–437.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[7] S. Halevi and V. Shoup, "Algorithms in HElib," in *CRYPTO 2014*, ser. LNCS, vol. 8616. Springer, 2014, pp. 554–571.

[8] S. Halevi and V. Shoup, "HElib—an implementation of homomorphic encryption," https://github.com/homenc/HElib, 2020.

[9] Microsoft Research, "Microsoft SEAL (release 3.6)," https://github.com/microsoft/SEAL, 2020.

[10] A. Al Badawi *et al.*, "OpenFHE: Open-source fully homomorphic encryption library," in *Proc. 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022.

[11] C. Mouchet, J. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in Go," in *Proc. 8th Workshop on Encrypted Computing*, 2020.

[12] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy," in *ICML 2016*, pp. 201–210.

[13] F. Li, B. Luo, and P. Liu, "Secure information aggregation for smart grids using homomorphic encryption," in *Proc. 1st IEEE SmartGridComm*, 2010, pp. 327–332.

[14] O. Baudron, P.-A. Fouque, D. Pointcheval, J. Stern, and G. Poupard, "Practical multi-candidate election system," in *Proc. 20th ACM PODC*, 2001, pp. 274–283.

[15] S. Truex, N. Baracaldo, A. Anber, T. Gursoy, R. Joshi, and E. Wei, "A hybrid approach to privacy-preserving federated learning," in *Proc. 12th ACM AISec Workshop*, 2019, pp. 1–11.

[16] L. Chen, H. Jordan, H. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," NIST IR 8105, 2020.

[17] D. Boneh and V. Shoup, "A graduate course in applied cryptography," 2020, https://toc.cryptobook.us/.

[18] European Parliament and Council, "Regulation (EU) 2016/679 (General Data Protection Regulation)," *Official Journal of the European Union*, 2016.

[19] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[20] O. Goldreich, "Secure multi-party computation," manuscript, 1998.

[21] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[22] IPLocate, "Free verified proxy list," https://github.com/iplocate/free-proxy-list, 2024, accessed 2026.

TABLE XII: Comparison of Privacy-Preserving Approaches

Approach	Operations	Latency	Ciphertext Size	Trust Model	Maturity
TLS only	Any	Baseline	1×	Trust gateway	Production
PHE (Paillier)	Add/Count	[–]×	[–]×	No trust	This work
FHE (BFV/CKKS)	Any	10^3–10^6×	10^2–10^4×	No trust	Research
TEE (SGX/SEV)	Any	∼1×	1×	Trust HW vendor	Production
MPC	Any	10^2–10^4×	N/A	Distributed trust	Research