A Verification and Privacy Layer for Agent Economy

Primus Labs

March 12, 2026

Abstract

A new class of AI agents is emerging that can act, transact, and coordinate complex tasks on behalf of users. The opportunity is vast: autonomous agents promise to turn high-friction workflows into simple delegated operations across every sector of the digital economy. Yet a fundamental gap stands in the way. There is no general mechanism for a user to verify that an agent actually did what it promised, nor for an agent to compute on sensitive data without exposing it. Until that gap closes, the agent economy cannot scale beyond low-stakes, low-trust interactions.

Primus closes it. Using zkTLS, every agent action that touches an external service or data source produces a compact cryptographic proof of exactly what happened, replacing self-reported claims with independently verifiable evidence. Using fully homomorphic encryption, agents can process private inputs without ever seeing the plaintext. The combination gives the agent economy its missing trust layer: provable behavior and protected data in a single execution stack. With Primus, users gain the confidence to delegate high-value tasks, platforms gain the assurance to expand agent permissions, and new markets emerge around services that were previously too sensitive to automate.

1 The Problem

The agent economy is taking shape rapidly. Autonomous AI agents already book travel, execute trades, manage subscriptions, negotiate procurement terms, and coordinate multi-step workflows that span dozens of services. As their capabilities grow, so does the economic surface they control: agent-mediated transactions are projected to account for a significant share of digital commerce within a few years. Yet the infrastructure beneath this shift was never designed for a world where software acts on a person's behalf with real authority over money, data, and identity. Two structural deficits block the agent economy from reaching its potential.

Agents cannot prove what they did. When a user delegates a task to an agent, the agent calls APIs, reads data, places orders, and reports back. But the user receives only the agent's own summary. There is no independent, cryptographic record that the agent contacted the service it claimed, received the response it reported, or executed the steps it described. In low-stakes settings this gap is tolerable; in high-value or regulated contexts it is disqualifying. A wealth-management agent that claims to have rebalanced a portfolio, or a compliance agent that reports a clean KYC check, must be verifiable, not merely trusted. Without provable execution, every delegation carries hidden counterparty risk, disputes are expensive to resolve, and platforms must restrict agent permissions to a narrow, conservative set.

Agents cannot safely touch private data. Useful agent work almost always requires sensitive inputs: account balances, health records, salary data, proprietary pricing, customer lists, legal documents. Today, giving an agent access to such data means exposing it in the clear. The agent's operator, its hosting environment, and any intermediary in the pipeline can observe, log, or leak the information. Users face deanonymization, competitive harm, and regulatory liability; organizations risk breaching data-protection obligations the moment an agent reads a protected field. The rational response is to withhold data, but data-starved agents deliver little value. The result is a vicious cycle: privacy concerns suppress delegation,

limited delegation keeps agent services shallow, and shallow services fail to justify the trust investment needed to break the cycle.

These two deficits compound each other. An agent that cannot prove its actions and cannot protect the data it handles is doubly untrusted. Markets that depend on such agents remain thin, fees stay high, and the most valuable use cases, those involving real money, regulated data, or cross-organizational coordination, never get built. Below we decompose the consequences into four concrete risk categories.

- **Confidentiality Risk.** Every agent interaction is a potential data-exposure event. Agents routinely process credentials, personal identifiers, financial statements, and strategic signals. A single leak, whether from a compromised agent, an insecure integration, or an over-permissioned API call, can cause irreversible harm: identity theft, front-running, regulatory penalties, or loss of trade secrets. Because agents operate at machine speed across many services, the blast radius of a confidentiality failure is far larger than in traditional human-driven workflows.
- **Accountability Gap.** When an agent acts for a user, any dispute about what happened devolves into a he-said-she-said between the user, the agent operator, and the service provider. Without tamper-evident execution logs that are cryptographically tied to the agent's identity and the user's delegation, there is no neutral ground for adjudication. This raises the cost of dispute resolution, discourages high-value delegation, and forces platforms to impose blunt permission limits rather than fine-grained, evidence-based trust.
- **Regulatory Friction.** Agents that cross jurisdictional or sectoral boundaries must satisfy overlapping regimes: AML, KYC, GDPR, HIPAA, sector-specific licensing, and emerging AI-governance rules. Complying typically requires sharing data with verifiers or auditors, yet sharing raw data can itself violate privacy regulations. This paradox pushes platforms toward the lowest common denominator: restricting agent capabilities until compliance can be proven, which in practice means most capabilities stay turned off.
- **Fragmentation and Lock-in.** The data agents need is scattered across banks, SaaS platforms, government registries, enterprise systems, and personal devices, each behind its own authentication, format, and access-control model. Without a universal, privacy-preserving way to attest data authenticity and compose signals from heterogeneous sources, agents are confined to walled gardens. Cross-system orchestration, the scenario where agents deliver the most value, remains impractical.

These are not edge-case engineering concerns; they are structural barriers that define the ceiling of the agent economy today. Overcoming them requires a layer that makes every agent action independently provable, keeps sensitive data encrypted throughout computation, and fits into existing compliance and identity processes without requiring the world to change first. The remainder of this paper introduces Primus, which provides exactly that layer.

1.1 The Data-Access Dilemma

The most consequential agent tasks, underwriting a loan, verifying an identity, auditing a treasury, optimizing a supply chain, depend on private data that lives outside any single system. Credit histories sit with bureaus, balances with banks, identity documents with government registries, and business records inside enterprise ERPs. Bridging this data into agent workflows without sacrificing privacy or authenticity is the central unsolved problem of the agent economy.

- **Exposure by Default.** Current integration patterns require data to leave its source in plaintext before an agent can act on it. Every hop, from source API to agent runtime to downstream consumer, is a point of potential leakage. Privacy-preserving architectures must invert this default: data should remain encrypted or committed at the source, and agents should compute on it without observing it.
- **Authenticity Without Trust.** An agent that bases a decision on forged or stale data can cause as much damage as one that leaks real data. Yet most APIs provide no cryptographic proof that a response is genuine, recent, and unmodified. Agents and their counterparties need tamper-evident attestations of data provenance, not trust in the agent's self-report.
- **Fragmentation at Scale.** Useful agent decisions often require fusing signals from five, ten, or fifty heterogeneous sources. Each source has its own schema, authentication mechanism, rate limits, and terms of service. Without standardized, privacy-preserving connectors,

every new integration is a bespoke engineering project, and cross-source composition, the scenario where agents add the most value, remains prohibitively expensive.

- **Consent and Lifecycle Control.** Users and data owners must retain fine-grained control over what data an agent may access, for how long, and under what conditions. Revocation must be immediate, audit trails must be verifiable, and data-minimization principles must be enforceable at the infrastructure level, not merely promised in policy documents.

Together, these constraints define the requirements for a practical trust-and-privacy layer: selective disclosure, provable data authenticity, confidential computation, composable cross-source access, and enforceable consent governance. Primus implements each of these capabilities and integrates them into a coherent execution stack for the agent economy, as described in the sections that follow.

2 The Solution

Primus delivers a comprehensive trust, verification, and privacy layer for the agent economy by integrating two advanced cryptographic technologies: Zero-Knowledge Transport Layer Security (zkTLS) for verifiable, privacy-preserving access to data and API calls, and Fully Homomorphic Encryption (FHE) for confidential computation and storage. Together, these technologies allow agents to prove what they did, access data and services without relying on self-reported claims, and compute over sensitive inputs without exposing them. This section formally describes these technologies, reviews related work, and details Primus's innovations and architectural framework.

2.1 Zero-Knowledge Transport Layer Security (zkTLS)

Transport Layer Security (TLS) [1, 2] is the dominant protocol for securing client-server communications on the Internet. TLS provides confidentiality and integrity for a session between a client, denoted C, and a server, denoted S. However, TLS was not designed to produce third-party attestations of application-layer payloads: it guarantees authenticity only between the endpoints of a session. If a client later presents TLS-protected data to a third-party application, a malicious client can forge transcripts that appear valid, because TLS does not bind application-level usage to external verifiers. This gap follows from the implicit assumption that both C and S follow the protocol; when the client is untrusted, that assumption no longer holds.

Common web mitigations require the user to mediate sharing between sites (for example, OAuth-style flows or direct site-to-site authentication). Those approaches work in practice but have two drawbacks for our setting: they often force users to disclose more information than necessary (for example, releasing an entire credit score to prove a threshold), and they require additional web infrastructure or protocol changes that complicate deployment. These limitations make such mitigations poorly suited for verifiable, privacy-preserving reuse of Web2 data in Web3 applications.

zkTLS, sometimes called web proofs, is a protocol that produces cryptographic attestations tying TLS-protected web data to an external verifier while preserving confidentiality. In this setting a client that fetches data from a TLS server can obtain an attestation that the observed plaintext originates from that server and corresponds to the recorded TLS transcript, without exposing unrelated secrets. Practical deployments introduce an attestor (or notary) that assists attestation generation or witnesses the session; the attestor learns only what the attestation intentionally discloses, and the server requires no protocol changes. DECO [3] initiated this line of zkTLS research by demonstrating different approaches for producing attestations from recorded TLS transcripts; although the original constructions incurred substantial overhead, they introduced core techniques that subsequent work has refined.

2.1.1 Overview of TLS

We recall the minimal TLS notions needed for zkTLS. TLS runs between a Client and a Server and separates (1) a cryptographic Handshake that establishes keying material and (2) the Record layer that protects application data using AEAD ciphers.

Handshake Phase The Handshake produces an authenticated shared secret: in informal terms a key-exchange yields a premaster secret pms. Session secrets used for traffic encryption and authentication are derived from pms together with the public handshake transcript (randoms, nonces and transcript hashes) via

the protocol's KDF. In TLS 1.3 this KDF is instantiated with HKDF (with separate "extract" and "expand" steps and explicit transcript binding); in TLS 1.2 the KDF is the PRF described in RFC 5246. Abstractly one may write

$$\text{secrets} \leftarrow \text{KDF}(\text{pms}, \text{transcript}),$$

where transcript denotes the public handshake material that is bound into derived keys. This binding is the reason TLS attests to the session transcripts: keys are a deterministic function of pms and the transcript.

Record Phase Once traffic keys are derived, the Record layer protects application data with an AEAD such as AES-GCM. Let K denote the application traffic key and let $m_0, \ldots, m_\ell$ be plaintext blocks; AES-GCM produces ciphertext blocks c_i together with an authentication tag τ :

$$c_i = \text{AES\_CTR}(K, \text{ctr}_i) \oplus m_i, \; i = 0, \ldots, \ell,$$

and

$$\tau = \text{GCM\_Tag}(K, c_{0:\ell}, \text{AD}),$$

where AD denotes any associated data (e.g., record headers) and ctr_i are per-record nonces/IVs. Correct decryption requires verification of τ before recovering the plaintexts. Many zkTLS constructions must reason about the non-committing behavior of AES-GCM and therefore either prove key derivation consistency or leverage public plaintext structure (padding/status lines) when constructing proofs. See RFC 5288 for AEAD particulars.

2.1.2 zkTLS Modes

zkTLS protocols broadly use two deployment modes: MPC-Mode and Proxy-Mode. Both achieve verifiable attestations for TLS-protected data but trade off privacy, performance, and network trust in different ways. Below we give concise high-level flows for each mode and summarize their pros and cons to guide design choices.

MPC-Mode In MPC-Mode the Client and Attestor jointly simulate the client's behavior for the TLS Handshake and, selectively, portions of the Record layer using two-party computation (2PC). The goal is to obtain attestations without exposing client secrets: the Client retains confidentiality of secrets (credentials, cookies, private keys) while the Attestor gains cryptographic assurance that observed ciphertexts correspond to claimed plaintexts. See Figure 1 and the following description for an overview.

1. 2PC on handshake: the Client and Attestor run a 2PC protocol that evaluates the KDF over the premaster secret and public handshake transcript. Neither party learns the full derived keys; instead they obtain secret shares (or masked values) of the session secrets.
2. 2PC on TLS request : to avoid revealing request plaintexts or keys, the Client and Attestor jointly evaluate the symmetric encryption and tag computation (AES/GCM) for the outgoing request under 2PC, producing a ciphertext the Client can send to the Data Source while keeping keys secret-shared.
3. Server response relay: the Client receives the response ciphertext from the Data Source and forwards it (or its transcript) to the Attestor for verification without locally reconstructing the keys.
4. Key-reconstruction and proof: after the Attestor has observed the response ciphertexts and performed any necessary checks, it releases its key shares to the Client. The Client reconstructs the full session keys and produces an interactive proof (for example, QuickSilver's iZK) that the recorded ciphertexts decrypt to the claimed plaintext and that the AES key used is consistent with the KDF output used in the Handshake 2PC.
5. Attestation issuance: upon successful verification of the proof, the Attestor issues a signed attestation binding the claimed plaintext to the recorded TLS transcript for external consumption (for example, on-chain submission).

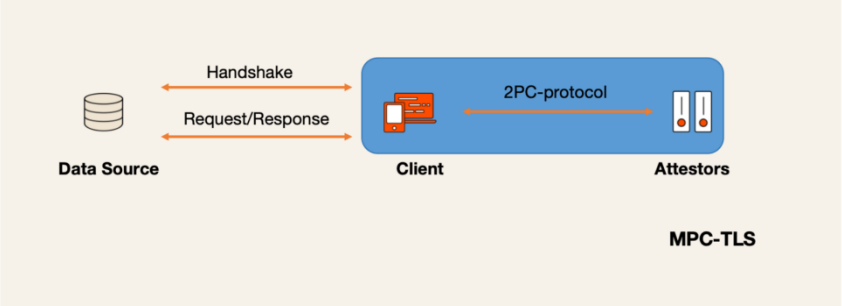

Figure 1: MPC-Mode high-level flow

Key points: The following items summarize the mechanism, privacy guarantees, performance cost, network robustness, and operational mitigations for MPC-Mode.

- **Mechanism:** Client and Attestor jointly execute 2PC to evaluate the KDF and any symmetric primitives (AES-GCM) needed to form and verify requests and responses. Parties typically hold secret shares or masked values of session keys rather than reconstructing them during protocol execution.
- **Privacy guarantees:** The Client's long-term secrets (credentials, cookies, private keys) remain private; the Attestor learns only the minimal information revealed by the 2PC outputs and the final attestation.
- **Performance cost:** Running 2PC for KDFs and AEAD incurs nontrivial CPU, memory, and bandwidth overhead. Optimizations (semi-honest garbled circuits, OT extensions, and garble-then-prove pipelines) reduce but do not eliminate this cost compared to native TLS.
- **Network robustness:** MPC-Mode reduces reliance on the Attestor's network position, the Attestor need not act as an inline proxy, lowering the attack surface from network-level manipulation.

Proxy-Mode In Proxy-Mode the Attestor is positioned on the network path (inline or as a trusted recorder) so it can archive authentic TLS handshakes and ciphertexts and later verify statements about those archives. See Figure 2 for an overview. This mode trades increased network trust for substantially lower runtime cryptographic cost compared to MPC-Mode.

1. Session capture: the Client establishes a TLS session with the Data Source while the Attestor proxies or passively records the handshake and subsequent record ciphertexts, computing a transcript hash and attaching a signed timestamp.
2. Client proof construction: the Client obtains response ciphertexts and constructs a zero-knowledge proof over the archived transcript demonstrating the claimed plaintext properties (KDF→ AES consistency or public-padding statements).
3. Verification and attestation: the Client sends the proof to the Attestor; the Attestor verifies it against the archived transcript and, on success, issues a signed attestation that can be consumed by on-chain or off-chain verifiers.
4. Publication (optional): to increase transparency, the Attestor or Client may publish the attestation or its succinct commitment (for example, a Merkle root or transcript hash) to a public ledger.

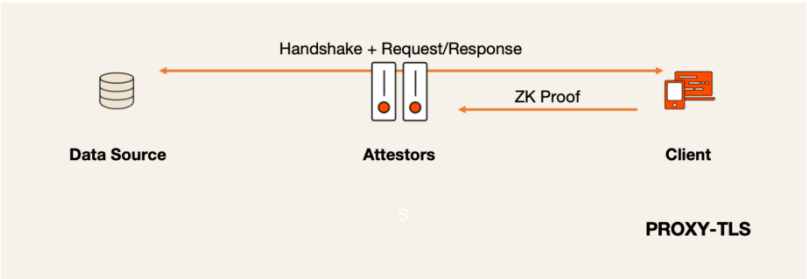

Figure 2: Proxy-Mode high-level flow

The Client can demonstrate correctness of claimed plaintexts using one of two complementary proof strategies:

- **Option 1: KDF-and-AES consistency.** The Client proves that the AES application key used to produce the recorded ciphertexts is the result of the protocol KDF applied to the premaster secret pms together with the public handshake transcript, and that decrypting the recorded ciphertexts under that key yields the claimed plaintexts.
- **Option 2: Public-padding and AES.** When responses contain known, fixed public structure (for example, a canonical HTTP status line or other predictable headers), the Client proves that selected ciphertext blocks correspond to the known public plaintext while the remaining blocks encrypt the secret payload, all under a single AES key.

Both approaches mitigate AES-GCM's non-committing behavior by tying ciphertexts to a single, proven key. Option 1 provides the stronger guarantee by cryptographically binding the key to the Handshake via the KDF; Option 2 is more efficient for responses with sufficient public structure but requires careful handling when such structure is absent. In either case, the proof must ensure key consistency across record blocks and, where applicable, that the key matches the KDF output established during the Handshake.

This mode has the following properties and tradeoffs:

- **Mechanism:** the Attestor forwards or observes traffic, stores the handshake transcript, associated record ciphertexts, and relevant metadata, and verifies client-submitted zero-knowledge proofs that tie claimed plaintexts to archived ciphertexts.
- **Performance:** proof generation on archived transcripts is usually far cheaper than full 2PC per handshake; proof cost depends on the chosen statement (KDF → AES vs public-padding) and the proving system.
- **Trust assumptions and mitigations:** Proxy-Mode assumes the Attestor has an authenticated path to the real Data Source. Mitigations for Attestor misbehavior include channel binding to the Attestor's TLS session, certificate transparency checks, OCSP/CT evidence, signed timestamps on archives, and optional third-party snapshotting of transcript hashes.

2.1.3 Primus zkTLS protocol

Primus proposes a highly optimized zkTLS pipeline based on garble-then-prove [4], applicable to both MPC-Mode and Proxy-Mode. It leverages the state-of-the-art VOLE techniques from QuickSilver [5] to amortize preprocessing time and dramatically reduce online symmetric and communication costs. Primus attains substantially improved concrete performance in proving runtime, peak memory, and prover-verifier communication.

- **Garble-then-prove workflow:** Instead of deploying a fully malicious 2PC protocol, we exploit TLS-specific semantics to relax adversarial requirements during emulation. A corrupted Verifier must not learn session keys during emulation, whereas a corrupted Prover may learn limited information that is harmless once the session completes. Primus requires that any Prover deviation be detectable at checking time: after emulation the Prover opens required shares and proves in zero-knowledge (via an interactive proof, e.g., QuickSilver) that the garbled execution produced outputs consistent with the committed inputs and the public transcript. This design removes expensive malicious 2PC machinery while preserving integrity guarantees for zkTLS.
- **TLS-specific optimizations:** We reduce circuit size and cost by carefully selecting which intermediate values may be revealed without enlarging attacker advantage. This yields more than a 2 × reduction in handshake circuit size in many cases. We also remove GMAC/AES tag computation from the main Boolean circuit: using Oblivious Linear Evaluation with errors (OLEe) to produce additive sharings of the random powers required for GMAC reduces GMAC cost by orders of magnitude while exposing at most negligible leakage that does not affect concrete security for our targets.
- **GMAC via OLE:** Instead of implementing GMAC entirely inside a Boolean circuit, Primus uses Oblivious Linear Evaluation (OLE) to produce additive sharings of the linear terms required by GMAC/AES-GCM. Treating tag computation as essentially linear work

lets us replace expensive per-wire Boolean operations with cheap field operations, yielding practical GMAC cost reductions by over two orders of magnitude.

- **Commitment conversion:** To connect garble-then-prove runs to zkSNARK-friendly checks, Primus converts information-theoretic MACs (IT-MACs) produced during execution into additive commitments suitable for SNARKs. We first map Boolean IT-MACs to arithmetic IT-MACs over a large field, then convert arithmetic IT-MACs to SNARK-friendly additive commitments; both steps are efficient and keep the malicious-case overhead close to the semi-honest cost. This pipeline enables hybrid proofs that combine garble-then-prove with occasional SNARK checks when needed.
- **Empirical improvements:** We implemented the protocol end-to-end and evaluated it across x86, ARM, and browser (WASM) platforms. The implementation outperforms DECO by over an order of magnitude (roughly 14 × communication improvement and 7.5 ×—15× runtime speedups in reported experiments). Primus achieves low, predictable prover latency (online time often measured in seconds) for real API payloads and scales stably with response size. These empirical results drove our choice to favor garble-then-prove plus QuickSilver for browser-centric deployments while retaining targeted key-consistency checks for high-risk assertions [4, 6].

2.1.4 Benchmarks

We benchmark several open-source zkTLS implementations: Primus, TLSNotary, Reclaim, and Pluto [7, 8, 9, 10]. Benchmarks were collected with our open framework (see [11]) across x86, ARM, and Browser (WASM) platforms; the framework supports configurable network parameters and workload sizes.

To simulate representative web access (browser clients sending cookies), we set request sizes to 1 KB or 2 KB and varied response sizes from 16 B to 2 KB. Actual performance depends on network and platform configuration; the runs reported below used the following environment: bandwidth 200 Mbps, latency 10 ms, request size 2 KB, Ubuntu 22.04, 8 vCPUs @ 3.10 GHz, 32 GB RAM. Results are reported for several representative implementations below.

MPC-Mode: We compare Primus and TLSNotary, both of which implement the MPC mode for TLS 1.2.

Response Size (Bytes)	Uploaded Size (KBytes)	Downloaded Size (KBytes)	Native (x86) Runtime (s)	Native (arm) Runtime (s)	Browser Runtime (s)	Memory (MBytes)
16	34,888	1,141	2.56	2.64	5.05	146
256	34,901	1,141	2.57	2.62	5.09	146
1024	34,930	1,141	2.57	2.64	5.04	146
2048	34,971	1,141	2.58	2.65	5.11	146

Table 1: Primus (MPC-Mode) benchmark results

Response Size (Bytes)	Uploaded Size (KBytes)	Downloaded Size (KBytes)	Native (x86) Runtime (s)	Native (arm) Runtime (s)	Browser Runtime (s)	Memory (MBytes)
16	61,089	62,144	20	32	47	147
256	61,121	65,309	20	33	48	148
1024	61,222	75,437	21	35	49	148
2048	61,356	88,940	23	39	53	148

Table 2: TLSNotary (MPC-Mode) benchmark results

1. Both Primus and TLSNotary have implemented MPC-Mode protocols for TLS 1.2, and both are memory-efficient across different platforms. In both implementations, the browser runtime is approximately **2x** slower than native executions.
2. Under these metrics, Primus is approximately **10x** faster than TLSNotary in total runtime across different platforms. Primus maintains a stable runtime regardless of response size, whereas TLSNotary experiences an increase in both download size and runtime as the response size grows, though the variation remains relatively small.

3. The primary reason for the benchmark results above is likely Primus's use of QuickSilver, optimized circuits, and an efficient Garbled Circuit (GC) implementation. **Notably, the TLSNotary team is also integrating QuickSilver with support from the Primus team.** Rough estimates suggest that adopting QuickSilver will significantly reduce download size and lower the computation overhead from approximately 3GC (running GC three times) to 2GC.

Proxy-Mode: In Proxy-Mode benchmarks we set the request size to zero so that implementations do not prove request data; this focuses the comparison on response-size costs. We compare Primus (Proxy-Mode, TLS 1.2) with Reclaim (Proxy-Mode, TLS 1.2/1.3) and Pluto (Proxy-Mode, TLS 1.3).

Response Size (Bytes)	Uploaded Size (KBytes)	Downloaded Size (KBytes)	Native (x86) Runtime (s)	Native (arm) Runtime (s)	Browser Runtime (s)	Memory (MBytes)
16	375	788	0.45	0.48	2.31	71
256	383	788	0.46	0.47	2.34	75
1024	416	788	0.47	0.49	2.48	91
2048	461	788	0.48	0.52	2.66	111

Table 3: Primus (Proxy-Mode, TLS 1.2) benchmark results

Response Size (Bytes)	Uploaded Size (1.2/1.3 KBytes)	Downloaded Size (1.2/1.3 KBytes)	Native (x86) Runtime (1.2/1.3 s)	Native (arm) Runtime (1.2/1.3 s)	Browser Runtime (1.2/1.3 s)	Memory (MBytes)
16	32/23	31/22	4.09/3.62	8.79/8.16	13.88/13.05	356
256	34/24	32/23	5.09/4.76	11.06/10.64	18.76/21.43	313
1024	38/29	38/29	8.53/8.43	18.71/19.10	36.63/49.21	298
2048	44/35	44/36	12.92/13.38	28.61/29.83	60.22/86.21	306

Table 4: Reclaim (Proxy-Mode, TLS 1.2/1.3) benchmark results

Response Size (Bytes)	Uploaded Size (KBytes)	Downloaded Size (KBytes)	Native (x86) Runtime (s)	Native (arm) Runtime (s)	Browser Runtime (s)	Memory (MBytes)
16	62	14	41	56	193	1395
256	62	14	41	57	206	1395
1024	63	15	58	89	293	1391
2048	65	16	76	119	390	1412

Table 5: Pluto (Proxy-Mode, TLS 1.3) benchmark results

1. Primus implements Option 1 in Proxy-Mode for TLS 1.2, while Pluto adopts Option 1 for TLS 1.3 using the ChaCha20-Poly1305 cipher suite, primarily because ChaCha20 is more compatible with their IVC system. Reclaim, on the other hand, implements Option 2 for both TLS 1.2 and TLS 1.3.
2. The total communication size of Reclaim and Pluto is nearly identical, with both being up to **18x** smaller than Primus, as expected given their use of zkSNARKs.
3. The total runtime of Primus is **5x** to **30x** faster than Reclaim across different platforms. Additionally, Primus consumes **3x** to **5x** less memory, though both remain lightweight and sufficient for most applications. Pluto, however, is significantly slower than both Primus and Reclaim, over **90** × slower than Primus, with substantially higher memory consumption. This is due to its use of IVC, which is highly computationally intensive.
4. The primary reason for the benchmark results above is likely Primus's use of Quick-Silver, while Reclaim relies on Groth16 and Pluto utilizes Nova, both of which are computationally intensive. This also explains why Primus's runtime remains relatively stable regardless of response size, whereas Reclaim and Pluto's runtime are highly sensitive to response length. **Note that, as some repositories are continuously updated, performance metrics may vary over time.**

These results are based on internal benchmarks and may vary depending on implementation and environment.

2.2 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) is a cryptographic technique that enables computation on encrypted data without requiring decryption. In an FHE scheme, data is encrypted into ciphertexts, and mathematical operations can be performed directly on these ciphertexts. The result of these operations, when decrypted, matches the outcome of performing the same operations on the original plaintext data. This property allows sensitive data to remain confidential throughout the computation process, making FHE particularly valuable for scenarios where data privacy is paramount.

2.2.1 Survey of FHE schemes

FHE research has produced multiple families of constructions, each optimized for differing computational patterns and deployment constraints. The following paragraphs summarize representative schemes and their principal tradeoffs. Citations are provided to foundational descriptions and practical implementations.

BGV and BFV (packed integer arithmetic) The BGV and BFV families are lattice-based schemes that support SIMD packing and efficient exact integer arithmetic [12, 13]. These schemes achieve high throughput for workloads that can be expressed as wide, vectorized operations over many independent slots. Their principal advantages are exact arithmetic semantics and well-studied parameter choices for batching. Their limitations include noise accumulation that constrains circuit depth, comparatively costly rotation and relinearization operations for nontrivial data movement, and bootstrapping costs that make arbitrary-depth evaluation less attractive without careful parameter selection. As a result, BGV/BFV are most suitable when the dominant workload is batchable integer arithmetic rather than fine-grained Boolean logic.

CKKS (approximate arithmetic) CKKS provides approximate arithmetic for real and complex values, enabling compact encodings for linear-algebra and machine-learning workloads [14]. CKKS ciphertexts are space-efficient for vectorized numerical computation and permit natural homomorphic evaluation of dot-products and matrix operations. The tradeoffs are the inherent approximation error, which grows with circuit depth and rescaling operations, and the need for careful precision management and (often) bootstrapping to preserve result fidelity. CKKS is therefore appropriate for encrypted inference or analytics where controlled approximation is acceptable, but it is less appropriate when exact, deterministic on-chain verification is required.

GSW / TFHE (gate-bootstrapping and bitwise evaluation) GSW introduced a matrix-style representation of ciphertexts that facilitates homomorphic gate evaluation; subsequent systems such as FHEW and TFHE refined these techniques to provide fast, low-latency bootstrapping and efficient per-gate evaluation [15, 16, 17]. These constructions excel for Boolean circuits: they support rapid, gate-level bootstrapping that effectively removes noise as part of evaluation, yielding predictable latency for arbitrarily deep circuits. The main disadvantage is that ciphertexts are typically managed at the bit level, which increases per-bit storage and communication compared with packed arithmetic schemes when performing wide numeric operations.

Design considerations for blockchain deployments Blockchain and smart-contract environments impose several practical constraints: verifiability and determinism of evaluation, low and predictable verification latency, constrained on-chain storage and gas costs, and straightforward mapping from application logic to homomorphic operations. Under these constraints, TFHE-style gate-bootstrapping is attractive: it provides deterministic correctness for discrete predicates, predictable latency due to on-the-fly bootstrapping, and a direct mapping from Boolean logic to homomorphic evaluation without elaborate noise management. By contrast, BGV/BFV are preferable when large, exact integer vectors can be processed off-chain and only succinct commitments or aggregated results are published; CKKS is preferable for approximate numerical workloads such as encrypted ML inference.

Accordingly, and in light of the target use cases considered in this work, Primus adopts a TFHE-centric FHE strategy as the pragmatic default for on-chain primitives.

2.2.2 The Verifiability of FHE Operations

Standard FHE guarantees that a designated computation applied to encrypted inputs yields a deterministically correct ciphertext, while preserving input confidentiality. However, canonical FHE

constructions do not provide public, efficiently checkable evidence that a specific ciphertext result indeed corresponds to the execution of a particular program on particular inputs. In permissionless settings this shortcoming is consequential: on-chain verifiers and external auditors cannot afford to re-execute costly homomorphic computations, and delegating acceptance to untrusted executors undermines the integrity guarantees of distributed ledgers. We therefore require verifiable FHE constructions to satisfy the following desiderata for blockchain deployment:

1. **Minimal impact on primitive performance:** any verifiability mechanism should avoid substantial changes to the underlying FHE primitive that would materially increase per-gate or per-ciphertext latency or reduce throughput.
2. **Succinct, public proofs:** proofs produced alongside evaluation must be compact and publicly checkable; verification should be feasible for constrained verifiers (including smart contracts) without re-executing the computation or relying on secret material.
3. **Practical proving time:** the time and computational resources required to produce proofs must be compatible with real-world service latencies and economic constraints; prohibitive prover costs or high latency render a design impractical regardless of theoretical security.

A widely studied approach composes homomorphic evaluation with an external succinct argument system: the prover constructs a SNARK attesting that the homomorphic evaluation transcript (or an arithmetic circuit that models the evaluation and any consistency checks) is correct. This composition produces compact, publicly checkable proofs whose verification cost is minimal and therefore attractive for on-chain validation.

Crucially, however, composing FHE with off-the-shelf, general-purpose SNARKs is not feasible at practical scale. Translating FHE operations into SNARK witnesses and proving the resulting enormous arithmetic circuits incurs prohibitive CPU, memory, and wall-clock costs. Empirical experience indicates proving times for nontrivial FHE workloads commonly range from hours to days on commodity hardware, which is incompatible with latency-sensitive blockchain applications. Therefore, practical verifiable FHE requires SNARK systems designed specifically for FHE operations rather than reliance on general-purpose SNARK backends.

2.2.3 HasteBoots: Primus Verifiable FHE

In outsourced, privacy-preserving computation, FHE provides confidentiality for data delegated to untrusted executors but does not by itself produce public, succinct evidence that the delegated computation was performed correctly. Clients and external verifiers thus require compact, efficiently checkable attestations that a specified program was executed on specified ciphertexts and that the published result corresponds to that execution.

Existing solutions based on general-purpose SNARKs or on compiling FHE implementations into zkVMs are not practical for realistic FHE workloads. Compiling high-level FHE code to generic circuits expands witness size and constraint counts substantially; direct SNARK verification of core FHE kernels (notably bootstrapping) continues to incur proving times on the order of minutes to hours. These inefficiencies derive from two structural mismatches: (a) FHE evaluation depends on specialized algebraic relations (NTT, modulus switching and key-switching, accumulator updates, ring↔polynomial conversions) that generic arithmetizations express inefficiently, and (b) FHE internal state is organized into bits, limbs, and ciphertext components, so naïve per-bit witness encodings produce large multiplicative blowup in circuit size and prover resources.

HasteBoots Contributions The principal contributions of HasteBoots [18] are as follows. HasteBoots is a succinct argument system tailored to attest the correctness of FHE evaluation that includes bootstrapping, and it is designed to minimize prover cost while producing compact, publicly verifiable proofs.

- **PIOP/PCS proof architecture:** We present a proof construction formulated within the Polynomial Interactive Oracle Proof (PIOP) model and instantiated with a Polynomial Commitment Scheme (PCS). The architecture decomposes FHE evaluation (including bootstrapping) into modular subprotocols whose correctness can be checked via succinct polynomial-time openings.

- **FHE instantiation and parameter regime:** We justify the selection of FHEW/TFHE as the target FHE primitive on account of its efficient gate-bootstrapping semantics, and we specify parameter regimes that strike a balance between cryptographic security and practical proof-generation cost.
- **Specialized PIOPs and algorithmic gadgets:** We develop bespoke PIOPs and arithmetic gadgets for core FHE operations: an optimized prover for Number Theoretic Transforms (including bit-reversal), a batched-NTT reduction that compresses batched monomial transforms to sparse linear queries, and a modulus-switching proof that reduces verification to inexpensive range and consistency checks while preserving soundness with respect to noise accumulation.
- **PCS packing and amortization techniques:** We introduce a PCS instantiation leveraging linear-code based commitments combined with packing strategies that amortize polynomial opening costs across many small polynomials, improving prover throughput without imposing restrictive field assumptions.
- Asymptotic and empirical complexity: We prove that, under the chosen parameterization and protocol decomposition, HasteBoots attains prover complexity that scales linearly in the size of the evaluated FHE computation. A reference implementation corroborates these claims: on Apple M4 hardware our implementation produces a proof for an FHE NAND with bootstrapping in roughly 5 seconds, representing a substantial practical improvement over naïve compositions with general-purpose SNARK backends.

2.2.4 HasteBoots Benchmarks

We evaluate the performance of HasteBoots on an Apple M4, measuring the proving time, verification time and proof size for each FHE operation listed in 6. As shown in the results, the bootstrapping procedure is the most time-consuming part in the NAND circuit. Within the bootstrapping procedure, accumulator updates dominate the cost, accounting for 95% of the total time due to its complex workflow. Overall, HasteBoots achieves efficient proving time around 5.13 seconds, verification time of 219 ms, and a proof size of approximately 107 MB.

FHE Operation		Proving Time	Verification Time	Proof Size
LWE Addition		17 ms	6 ms	11 MB
Bootstrapping	Mono-NTT	151 ms	36 ms	35 MB
	Acc Update	5.02 s	219 ms	107 MB
	Modulus Switching	22 ms	7 ms	12 MB
NAND		5.13 s	219 ms	107 MB

Table 6: Performance of HasteBoots on Apple M4

Proving Time v.s. Proof Size To better understand the performance trade-offs in our SNARG construction, we isolate the contributions of the PCS and PIOP components in 7. The PCS cost consists of the polynomial commitment time and the time to generate evaluation proofs for large committed polynomials.

With the Breakdown PCS, the prover spends around 25% of the total proving time on PCS, while PCS accounts for 95% of the verifier's total time and proof size. To explore the trade-off between proof size and proving time, we replace the Breakdown PCS with BaseFold, using its implementation in Jolt-b. As shown in 7, this substitution significantly reduces both the verification time and proof size, albeit at the cost of increased proving time. Nonetheless, even with this trade-off, our total proving time remains faster than the state-of-the-art result, around half an hour, reported in [19].

		Proving Time	Verification Time	Proof Size
NAND w/ Brakedown	PCS	1.39 s	212.5 ms	104 MB
	PIOP	3.75 s	6.9 ms	3 MB
	SNARG	5.13 s	219.4 ms	107 MB
NAND w/ BaseFold	PCS	63.74 s	4 ms	8 MB
	PIOP	3.75 s	6.9 ms	3 MB
	SNARG	67.49 s	10.9 ms	11 MB

Table 7: Comparison using Brakedown PCS vs. BaseFold PCS

Proving Time	**RISC0** [20]	**SP1** [21]	[19]	**HasteBoots**
M3 Pro (8 cores)	–	–	40 min	**7 s**
C61.meta (128 cores)	–	–	21 min	**5 s**
Hpc7a.96xlarge (192 cores)	4600 min	1500 min	18 min	**4 s**
M4 Pro	–	–	–	**5 s**

Table 8: Proving time for a single bootstrapping procedure across various devices

Comparison with Prior Work We compared the performance of HasteBoots with previous work on proving the FHE bootstrapping, as summarized in 8. zkVM-based approaches such as RISC0 [20] and SP1 [21], require roughly 3 days and 1 day of proving time, respectively, primarily due to performance losses when compiling complex FHE pro- grams. Thibault and Walter [19] design a SNARK-friendly arithmetic circuit for TFHE bootstrapping procedure and prove it with Plonk, reducing proving time to around half an hour. However, as a general-purpose SNARK, Plonk still suffers from some inefficiency when applied FHE-specific operations. In contrast, avoiding proving the full arithmetic circuit, HasteBoots directly prove the relation between the inputs and the outputs of these FHE operations, enabling sub-10 second proof generation. Concretely, [19] proves an explicit NTT circuit of size $O(N \log N)$, resulting in at least quasi-linear prover time. Our approach, by contrast, proves the NTT relation directly, avoiding circuit encoding and achieving an optimal linear-time prover. Moreover, [19] employs a decomposition method to compute multiplication by a monomial, leading to another circuit of size $O(N \log N)$, while our proto- col achieves a linear-time protocol for batched NTT operation on monomials. Finally, [19] introduces additional noise to FHE ciphertexts in modulus switching, which is avoided in our protocol. [19] achieves a verification time of approximately 8 ms and a proof size of around 200 KB, whereas HasteBoots requires roughly 219 ms and 107 MB. Notably, both the proof size and verification time of HasteBoots can be further optimized by adopting recursion techniques and replacing Brakedown with a more efficient PCS.

3 The Design of Primus Network

In this section, we present the architecture and key components of the Primus Network, a trust, verification, and privacy layer designed for the agent economy. The Primus Network leverages advanced cryptographic techniques, including zkTLS and FHE, to enable verifiable agent actions and confidential data processing while maintaining compatibility with existing blockchain infrastructures.

Primus serves as a dedicated verification and computation layer for the agent economy, enabling agents to access data, execute API-driven actions, and perform confidential computation through specialized, economically aligned infrastructure. Its principal design objectives are: (i) verifiable data and action provenance, (ii) confidentiality of computation, (iii) fault tolerance and high availability, and (iv) explicit economic incentives for network operators.

3.1 The Data Verification Layer

The Data Verification Layer implements zkTLS to provide verifiable guarantees of data authenticity, API-call integrity, and action provenance while preserving confidentiality of the underlying content. This layer

is modular and extensible, enabling a wide range of agent-economy workflows in which agents must securely access data, interact with external services, and return independently verifiable results.

3.1.1 Primary roles

The Primus Network distinguishes several participant roles with narrowly scoped responsibilities:

Developer Integrators who embed Primus capabilities into applications via the Primus SDK. Developers construct task payloads (data sources, verification predicates, expected output forms, and payment parameters), initiate secure rendezvous and submission flows, and consume and verify returned attestations and proofs.

Client The entity that initiates verification requests and consumes attestations (this may be an end user, a wallet, or a gateway acting on behalf of users). Clients fund tasks (depositing fees or payment commitments), select task preferences (for example privacy or latency), and validate attestations received from attestors.

Attestor node A computing node that executes zkTLS tasks and produces compact, verifiable attestations. Attestor operators must register and post collateral according to registry policy. Nodes run attestor software (including web and mobile runtimes) inside a protected environment, earn fees for correctly completed tasks, and may be penalized (for example via slashing) for incorrect or missing work.

Security of Attestor Nodes Attestor node security is essential to protect node operators and clients and to enforce the computational integrity of off-chain zkTLS execution. Because the base zkTLS protocol does not by itself provide an execution-integrity enforcement mechanism, Primus augments the software stack with hardware-backed assurances. Specifically, the network integrates Phala's TEE platform to constrain runtime behavior, produce verifiable attestation of execution, and protect critical secrets.

Key Management Node keys are managed by a KMS running inside the TEE. Keys are generated and stored only within the enclave and never exposed to the host operating system. All signing and sensitive cryptographic operations required for zkTLS sessions and attestations are performed inside the TEE and gated by evidence that the node executed the correct protocol code and inputs.

Version and Deployment Controls To mitigate risks from code injection or supply-chain attacks, only signed, officially released software images are permitted for attestor deployment. The TEE enforces runtime integrity via remote attestation and rejects unverified binaries. Upgrade and deployment processes rely on signed releases and reproducible build artefacts, and operators must follow authenticated update channels to ensure only approved versions run inside enclaves.

3.1.2 Network architecture and workflow

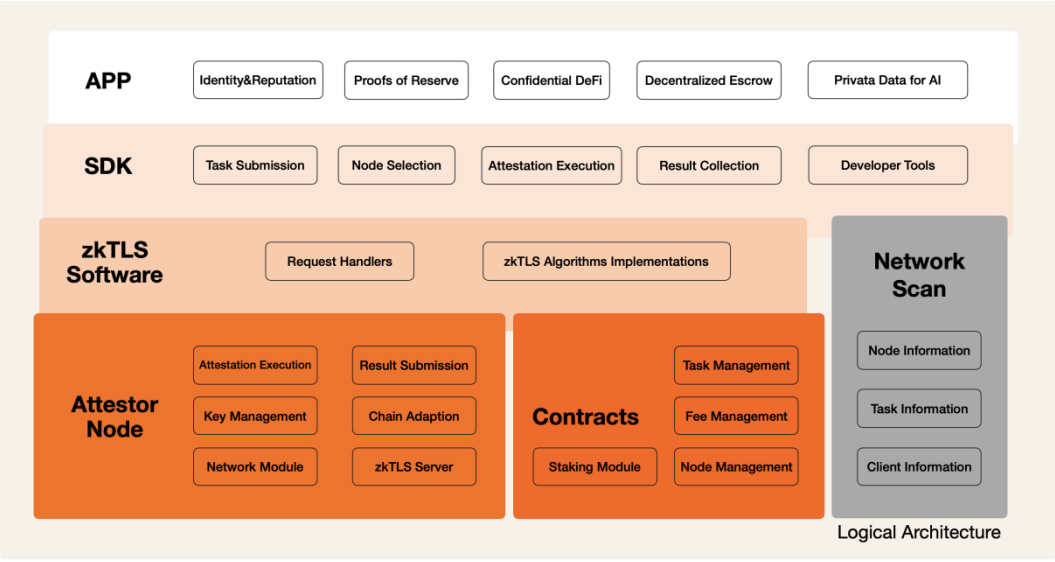

Figure 3: High-level Primus network architecture

At a high level, as in Figure 3, the Primus system comprises on-chain system contracts (registry, staking, and settlement), the Primus SDK used by dApps and clients, attestor workers, and optional verifiers/auditors. A typical session proceeds as follows, as in Figure 4:

1. The client creates a task and posts the required fee or payment commitment to the settlement contract.
2. The registry selects candidate attestors uniformly at random from the set of registered attestor nodes.
3. The client and attestor perform the zkTLS protocol (MPC or Proxy mode), produces an attestation, and the attestor returns the result to the client.
4. The attestation is validated by the on-chain verifier, upon successful verification the settlement contract releases payment to the attestor and updates any relevant state (for example reputational scores or logs).

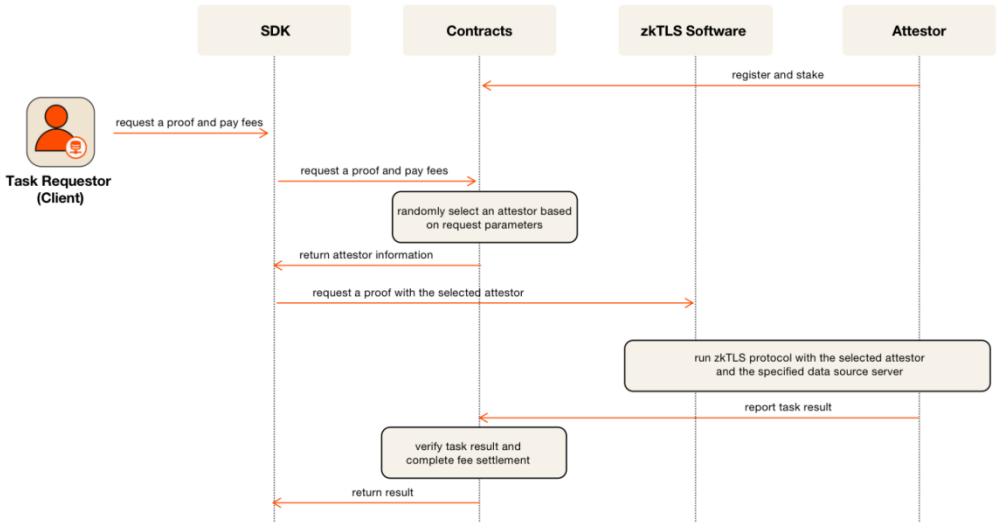

Figure 4: Core task flow for zkTLS sessions

3.1.3 Network economics

Clients may pay task fees using PRIM or other supported tokens for verification services provided by the network. The fee shall be paid to the system contracts prior to the task execution. The fees from the executed task will be shared by the attestor which actually runs the zkTLS session with the client, and the protocol. Attestor shall stake a certain number of tokens and waits for a while before it can onboard as an official node. Attestor will have to wait for a certain period to exit the staking before returning the locked funds.

3.1.4 Penalties and dispute resolution

Misbehavior by an attestor, for example signing a zkTLS session without correctly executing the protocol, is subject to penalties. Protocol correctness is established by TEE attestations and any supporting on-chain or off-chain evidence submitted during dispute resolution.

If a node's instability causes execution failures or repeated timeouts, the incident must be reported to the system contracts. A predefined tolerance window and failure threshold determine whether penalties apply: occasional failures within the tolerated rate incur no penalty, while exceeding the configured timeout or failure threshold triggers sanctions (for example warnings, temporary suspension, or slashing of stake).

The penalty mechanism is deployed gradually. During initial staging there is no automated slashing and only authorized nodes may participate; punitive measures are introduced after the network demonstrates operational stability. When penalties are enabled, task execution and disputes are verified using TEE

attestations and the agreed adjudication workflow so that correctness and liability can be reliably determined.

3.1.5 Governance

The governance mechanism will primarily be responsible for network upgrades, parameter adjustments, and other key decisions affecting the system. In the early stages, governance will not be activated, allowing the network to focus on stability and adoption. Once introduced, governance will cover parameters such as the minimum staking requirement for attestor nodes, the unbonding period for attestor nodes, the minimum staking requirement for clients, the unbonding period for regular stakers, and the allowable range for the proportion of protocol-based incentives distributed by participants to their delegated attestor nodes.

3.2 The Data Computation Layer

The Data Computation Layer extends the Data Verification Layer to enable richer privacy- preserving computations. It allows applications to perform complex operations over sensitive inputs without revealing those inputs, while producing outputs and succinct proofs that can be efficiently verified.

3.2.1 The DVC Mode with zkVM

The DVC (Data Verification and Computation) mode uses zkTLS to prove the encrypted TLS transcript, yielding verifiable correctness and authenticity of HTTPS requests and responses, and the processing of the attested private data is outsourced to external zkVMs. In this mode, Primus zkTLS can be integrated with any zkVM or zkDSL that supports the required cryptographic primitives, enabling a wide range of applications to leverage the privacy guarantees of zkTLS while performing complex computations on the attested data. A few components are involved in the DVC mode, as described below:

- **Dapp:** The end application built by developers using the Primus SDK; it submits verification tasks, receives attestations and proofs, and consumes verified outputs.
- **Primus Network SDK:** A client library and toolset that helps developers construct zkTLS tasks, validate attestations locally, and extract plaintext outputs for use in application logic.
- **Primus Data Verification Network:** The decentralized verification layer (registry, staking, settlement contracts, and attestor nodes) that runs zkTLS tasks, issues attestations, and enforces economic incentives and penalties.
- **Verification Programs:** Trusted programs executed inside a TEE/zkVM that perform attestation verification and domain-specific computation:
  - **Primus zkTLS Verification Program:** A canonical Rust component that, inside the enclave, verifies attestation signatures, checks source URLs and transcripts, and extracts authenticated plaintext payloads for downstream processing.
  - **Business Program:** Developer provided code (Rust or other supported runtimes) that runs inside the TEE/zkVM, consumes verified plaintext, enforces application-specific checks, and emits zkVM proofs or attestations of the performed computation.

DVC Workflow The DVC workflow can be viewed as two tightly coupled phases: (A) a zkTLS session that yields a sealed attestation and authenticated transcript, and (B) a proving phase inside a TEE/zkVM that consumes the attestation and emits a succinct proof of the requested computation. The diagram in Figure 5 shows the actors and message flows; below we list the detailed steps.

Phase A: zkTLS session and attestation

1. **Task submission.** The Dapp constructs a task descriptor (source URL(s), request parameters, verification predicates, expected output fields, and payment parameters) and submits it to the Primus settlement contract via the Primus SDK. The settlement contract records payment commitments and returns a task identifier and a randomly assigned attestor drawn from the registry.

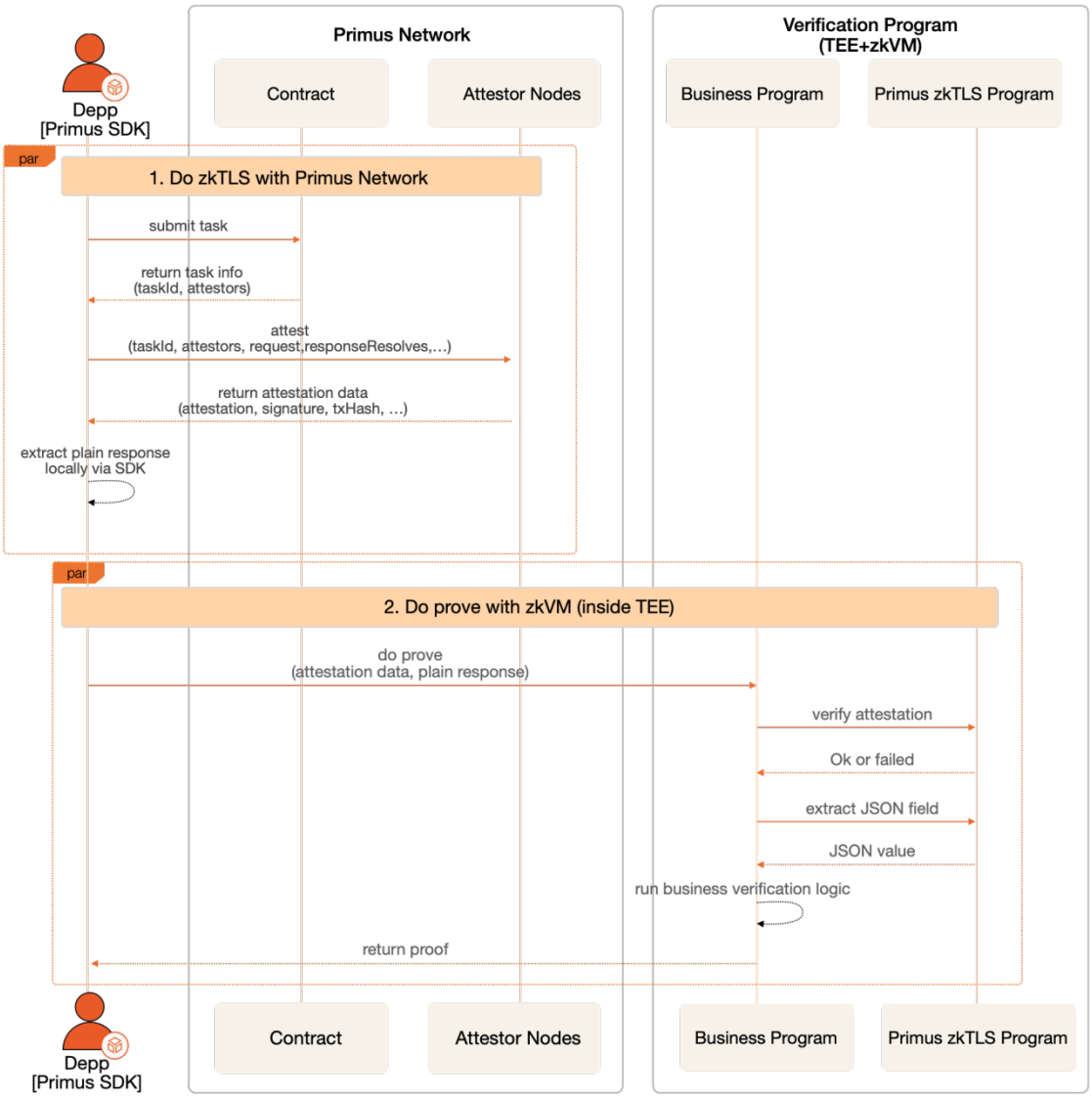

Figure 5: DVC workflow

2. **zkTLS protocol.** The selected attestor and the client perform the zkTLS protocol in a chosen mode (MPC or Proxy). The attestor runs the zkTLS session inside the TEE, ensuring that all code executes as expected. The client receives an attestation from the attestor that includes a signature over the sealed metadata; this attestation is used in subsequent steps.
3. **Private data extraction (developer-local).** If the attestation proves correct execution, the SDK enables the Dapp to extract authenticated plaintext fields necessary for application logic. Extraction is performed without exposing other secrets back to the network, the plaintext is decrypted locally.

Phase B: zkVM proving and verification

4. **Forwarding to the verification programs.** The Dapp forwards the minimal attestation package and the extracted plaintext to the Verification Program running inside a TEE/zkVM. This message is typically delivered over an authenticated channel and may include the task ID and any parameters for the business logic. In zkVM, the programs verify the following steps, and generate a succinct proof of correct execution.
5. **Attestation verification.** It first invokes the Primus zkTLS Verification Program to recheck the attestation: verify the enclave signature, validate the enclave measurement (ensuring the canonical verifier binary is running), confirm the transcript hash and the

source URL, and enforce replay protection (nonces, timestamps, or on-chain task bindings).

6. **Domain-specific processing.** Upon successful attestation verification, the Business Program extracts the required JSON fields or other inputs, performs application-specific computation defined in the program.
7. **Return proof.** It returns the zkVM proof to the Dapp. The Dapp can submit the proof to an on-chain verifier or a settlement contract.

3.2.2 Confidential Computation with FHE

To further enhance privacy guarantees, Primus integrates FHE into the Data Computation Layer. This allows computations to be performed directly on encrypted data, ensuring that sensitive information remains confidential throughout the processing lifecycle.

Blockchains excel at transparency, security, and trustlessness, but that transparency entails trade-offs. On-chain activity such as transactions, contract calls, and account state is publicly visible, which makes blockchains unsuitable for many sensitive use cases such as finance or healthcare. Fully homomorphic encryption (FHE) mitigates this transparency problem by enabling computation over ciphertexts: the chain interacts only with encrypted data, not the underlying plaintext, while authorized parties can still recover correct outcomes after decryption. In short, smart contracts can operate as usual while preserving privacy, without weakening decentralization or security.

Architecture. Primus introduces FHETransform, a modular, layered architecture for privacy-preserving computation on blockchain networks. FHETransform combines on-chain smart contracts with off-chain processing to deliver secure and efficient fully homomorphic encryption workflows.

Key components. FHETransform primarily consists of the following components:

- **FHE Solidity library:** (a.k.a. FHE system contracts) enables developers to author confidential contracts in standard Solidity using encrypted data types and operations.

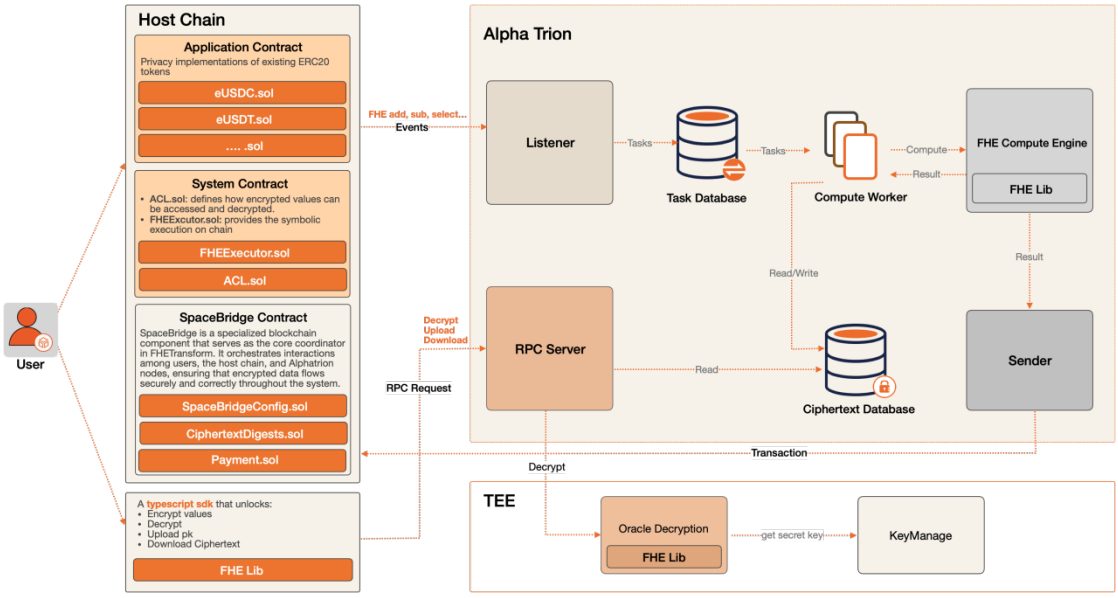

Figure 6: FHETransform architecture

The FHE library performs symbolic, on-chain computation for encrypted-data operations while delegating the heavy FHE computation to the off-chain service layer (AlphaTrion). It exposes a compact API surface for contracts to reference encrypted values and operation descriptors without revealing plaintext.

The FHE library primarily comprises two contracts:

- FHEExecutorContract: handles encrypted-data processing (data types and operator encodings).

– ACLContract: manages access control and determines whether a given address is authorized to access or decrypt a ciphertext handle.

The library avoids executing expensive FHE routines on-chain; instead, it records symbolic operation descriptors (for example operation identifiers and argument hashes) which AlphaTrion consumes to perform the actual encrypted computations off-chain.

- **AlphaTrion:** AlphaTrion is an off-chain compute component that executes FHE workloads. It is implemented on top of a TFHE library with CPU and GPU acceleration and monitors the host chain for FHE-related events. Events are queued in a task database and consumed by one or more computational workers; upon completion, computed ciphertexts and result metadata are written back to the database. We distinguish two execution modes:

  Symbolic execution (on-chain) When a smart contract invokes an FHE operation, the on-chain FHE contract does not perform the heavy FHE computation. Instead it performs a symbolic execution that:

  – accepts 'bytes32' handle values as inputs;

  – produces a new output handle (computed as a hash over input handles, operation type, result type and parameters);

  – emits an on-chain event containing the relevant handles to notify AlphaTrion to perform the actual computation off-chain.

  Actual execution (off-chain) When AlphaTrion workers pick up a queued task they:

  – retrieve ciphertexts from the database using the input handles;

  – perform the actual FHE computation using the TFHE library;

  – compress and store the resulting ciphertext under the output handle in the database;

  – upload a cryptographic commitment (digest) of the ciphertext to the chain for verification.

  Each AlphaTrion computation or input verification is accompanied by a cryptographic commitment and a digital signature so third parties can independently verify result correctness.

- **SpaceBridge:** A blockchain-side coordinator that mediates interactions among users, the host chain, and AlphaTrion nodes. SpaceBridge manages the lifecycle of encrypted data operations, including validation, consensus enforcement, and payment settlement.

  SpaceBridge implements a small set of on-chain contracts that provide configuration, ciphertext bookkeeping, and payment logic. The primary contracts are:

  – SpaceBridgeConfig: the single source of truth for SpaceBridge configuration. It manages the registry of AlphaTrion nodes (transaction sender and signer addresses, consensus thresholds) and enables authenticated updates to the network topology and validation rules.

  – CiphertextDigests: stores and manages ciphertext commitments on EVM-compatible chains. It validates encrypted inputs and records digests submitted by AlphaTrion nodes; when multiple nodes submit matching digests, the contract confirms consensus for the corresponding result.

  – PaymentContract: governs pricing and fee collection for FHE services (for example public decryption, user-specific decryption, and callback execution). It enforces payment settlement and distributes rewards to AlphaTrion nodes according to protocol rules.

  Together these contracts allow SpaceBridge to coordinate trust-minimized, privacy-preserving computations across a decentralized set of coprocessors while preserving on-chain auditability and economic incentives.

- **Preprocessor:** An off-chain component that converts conventional EVM smart con- tracts into FHE-enabled contracts, enabling confidential transactions with minimal developer workflow changes.

- **Key Management Service (KMS):** A threshold MPC-based network responsible for generating, rotating, and managing FHE keys, and for performing secure, verifiable decryption when authorized.

4 Use Cases

The following scenarios illustrate how Primus enables agents to operate with provable behavior and privacy-preserving data access across key verticals of the agent economy.

4.1 Agent-managed Under-collateral Lending

Agents can unlock more efficient credit markets by combining private off-chain attestations with on-chain behavior signals. A borrower delegates to a lending agent that collects authenticated responses from payroll providers, banking APIs, and repayment services via zkTLS and produces compact attestations revealing only the policy-relevant facts, such as an income band, a reserve floor, or a repayment score. On the other side, lender agents fuse these attestations with on-chain activity history and real-time market pricing to compute credit decisions, set dynamic collateral ratios, and offer risk-adjusted interest schedules.

This flow lets borrowers participate in credit markets without sharing raw financial records, while lenders gain higher-fidelity, auditable signals. The result is faster underwriting cycles, deeper capital efficiency, and a safer delegation model in which users retain granular control over which attestations agents may cache, reuse, or revoke.

4.2 Agent-enabled Proofs of Reserve

Agents that manage treasury or custody functions can produce authenticated, privacy- preserving proofs of reserve without disclosing account-level details. A reserve agent ingests authenticated balance and liability data from custodians and counterparties over zkTLS, aggregates it within a confidential execution boundary, and issues a compact attestation of coverage metrics that any auditor or market participant can independently verify.

This architecture supports higher-frequency attestations and near-real-time reserve monitoring while keeping proprietary positions and counterparty relationships confidential. Market participants gain assurance from verifiable proofs, issuers protect trading strategy, and auditors receive portable, revocation-aware attestations that integrate directly into automated compliance pipelines.

4.3 Privacy-preserving Identity and Delegation

Agents frequently act on behalf of users in identity-sensitive contexts. An identity agent combines zkTLS-based API capture with selective-predicate proofs to extract only the policy-relevant outcomes (for example, age above a threshold, jurisdiction clearance, or a KYC verification pass) and packages them into attestations that carry freshness, scope, and optional revocation metadata. Relying parties can verify these attestations without accessing raw documents or underlying personal attributes.

This model eliminates repeated identity friction, allows users to grant narrow, time- bound consent to their agents, and supports richer agent services, including personalized onboarding, automated entitlement checks, and compliance gating, all without broad data exposure. Regulators and auditors can request scoped attestations on demand and hold agents accountable to their declared verification policies.

4.4 Agent-orchestrated On/Off Ramping

Agents can orchestrate multi-rail fiat-to-crypto and crypto-to-fiat flows while preserving payer privacy and producing verifiable settlement proofs. In a typical flow, a buyer agent initiates payment on a selected rail; a payment-attestor agent monitors settlement confirmations via authenticated APIs and emits a zkTLS-backed attestation containing only the required fields; finally, a verification agent or zkVM validates the attestation and instructs the on-chain escrow to release tokens.

This separation of concerns yields non-custodial, auditable settlement that scales across payment rails and jurisdictions. Users delegate routing, reconciliation, and compliance steps to specialized agents while retaining provable settlement guarantees and minimal disclosure of payment details.

4.5 Confidential DeFi Workflows for Institutional Agents

Institutional agents can participate in DeFi while keeping strategy and client data confidential through a combination of FHE, confidential execution layers, and verifiable attestations. Agents submit encrypted order books, position vectors, and policy constraints; the execution layer performs netting, pricing, risk calculation, and compliance checks directly on ciphertexts and returns encrypted settlement outputs together with attestations that the declared policies were followed.

Concrete applications include private matching and dark-pool-style execution, confidential credit assessment for bilateral lending, and encrypted AMM provisioning that conceals individual order flow. These workflows mitigate front-running and information leakage, protect trading alpha, and let institutions safely delegate execution and liquidity operations to agents under enforceable privacy and audit guarantees.

4.6 Social and Identity-based Payments via Agents

Agents simplify payments to social identifiers by abstracting identity verification and claim settlement on behalf of both senders and recipients. A sender agent locks funds against an identity commitment or a time-locked claim; a recipient agent proves control of the corresponding social identifier or credential through a zkTLS-backed attestation and redeems the commitment to a verified wallet. When higher assurance is needed, agents coordinate challenge periods and multi-attestor checks.

This pattern lowers onboarding friction for Web2 users, reduces errors in identity-to-wallet mapping, and enables large-scale, identity-mediated disbursements for creators, social platforms, and cross-border remittances, all while keeping personally identifying data off-chain and private.

Summary. Across these scenarios, Primus equips agents with verifiable behavior and privacy-preserving data access. This combination forms the foundation for an agent economy in which users delegate with confidence, platforms scale agent permissions safely, and markets achieve deeper liquidity and broader services.

5 Economic Model

This section summarizes the native economic model for the Primus Network and the design rationale for the PRIM token, the network's utility, distribution and value-capture mechanisms.

5.1 What is PRIM

PRIM is the native utility and governance token of the Primus Network, a decentralized verifiable confidential-computing infrastructure that combines authenticated Web2 data attestation with privacy-preserving computation.

PRIM serves as a core coordination mechanism within the network, facilitating payments for verifiable computation, supporting attestation nodes, enabling governance participation, and aligning activity across operators, developers, and enterprise clients.

PRIM may be used in protocol-level staking mechanisms to support node participation, task execution eligibility, and network reliability. Participants may receive protocol-defined incentives based on their contribution to network operations and correct execution of tasks.

Important Notice: PRIM is designed as a utility token for use within the Primus Network. It is not intended to be a security, financial instrument, or investment product, and does not provide any rights to ownership, profits, or dividends. The functionality of the token may evolve over time and is subject to regulatory considerations in different jurisdictions.

5.2 Token Supply and Allocation

- **Total Supply:** 1,000,000,000 PRIM
- **Token Standard:** BEP-20 (BNB Chain) and ERC-20 (Base)
- Token issuance and circulating supply are determined by the vesting schedule described below.

Initial Token Allocation:

- **Ecosystem (33%):** Supports network-level incentives and long-term ecosystem development, including infrastructure incentives, node operator participation, research and development, application integration, strategic partnerships, and protocol expansion.
- **Community (18%):** Allocated to support community growth and developer adoption, including developer programs, community engagement, user onboarding, and ecosystem participation initiatives.
- **Early Investors (16.81%):** Allocated to strategic and institutional partners, including angel, seed, and Series A investors, supporting early-stage development and network formation.
- **Core Contributors (18%):** Reserved for core contributors, including engineers, researchers, and operational team members responsible for protocol development and maintenance.
- **Airdrop & Community Distribution (4.49%):** Allocated for TGE airdrops and community distribution initiatives, including hackathon contributors, early users, and Points & Passport participants, to support initial adoption and ecosystem growth.
- **Liquidity (3%):** Allocated for liquidity provisioning to support market stability and efficient token trading.
- **Advisors (6.7%):** Allocated to advisors contributing to cryptography, research, strategic development, and global market expansion.

5.3 Unlock and Vesting

- **Ecosystem:** 4% of the total token supply unlocked at genesis; remaining tokens vest linearly over 48 months following TGE.
- **Community:** 2% of the total token supply unlocked at TGE; remaining tokens vest linearly over 48 months following TGE.
- **Early Investors:** 12-month cliff from allocation, followed by linear monthly vesting through month 36.
- **Core Contributors:** 12-month cliff from allocation, followed by linear monthly vesting through month 36.
- **Airdrop and Community Distribution:** 4% of the total token supply unlocked at TGE; remaining tokens vest linearly over 12 months following TGE.
- **Liquidity:** 100% unlocked at TGE.
- **Advisors:** 12-month cliff from allocation, followed by linear monthly vesting through month 36.

At TGE, only a portion of the total token supply will be in circulation, with the remainder subject to the vesting schedule described above.

5.4 PRIM Value Capture and Utility

PRIM is used to coordinate network activity and enable access to services within the Primus Network.

- **Computation Payments:** PRIM may be used to pay for verification and computation tasks, facilitating coordination between clients and network nodes.

- **API Access:** PRIM may be used by developers to access network services and infrastructure.
- **Staking Mechanisms:** PRIM may be used in protocol-level mechanisms to support node participation, service reliability, and network security.
- **Governance Participation:** PRIM holders may participate in governance processes including proposals, voting, and protocol parameter adjustments.
- **Protocol Incentives and Assurance:** PRIM may be used to support network participation and, as part of protocol mechanisms, discourage inactive or malicious behavior.

5.5 Economic Roles and Participants

The Primus economy organizes around four primary actors:

- **Nodes:** Participants who provide computational resources and infrastructure to the network. A minimum stake of PRIM may be required for node participation. Nodes perform verification and computation tasks, including proof generation and confidential computation.
- **Developers:** Developers access network APIs and SDKs to build applications utilizing Primus's verification and privacy-preserving computation capabilities.
- **Clients:** Clients submit tasks requiring verifiable proofs and privacy-preserving computation.
- **Token Holders:** Token holders may participate in governance and other network functions in accordance with protocol rules.

5.6 Sustainability and Growth

The distribution and vesting schedule aims to bootstrap network security and developer adoption while tying longer-term allocations to network activity. Ecosystem funds are released in step with demonstrated demand for confidential computation, reducing inflationary pressure while providing resources for growth.

6 Roadmap

Primus is building the trust and privacy layer for the agent economy, combining zkTLS for verifiable access to external data with FHE for confidential computation. The roadmap below tracks the network's progress from foundational infrastructure to ecosystem expansion and long-term governance for AI agent-driven applications.

Past Milestones

- **Q4 2024 – Core Primus network completed:** Decentralized verification and computation infrastructure with zkTLS and FHE was completed, providing trusted identity and privacy-preserving capabilities for AI agents.
- **Q2 2025 – Initial MVPs launched:** zkCredential, programmable social payments, verification-rule red envelopes, Proof of Reserve (PoR), and confidential transactions were launched, enabling autonomous verification and execution by AI agents.
- **Q4 2025 – Developer tools deployed:** APIs, SDKs, and DevHub support ecosystem integration and facilitate third-party applications interacting with the agent network.

Future Milestones

- **Q2 2026 – Network deployment and optimization:** Full deployment of the decentralized verification and computation network; optimization of zkTLS and pre- liminary FHEVM applications for privacy-preserving smart contracts and compliant, confidential on-chain transfers, enabling verifiable execution for AI agents.

- **Q3 2026 – Node onboarding and early application adoption:** Onboard and incentivize node operators using PRIM staking and reward mechanisms; support early adoption of AI and fintech applications, enabling agents to autonomously execute actions and payments.
- **Q4 2026 – First ecosystem application wave:** Launch privacy-preserving AI data training, on-chain verifiable voting, and confidential payments, demonstrating trusted, privacy-preserving agent execution in real-world scenarios.
- **H1 2027 – Community and developer expansion:** Expand community engagement through hackathons, developer incentives, and ecosystem grants for third-party application integration, fostering agent ecosystem growth.
- **H2 2027 and beyond – Sustainable ecosystem and governance:** Establish a privacy data marketplace, expand enterprise adoption across Web3 and fintech, strengthen on-chain governance, and maintain sustainable token incentives for developers, nodes, and users to support long-term AI agent economy growth.

Summary

Primus has already established its foundational infrastructure, early products, and developer tooling. The next phase focuses on scaling the network, broadening application adoption, and building the governance and incentive systems required for a durable agent economy.

References

[1] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, August 2008.

[2] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, August 2018.

[3] Fan Zhang, Zeyu Liu, Zhi-Jian an, Sanket Kanjalkar, and Ari Juels. DECO: liberating web data using decentralized oracles for TLS. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, Virtual Event, USA, November 9-13, 2020, pages 1447—1466, 2020.

[4] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. Lightweight authentication of web data via garble-then-prove. Cryptology ePrint Archive, Paper 2023/964, 2023. https: //eprint.iacr.org/2023/964.

[5] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Paper 2021/076, 2021.

[6] Xiang Xie and Xiao Wang. Unearthing the reality of zktls: A benchmarking and crypt- analysis report. HackMD, 2023. URL: https://hackmd.io/X-NSDlUtQoC66aPOyGo5NA.

[7] Primus. https://primuslabs.xyz/.

[8] TLSNotary. https://tlsnotary.org/.

[9] Reclaim. https://www.reclaimprotocol.org/.

[10] Pluto. https://pluto.xyz/.

[11] Primus. https://github.com/primus-labs/zktls-bench/tree/main.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Leveled fully homomorphic encryption without bootstrapping. In Proceedings of ITCS, 2012. Foundational lattice-based leveled FHE constructions (BGV).

[13] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive, 2012/144, 2012. BFV-style scheme optimized for integer arithmetic.

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In ASIACRYPT 2017, 2017. Introduces CKKS for efficient approximate arithmetic.

[15] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors and the gsw framework. IACR Cryptology ePrint Archive, 2013. Matrix/GSW-style ciphertexts enabling gate evaluation and bootstrapping.

[16] L´eo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, Advances in Cryptology - EUROCRYPT 2015, Lecture Notes in Computer Science, pages 617—640. Springer, 2015.

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Tfhe: Fast fully homomorphic encryption over the torus. IACR Cryptology ePrint Archive, 2016/421, 2016. Practical gate-bootstrapping FHE library and algorithmic improvements.

[18] Fengrun Liu, Haofei Liang, Tianyu Zhang, Yuncong Hu, Xiang Xie, Haisheng Tan, and Yu Yu. HasteBoots: Proving FHE bootstrapping in seconds. Cryptology ePrint Archive, Paper 2025/261, 2025.

[19] Louis Tremblay Thibault and Michael Walter. Towards verifiable fhe in practice: Proving correct execution of tfhe's bootstrapping using plonky2. ePrint, 2024.

[20] The RISC Zero Team. Universal zero knowledge. https://risczero.com/.

[21] The SuccinctLabs. The fastest, most feature-complete zkvm for developers. https: //github.com/succinctlabs/sp1