ANKR / WEB3 TECHNOLOGIES, INC.

Verifiable RPC

Reading a Blockchain Without Trusting the Provider

Draft ▪ July 2026 ▪ v0.9

The operator becomes untrusted infrastructure.
You verify the math.

github.com/w3tech/verifiable-rpc-sdk ▪ verifiable-rpc-sidecar

When an app reads a blockchain through a hosted RPC provider, it takes the provider's word that the data is real. vRPC removes that trust by running the node inside a sealed hardware enclave - an Intel TDX confidential VM the operator, Ankr included, cannot read or alter. Every response is signed inside that enclave, and the client verifies the signature locally before application code sees the data; anything tampered with, stale, or for the wrong chain is rejected. The node is the chain's own official image, not an Ankr build, so you verify it exactly as you would if you ran it yourself. The operator becomes untrusted infrastructure - you verify the math.

Abstract

An app that reads a blockchain through a hosted RPC provider is trusting that provider to tell the truth: that it runs the node software it claims, and that the bytes it returns are real chain data, and an ordinary RPC call proves neither. A compromised server or a tampered process can return a wrong balance or a forged log with nothing for the caller to catch it by. vRPC removes that trust, ours included: each response is signed inside an Intel TDX enclave the operator cannot read or modify, and the client verifies the signature locally before the data reaches application code. The node is the chain's own official image, run unmodified and pinned by digest, so the caller is not trusting an Ankr build of the node either. A response a client accepts is the exact output of that attested software, for the chain requested, produced within a bounded freshness window. The cryptography is standard and independently checkable. The one trust a client cannot remove by math is Intel's hardware attestation, and much of this paper is about how far we shrink even that.

Contents

1. The trust you didn't ask for

Run your own full node and you trust no one for chain state. Most apps can't, so they read through a hosted provider and inherit a trust they never signed up for. Three things are assumed honest at once: the provider, every network hop between it and you, and the node software the provider says it runs.

None of that is checked anywhere in the path: nothing ties the returned bytes to a particular node, a particular chain, or a particular moment, so someone who controls a proxy hop, or who has broken into the machine running the node, can hand back a response that parses fine and is simply false - a stale block, a forged log, a balance that isn't real - and the app acts on it.

An attacker with host access doesn't need to be subtle: they can run their own build of the node in place of the real one. The stealthiest version is the in-memory compromise - the node's behavior is changed in RAM while the binary on disk is left untouched, so file checksums pass, reputation stays intact, and the lies go out with nothing looking wrong. Cloud setups that can't bind a running workload to specific hardware are wide open to all of it. This isn't unique to hosted providers - a compromised host defeats a node you run yourself just as thoroughly. Binding the workload to attested hardware is the only thing that closes it.

Bitcoin removed the trusted third party from payments [1]. vRPC removes the trusted operator from reads. The goal is narrow and worth stating flatly: the provider, us included, should be something you verify, not something you trust. This is not about whether the chain itself is correct. It is about whether the bytes you got are the real, unaltered output of the node you think you are talking to.

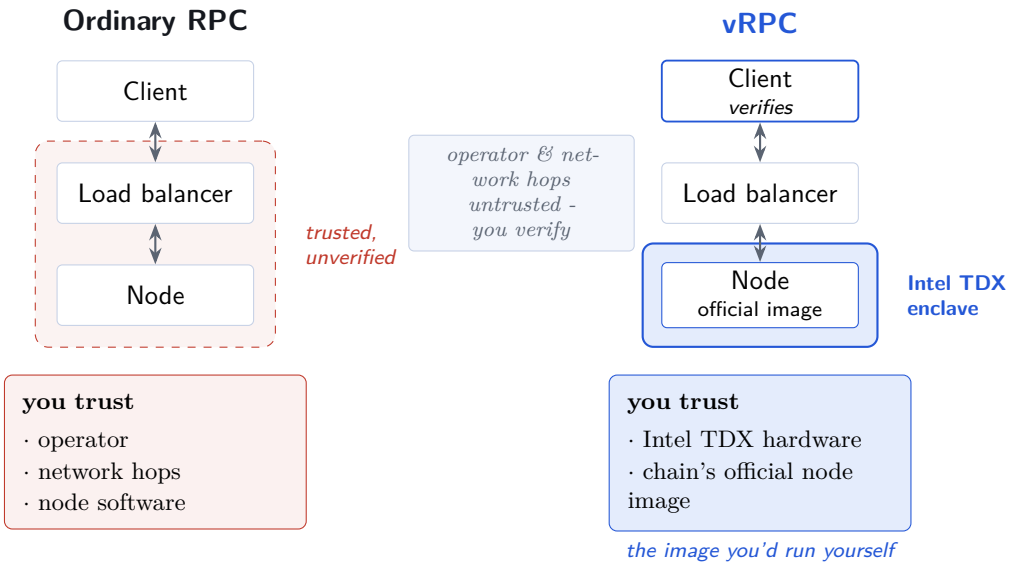

Figure 1. *Trust doesn't shrink to nothing, it moves. The request path is the same in both - client, load balancer, node - but ordinary RPC trusts the operator, the network hops, and the node software on faith, while vRPC verifies each response at the client and trusts only Intel's attestation and the chain's own official image, leaving the operator and the hops untrusted.*

Set against the alternatives, reading a chain is a trust-versus-cost trade, and vRPC changes what sits in the first column:

Approach	What you trust	Overhead	What it proves
Plain hosted RPC	The operator, every hop, the node software - all on faith	None	Nothing
Run your own node	No one for chain state	Full sync, hardware, ops	Everything, if you can afford it
Light client	The chain's consensus rules	Header sync, plus a proof round-trip per query; limited method coverage	A value is on-chain
vRPC	Intel's attestation and a published measurement	One cached attestation per node; a microsecond check per call	The response is the untampered, recent output of the chain's own official node

2. The idea

The mechanism is small enough to state in one sentence: run the node inside a hardware enclave the operator cannot see into or change, sign every response inside that enclave, ship the signature with the response, and have the client check it locally and refuse anything that fails.

Two things make the shift real. First, the signing key lives only inside the enclave, released to it by an attested key service and never exposed to the host or the operator, so a signature that verifies could only have come from the approved software running in genuine hardware. Second, the node is not operator software: it is the chain project's own official image, run unmodified and pinned by digest, so you verify the node the way you would if you ran it yourself. What is left to trust the operator on is close to nothing, and the rest of this paper accounts for exactly what.

Where this matters most. The trust you recover scales with what a wrong answer costs. A bridge deciding whether to release funds, or a protocol reading a price before it moves money, is the case for it - there, a forged log or a stale balance is the gap between safe and drained. For a low-stakes read, plain RPC is fine. vRPC is for the reads you cannot afford to be lied to about.

3. What you have to trust

Most papers bury this, but for vRPC it is the point, so it goes up front. Here is the whole of what a vRPC client trusts, and what it explicitly does not.

The adversary. Assume the attacker controls the host OS, the hypervisor, the network, and the operator's own infrastructure, Ankr included. They can read and write host memory and disk outside the enclave, drop or alter or replay anything on the wire, and restart or redeploy the node at will. What they cannot do is break standard cryptography or violate the TDX hardware guarantees. Attacks on the CPU itself (side channels, speculative execution, fault injection, physical probing) are out of scope, as they are for TDX in general [4].

What is trusted (the TCB). A client that accepts a vRPC response is trusting, and only trusting:

- Intel's hardware: the CPU, the TDX module [2], and the DCAP attestation chain that Intel signs [3].
- The approved node and sidecar software, at a version whose measurement is published.
- dstack-os and dstack-kms, the confidential-VM base and the key service that derives and hands the signing key to the enclave. KMS is in the trust path because it holds the key material [5].

That is the entire list, and the operator is not on it.

The assumptions. Two kinds, and keeping them apart is the honest version of the guarantee.

Assumptions

Cryptographic assumptions, the standard hardness kind:

- **A1** Ed25519 is unforgeable under chosen-message attack [8].
- **A2** SHA-256 is collision- and preimage-resistant [9].

Trust assumptions, which rest on hardware and process rather than a hardness bound:

- **A3 (TDX soundness)** A genuine trust domain's memory is sealed from the host and hypervisor, and a DCAP quote cannot be forged for a measurement and REPORTDATA the hardware did not actually produce. This is trust in Intel's silicon and PKI, not a reduction, and it is the root we work hardest to shrink. dstack-kms is itself such a trust domain, so key derivation happens inside the attested boundary [5].
- **A4 (measurement to source)** The published measurement honestly names the software it stands for. Where a build is reproducible, anyone can rebuild it and confirm the mapping. Where it is not, the mapping rests on the publisher (§6).

The next section states what follows, and §6 is specific about how far the trust assumptions reach.

4. How it works

Four moving parts: the enclave and its key, the per-response signature, byte-exact transport, and the client check. Attestation ties them to hardware.

The enclave and its key. A vRPC node runs as an Intel TDX confidential VM. Two processes run inside it: the blockchain node (say, Arbitrum Nitro) and the vRPC sidecar, which fronts the node and signs what it returns. The sidecar's Ed25519 signing key is not minted at boot. It is derived by dstack-kms, a separate attested confidential VM, and handed to the sidecar over a channel KMS opens only after checking the sidecar's TDX quote and measurement (Phala's docs call this RA-TLS) [6]. The private key never leaves the enclave. Its public half is written into the sidecar's own quote, so anyone can confirm the key signing their responses belongs to a genuine enclave running the approved measurement.

One consequence shapes what the guarantee means. KMS derives the key deterministically from the workload's identity - the hash of its code and configuration - and releases it only to an enclave whose TDX quote it has checked against that identity. Any genuine enclave running the approved measurement is handed the same key, so a fleet can restart and rebalance without

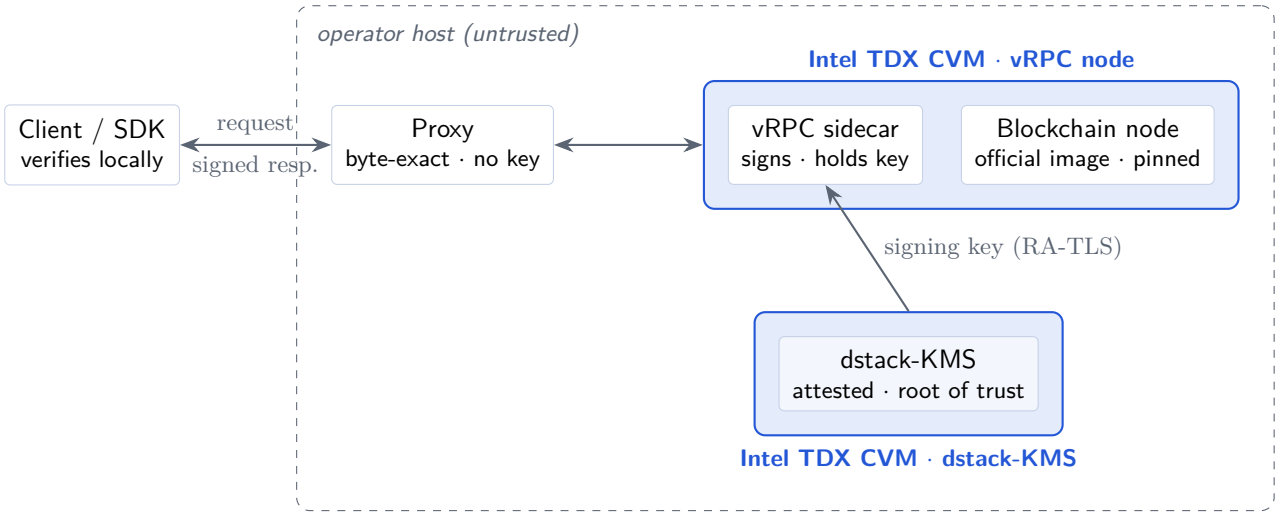

Figure 2. *The vRPC node and the key service each run as their own Intel TDX confidential VM on the operator's untrusted host. The blockchain node and vRPC sidecar share one CVM; dstack-KMS is a separate attested CVM that releases the signing key to the sidecar over RA-TLS, only after checking its quote. The proxy passes bytes through unchanged and holds no key.*

re-establishing trust, and the signature proves you are talking to the approved workload, not one specific box. §5 (G2) covers what that costs.

The signature. For each response the sidecar signs an 80-byte pre-image:

```
chain_id (8 bytes, little-endian)
  ‖ sha256(request_body)
  ‖ sha256(response_body)
  ‖ timestamp_ms (8 bytes, little-endian)
```

The signature and its fields ride back as HTTP headers (`vRPC-Signature`, `vRPC-Pubkey`, `vRPC-Timestamp`). Binding the request hash, the response hash, the chain, and the time into one signed object is what lets a single check catch tampering, replay, and cross-chain mix-ups at once.

Byte-exact transport. Because the client hashes the exact bytes it sent and got back, the proxy in front of the node has to pass them through unchanged. vRPC nodes serve on a dedicated passthrough path: no JSON-RPC re-marshaling, no response cache, and no splitting a batch across nodes. Anything that rewrote a byte would break the client's hash, so none of it runs here. When several nodes serve one chain behind a load balancer, a node identifier (NodeId) travels with each response so the client can attest the exact node that answered; it sits outside the signature and carries no trust.

The client check. The client rebuilds the pre-image from what it sent and received and verifies the signature under `vRPC-Pubkey`. The data is already in hand, so this adds no round trip. If anything is off (bad signature, stale timestamp, a missing header) the SDK throws a typed error and returns nothing. No code path hands back an unverified response.

Attestation. A signature is worth only as much as the proof that its key lives in real hardware running approved code. That proof is fetched once per node and cached (default one hour), off the hot path:

Step	Check	What it establishes
1	Client sends a fresh 32-byte nonce to the node's attestation endpoint; gets back the TDX quote, the signing pubkey, and the app-compose that pins every image by digest	A hardware-signed statement of what is running
2	The quote chains to Intel's root	Genuine TDX, not an emulator
3	The app-compose hash matches the approved value, and every image it pins can be pulled and inspected	The approved config and images are what run
4	REPORTDATA holds `pubkey` ‖ `nonce`	This quote was made for this request, and this key belongs to this enclave

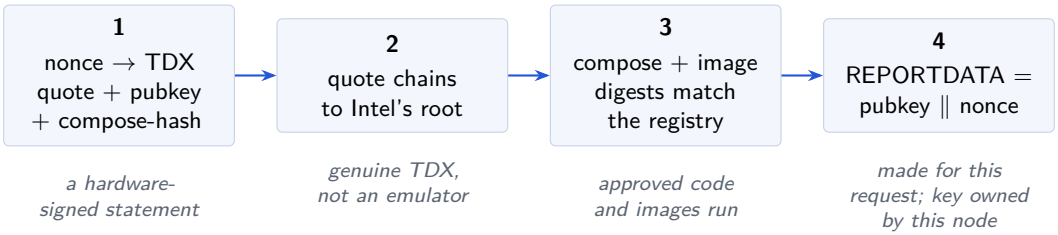

Figure 3. *Attestation, fetched once per node and cached (default one hour), off the hot path. A fresh nonce ties a hardware-signed quote to this node and this signing key.*

The pubkey in the attestation must equal the `vRPC-Pubkey` on the responses, so the node identifier needs no trust and the proxy keeps no key registry. The compose hash is recorded in the TDX runtime register RTMR3, next to the app and instance identity [7]; the client reproduces it by replaying the event log and matches it against the approved value in a public registry.

Integration. There are two ways in. The SDK is a one-line swap that leaves every existing call working, now verified: with ethers v6, `new VrpcProvider(url, chainId)` replaces `new ethers.JsonRpcProvider(url)`; with viem, the `vrpcHttp` transport replaces `http`. Every call underneath (`getBalance`, `eth_call`, `getLogs`, contract reads, gas estimates) runs unchanged and is verified before any value comes back. For stacks that are not TypeScript, or that would rather not embed the SDK, the same verification runs as a thin local verifying proxy: it sits beside your service, does the checks, and re-exposes a plain RPC endpoint you point your existing client at, so a Go or Rust backend gets verified responses with no change to application code.

5. The guarantee

Here is exactly what a client gets, stated once and precisely.

The guarantee

When a vRPC client accepts a response, then short of a break of the §3 assumptions: those exact bytes are the output of the approved node-plus-sidecar software, running in a genuine TDX enclave, for the chain the client asked, produced within the last Δ seconds (60 by default); and nothing outside the enclave - the operator and its whole infrastructure included - could read or change the node's code, memory, or disk.

The argument is a chain of smaller claims, each the same shape: if it failed, some §3 assumption would break. Where that assumption is cryptographic (A1, A2), the claim is as strong as the primitive. Where it is a trust assumption (A3, A4), it is exactly as strong as that trust, and we say so.

G1 - the bytes are the signed bytes. Suppose a client accepts R' the enclave never signed. Acceptance needs a valid signature over a pre-image carrying sha256(R'). Either R' collides with the signed body under SHA-256 (breaks A2), or the signature is over something the enclave never signed, an Ed25519 forgery (breaks A1), unless the attacker holds the key, which A3 confines to the enclave. The same argument covers the request hash. So the accepted bytes are the signed bytes.

G2 - the key belongs to a genuine enclave. The client verifies the quote to Intel's root and reads REPORTDATA = pubkey $\|$ nonce with its own fresh nonce; forging that quote breaks A3. So the verifying key belongs to a genuine enclave running the named measurement. What it does not say: because the key derives from the software's identity, not the machine's, G2 proves you are talking to the approved workload, not a particular node. A broken KMS release policy or a shared measurement would affect the key everywhere it is used. That residual is A3 and A4 applied to KMS, which the TCB names.

G3 - the software is the approved software. The quote certifies a measurement M; the client checks M against the published value. Serving a different binary under an approved M means a measurement collision (A2) or a forged quote (A3). How firmly M ties back to real source depends on the build, and this is the honest seam in the whole system: for a reproducible build anyone can rebuild and confirm; for a non-reproducible one the tie rests on the publisher. §6 is specific per component. For the node image in particular, the guarantee is that the pinned, public image runs unmodified, not that its source was independently reproduced.

G4 - memory cannot be read or changed from outside. Straight from A3: TDX seals the trust domain's memory against host, hypervisor, and operator. Composed with G3, what is sealed is exactly the approved image. This is the property an ordinary cloud cannot offer, and it is what closes the in-memory compromise of §1.

G5 - disk is confidential, and a tampered image will not run. dstack encrypts the node's data volume with LUKS, using a key released only to the attested workload [5]. Read it off the host and you get ciphertext. The read-only OS image is integrity-protected with dm-verity and measured into the quote, so a tampered image fails verification and KMS withholds the key: it will not boot rather than run corrupted. Two honest limits. Plain volume encryption gives confidentiality, not authenticated integrity, on the read-write data volume, and rollback protection is not automatic; an app that needs anti-rollback uses dstack's monotonic counters [5]. We state this rather than imply the disk is tamper-proof.

G6 - responses are recent, quotes cannot be replayed. The nonce in REPORTDATA kills quote replay: a replayed quote carries the wrong nonce (guessing it is 2^{-256}). For responses, the client compares the signed timestamp against its own clock and rejects anything older than Δ (default 60 s). One caveat, stated plainly because TDX gives no trusted clock: the timestamp is the enclave's, and a host that skews the enclave clock forward could make stale data look fresh within that skew. vRPC bounds staleness to Δ under the assumption the host does not do this. It does not provide per-request non-replay unless the client also tracks request ids.

G7 - the response is for the chain you asked. chain_id is the first field of the signed pre-image. A response signed for another chain rebuilds to a different pre-image and fails the check. Cross-chain substitution collapses into G1.

That is the whole guarantee: the cryptographic parts (G1, G7, the nonce in G6) are as solid as Ed25519 and SHA-256, and the rest is exactly as solid as TDX and the build provenance behind each measurement, which the next section takes up.

6. Staying out of the supply chain

The guarantee makes the operator untrusted. This section is about the software, because that is the other place trust hides, and it is where a careful reviewer pushes hardest.

Everything the attestation commits to is public. The app-compose that pins every running image by digest travels with the quote, so a verifier pulls each image and looks at exactly what runs, with nothing hidden. The SDK (`verifiable-rpc-sdk`, Apache-2.0) and the sidecar (`verifiable-rpc-sidecar`, AGPL-3.0) are open, and the npm packages publish through trusted publishing with no long-lived token. What differs across the pieces is how far you can trace an image back to its source.

The sidecar is the one image the operator builds. It is open source, built in public GitHub CI, Cosign-signed, and pinned by digest. Cosign proves the image came from that CI unmodified; a reproducible build, byte-for-byte from the public source, proves the image matches that source. Anyone can inspect the sidecar and rebuild it from the repository.

The node is not the operator's image, on purpose. The node runs the chain project's own official image - the one it publishes from its own org and CI (`ghcr.io/ton-blockchain/ton`, `nearprotocol/nearcore`, `offchainlabs/nitro-node`) - pinned by digest and run unmodified: not rebuilt, re-signed, or forked by the operator. The image you verify is the one the project shipped, exactly as if you pulled and ran it yourself, and the operator never enters its trust chain.

The honest part: these are official from the project, which is not the same as cryptographically verifiable. They are not Docker's curated Official Images, and most ship unsigned, without provenance, and not reproducibly. So "official" buys you the project's own pipeline, not an independent rebuild. How far you can check varies by client, on a gradient:

- **reth** - signed release binaries and a reproducible image. Full source to image.
- **erigon, nearcore, TON** - built in the project's own public CI, so the build is visible, but unsigned and not reproducible.
- **geth, Arbitrum Nitro** - official project images whose release CI is not publicly runnable,

so they are the hardest to check independently.

Where an image is not reproducible, what is inside a digest rests on the project's pipeline, not your own rebuild. That gap narrows without the operator inserting itself: these images run across many operators, so a bad release faces broad scrutiny, and a build-age window can hold back a release until it has been public long enough to be looked at. Whatever the provenance, the attestation exposes the exact digest, and G4 guarantees that whatever is pinned runs unmodified in memory. A reproducible node build is preferred where it exists. What the design will not do is put an operator build in the path and ask you to trust it.

The key service. dstack-kms, itself an attested confidential VM, derives and releases the signing key over a channel it opens only to a workload whose measurement it has checked [6]. That puts KMS in the trust base, and the TCB names it. dstack treats KMS as the system's root of trust and assumes its core components are correctly built; a root-key compromise would be serious, which is why dstack shards the key across nodes and supports rotation [5]. KMS is where the residual trust concentrates, kept as small as the design allows.

7. Performance

Verifying a response is a local signature check over bytes already in hand. It costs microseconds and adds no round trip. The only extra network call is the attestation, once per node, then served from cache. On Ankr's live Arbitrum vRPC endpoint a verified `eth_blockNumber` comes back in about the same time as a plain one; that time is network round-trip, not the check. A light client, by contrast, fetches and verifies a proof on each state read - a round trip vRPC's local check avoids.

By the numbers

- **Per-call cost:** a local Ed25519 verify over bytes already in hand - microseconds, no extra round trip.
- **Attestation:** one fetch per node, cached (default ∼1 hour), off the hot path.
- **Signed pre-image:** 80 bytes - chain_id, sha256(request), sha256(response), timestamp.
- **Freshness window:** $\Delta = 60\,\text{s}$ (default).
- **Coverage:** HTTP JSON-RPC; verified live on Ankr's Arbitrum vRPC endpoint.

8. Limitations and what we do not claim

Better to list these ourselves than have a reviewer do it.

- **Provenance, not canonical truth.** vRPC proves a response is the untampered, recent output of the attested node. It does not prove the node's view of the chain is the canonical one. An eclipsed or minority-fork node returns wrong data with a perfectly good signature. Pairing vRPC with a light client is future work, and the two are complementary: a light client can check a value is on-chain, vRPC can vouch for answers a light client cannot, such as whether a node quietly dropped logs.

- **Non-reproducible node images (§6).** For clients that do not build reproducibly, source

provenance rests on the publisher. vRPC still guarantees the pinned image runs unmodified.

- **Trusted time.** TDX provides no trusted clock; freshness holds under the assumption the host does not skew the enclave clock forward (§5, G6).
- **Fleet-scoped key.** The signing key proves the approved workload, not a specific node (§5, G2).
- **Inherited TDX limits** [4]: side channels, speculative execution, fault injection, physical attacks, and denial of service are out of scope.
- **KMS and root of trust.** vRPC roots trust in the Intel DCAP chain plus an approved-version registry, not in dstack's optional on-chain governance. KMS is a trusted, attested component and its integrity is assumed (§6).
- **Coverage.** HTTP only; WebSocket subscriptions are off the signed path, and ENS off-chain reads (CCIP, IPFS) are outside it.
- **Not claimed:** on-chain KMS, a decentralized or DAO root of trust, MPC key custody. vRPC's root of trust is centralized but auditable: Intel DCAP plus a registry anyone can check.

9. Related work

Town Crier put an oracle in an SGX enclave and proved the same class of thing for HTTPS feeds, one attested node serving an integrity-guaranteed answer [10]; vRPC does it for generic JSON-RPC on TDX. Automata's 1RPC runs an RPC relay in a TEE, but for metadata privacy, not response verifiability [11]. Flashbots runs block builders in TDX with RA-TLS and reproducible images, which proves the model for consensus-critical code but not for the RPC market [12]. redpill frames verifiable AI inference in a TEE with per-request attestation and an open verifier, the closest analog to what we are doing [13]. What none of them cover is the specific thing here: generic blockchain RPC, on TDX, with the response traced to the chain's own official image rather than a vendor build.

10. Conclusion

vRPC turns a hosted RPC endpoint from something you trust into something you check. The mechanism is small: run the chain's own node in a hardware enclave, sign each response inside it, ship the proof in the headers, verify locally, fail closed. The cost is one cached attestation per node and a microsecond check per call. What you trust shrinks to Intel's attestation and the build provenance behind a public measurement, and every improvement still ahead (local verification, reproducible builds, a public registry) makes even that smaller. vRPC does not ask you to trust Ankr. It shows you that you do not have to.

References

[1] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* https://bitcoin.org/bitcoin.pdf

[2] Intel. *Intel Trust Domain Extensions (Intel TDX) Whitepaper.* Feb 2022. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[3] Intel. *Intel SGX/TDX Data Center Attestation Primitives (DCAP): Quote Generation and Verification.* https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/

[4] P.-C. Cheng et al. *Intel TDX Demystified: A Top-Down Approach.* arXiv:2303.15540. https://arxiv.org/abs/2303.15540

[5] S. Zhou, K. Wang, H. Yin (Phala Network). *Dstack: A Zero Trust Framework for Confidential Containers.* arXiv:2509.11555, Sep 2025. https://arxiv.org/abs/2509.11555

[6] Phala. *dstack KMS and key delivery.* https://docs.phala.com/dstack-cloud/kms-and-key-delivery

[7] Dstack-TEE/dstack. *Attestation and RTMR mapping.* https://github.com/Dstack-TEE/dstack/blob/master/attestation.md

[8] S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA).* RFC 8032.

[9] NIST. *Secure Hash Standard (SHS).* FIPS 180-4.

[10] F. Zhang et al. *Town Crier: An Authenticated Data Feed for Smart Contracts.* ACM CCS 2016. https://eprint.iacr.org/2016/168

[11] Automata Network. *1RPC.* https://docs.ata.network/backed-by-pom/1rpc

[12] Flashbots. *Building Secure Ethereum Blocks on Intel TDX.* https://collective.flashbots.net/t/building-secure-ethereum-blocks-on-minimal-intel-tdx-confidential-vms/3795

[13] redpill. *Confidential AI Inference.* https://docs.redpill.ai/confidential-ai/how-it-works

vRPC SDK: https://github.com/w3tech/verifiable-rpc-sdk · sidecar: https://github.com/w3tech/verifiable-rpc-sidecar