TEE Architecture and Remote Attestation in t1

Abstract – Executive Summary

t1 leverages cutting-edge *Trusted Execution Environments (TEEs)* to achieve real-time, provably secure transaction execution on Ethereum, with any interim economic risk capped by an on-chain insurance fund that limits the value processed between periodic zero-knowledge (zk) proof checkpoints. This document specifies the planned *TEE architecture, event-driven remote-attestation model, and enclave lifecycle* that underpin t1's design. We detail how *Intel TDX* "Trust Domains" (isolated confidential VMs) serve as secure Executors, protecting roll-up computation with hardware-enforced isolation. We describe a novel *attestation pipeline* in which each Executor Trust Domain generates an attestation quote—a hardware-signed blob, produced by Intel's TD Quoting Enclave, that contains the TD's measurement hash (*MRTD*), current security-version numbers (SVNs), and a 64-byte `REPORT_DATA` payload bound to a verifier-supplied nonce—when it first joins the validator set, upgrades its software, or refreshes its Trusted Computing Base (TCB); the quote is then *verified inside a zero-knowledge VM (zkVM)*, compressing the result into a Remote Attestation Proof.
While this Remote Attestation Proof is built with zero-knowledge technology, it serves a different purpose from a ZKP checkpoint: it certifies the authenticity of the Executor's hardware and binary, whereas a ZKP checkpoint cryptographically certifies the correctness of the roll-up's state-transition computation. On-chain smart contracts can therefore trustlessly verify—at every security-relevant event—that authorised enclaves remain authentic and untampered, and are running the expected code.

The same architecture enables t1 enclaves to operate a *shared (partially) encrypted mempool* based on hybrid public-key encryption (HPKE) (eliminating MEV by keeping pending transactions confidential) and to enforce enclave-signed *access controls* for protocol operations and upgrades. We further introduce t1's cross-chain interoperability support: enclaves can perform authenticated reads of and writes to external blockchain state via co-located full nodes.

We distinguish hardware attestation from full cryptographic proofs, such as ZKP with formal verification methods, thereby clarifying TDX isolation guarantees. We are using unpredictable nonces (`report_data`) to prevent replay of attestation quotes, and requiring open-source, reproducibly built enclave binaries so that measurements can be independently verified. Finally, we present a robust TCB recovery and patching strategy: the protocol detects outdated or compromised enclaves, triggers mandatory re-attestation after critical updates, and rotates sealed keys to maintain security. A companion paper will provide a detailed description of the insurance-fund mechanism and ZKP checkpoint cadence.

In summary, this document provides both high-level and in-depth specifications of t1's TEE-based security architecture, ensuring transparency and rigor in how t1 combines TEEs and Ethereum to achieve fast yet trust-minimized roll-up execution.

Medium-Depth Summary

t1's design marries hardware-based trusted computing with blockchain verification to enable instant yet secure roll-up finality. Each t1 Executor runs inside an Intel TDX Trust Domain (TD)—a hardware-isolated virtual machine whose memory is encrypted and inaccessible to the host system. Within these TDs, the roll-up's state-transition function (transaction-execution engine) operates on private state data and mempool transactions, shielded from even the node operator's OS. An on-chain counter tracks the total value handled via this fast TEE path; when that counter reaches a preset insurance limit, the protocol first incentivizes the posting of a ZKP checkpoint. Then it ramps up the incentive and halts once too close to the crypto economic budget. Only then does it continue processing further funds.

To assure all parties that these protected Executors are running the correct code, t1 employs an event-driven remote-attestation system: each TD produces a cryptographically signed quote when it is first authorised or whenever the protocol image, version tag, or platform TCB changes. The attestation flow uses TDX's `SEAMREPORT` instruction to create a measurement report, which is then signed by Intel's quoting enclave to yield a remotely verifiable quote that embeds a fresh nonce in `report_data`, preventing replay attacks.

Because these quotes are large and computationally expensive to verify, t1 makes verification trustless and efficient on Ethereum through a zkSNARK-based Remote Attestation Proof verifier. Off-chain, a specialised zkVM (e.g., Risc0) re-executes the DCAP verification logic against Intel's certificates and revocation lists, then emits a constant-size proof. On Ethereum, t1's verifier smart contract registers or renews a validator only when it receives such a valid Remote Attestation Proof; thereafter, block proposals and cross-chain signatures from that validator are accepted so long as the Node Registry marks its attestation as allowed.

Beyond attestation, the document details how t1's TEEs support an encrypted mempool, enclave-keyed protocol authorization, and secure cross-chain reads/writes. Encrypted transactions remain confidential until executed inside the enclave, thwarting MEV. Crucially, only the privacy-sensitive payload is cipher-texted—sender, nonce, gas-limit, max-fee and signature stay in plaintext so sequencers can still rate-limit spammers, enforce fees and size limits, and drop malformed blobs without first decrypting them, avoiding the DoS pitfalls of a fully opaque mempool.

A secure provisioning mechanism transfers the sealed `root_secret` peer-to-peer via remote attestation to enclaves whose measurement has been verified on-chain. The transfer occurs over an attested ECDH channel established between allowed nodes and newly verified ones;

no centralized distributor or plaintext exposure exists. All critical protocol actions—state updates, upgrade approvals, bridge messages—require signatures from enclave identity keys that are themselves bound to these attestations.

Finally, we explain how Executor TDs act as followers of external chains, verify state proofs inside the enclave, and record every external read in a *Proof-of-Read Trie* that is published alongside the roll-up state root. Outbound cross-chain writes are best-effort and must be confirmed by subsequent reads.

This architecture clarifies what guarantees TEEs provide (hardware-backed isolation and binary-level integrity) versus what ZKP checkpoints provide. By insisting on open-source, reproducible binaries and nonce-bound quotes, t1 extends traditional TEE trust into an auditable, permissionless blockchain context—delivering real-time settlement while requiring fresh proofs only at genuine changes in the system's trust assumptions.

1. Introduction and Background

1.1 Trusted Execution Environments (TEEs) in Blockchain. Blockchains typically assume that all computation by validators or sequencers is fully deterministic and eventually verified (either by every node's re-execution or via succinct proofs). The *t1* rollup introduces a semi-trusted layer in this model: specialized hardware environments known as **Trusted Execution Environments (TEEs)** perform the rollup's transaction execution and immediately vouch for their results. A TEE is a secure area of a processor that guarantees code and data loaded into it are protected with respect to integrity and confidentiality. Even if the host operating system or hypervisor is malicious, it cannot tamper with or inspect the TEE's execution. Intel SGX, Intel TDX, AMD SEV-SNP, and AWS Nitro Enclaves are prominent TEE technologies that isolate computation from a potentially compromised OS. The security of TEEs ultimately relies on hardware and low-level firmware (the TCB) that is trusted to enforce said isolation. In exchange, TEEs offer powerful capabilities for blockchain systems: they can keep data secret (e.g., mempool transactions, private state), and they can produce proofs that convince others of what code they are running, and of what output it produces. Projects like *Town Crier* demonstrated that TEEs (SGX enclaves) can act as secure oracles, fetching external data and proving to smart contracts that the data came from an untampered source. More recently, Layer-2 protocols and research prototypes have begun integrating TEEs to accelerate off-chain computation while maintaining verifiability. In a multi-prover rollup architecture, a TEE-based prover can provide instant attestations of correctness (based on hardware trust) to complement slower cryptographic proofs.
t1 follows this path by using TEEs to achieve *real-time settlement* of execution results while requiring fresh hardware proofs only at onboarding, upgrades, or TCB-revocation events, bridging the gap between honest-majority consensus assumptions and full zero-knowledge proofs.

1.2 Intel TDX and Trust Domains vs. Enclaves. t1 specifically employs **Intel Trust Domain Extensions (TDX)** for its Executor environments. It is important to clarify TDX's model of trust domains vis-à-vis the classic enclave model (as in SGX). In SGX, an *enclave* is a protected region within a process's address space – a relatively small memory region with strict limits, requiring applications to be refactored into enclave and non-enclave parts. By contrast, **Intel TDX** creates isolated virtual machines called **Trust Domains (TDs)**. With TDX, an entire VM (including its OS and applications) runs in a special isolated context (SEAM mode) with memory encryption, so the *whole VM is effectively an enclave*. The hypervisor (VMM) and host OS are untrusted and cannot read or modify the TD's memory. Memory belonging to a TD is automatically encrypted with a per-VM key and integrity-protected. This design overcomes some SGX limitations (notably the memory size constraints and the need to rewrite apps for enclaves). In t1, each node runs its critical execution component inside a TDX TD – essentially, we deploy the rollup client as a dedicated confidential VM. We will use the term '**Trust Domain**' for these TDX-protected VMs, and reserve '**enclave**' for SGX-style enclaves or as a general term for isolated execution contexts. For example, Intel's quoting enclave (part of the attestation process) is an SGX enclave running on the host, while the rollup Executor is in a Trust Domain. Table 1 below summarizes the structural and operational differences between SGX enclaves and TDX Trust Domains in the context of t1.

Table 1. Comparison of SGX Enclaves vs. TDX Trust Domains

Feature	SGX Enclave	TDX Trust Domain (TD)
Isolation scope	Process-level enclave	Entire virtual machine (VM-level enclave)
Memory size	Limited (typically <128 MB)	Large (can span multiple GBs)
Application model	Requires enclave-aware application design	Supports unmodified Linux OS and applications
Measurement identity	`MRENCLAVE` (enclave content hash)	`MRTD` (initial VM memory hash)
Attestation signer	Quoting Enclave (QE)	TD Quoting Enclave (TDQE, runs via SGX)
Sealing and persistence	CPU-bound sealing key	VM-level sealing or SGX-assisted sealing
Integration in t1	Used for attestation infrastructure only	Used to host Executor environment

By adopting TDX, t1 places each Executor in its own trust domain, achieving strong hardware isolation without requiring a redesign of the entire Ethereum execution stack for enclaves. TDX is a recent technology – available on Intel 4th/5th Gen Xeon CPUs (Sapphire Rapids, Emerald Rapids) – and offers a path forward as SGX is repositioned for more specialized use cases on newer chips. We assume our infrastructure (cloud bare-metal deployments) provides TDX-capable servers. The threat model assumes the worst-case scenario for everything outside the trust domain: we do not trust the hypervisor, host OS, or any external party. Only the CPU/TDX module and the code running inside the TD (which we control and attest) are trusted to maintain security.

1.3 Remote Attestation Fundamentals. A cornerstone of confidential computing is **Remote Attestation (RA)**. Attestation is the process by which a TEE proves to a remote verifier what software is running inside it, and that the TEE is genuine and up-to-date. In simpler terms, it's how our t1 trust domains convince Ethereum (and users) "I am running the official rollup software in a secure enclave, not a modified or compromised version." Attestation typically works by generating a cryptographic *measurement* of the enclave/TD (like a hash of its initial state or binary) and then having the hardware or its manufacturer sign a certificate of that measurement. Intel's attestation infrastructure comprises multiple components: each CPU is equipped with embedded keys (e.g., the Provisioning Certification Key, or PCK, tied to the CPU's EPID or ECDSA identity). Intel issues certificates for these keys (via the Provisioning Certification Enclave, PCE) and provides a quoting mechanism (via the Quoting Enclave, QE, or TD Quoting Enclave, TDQE for TDX). When a trust domain is launched, the TDX module can produce a TD report (using `SEAMREPORT`), which contains the TD's measurements, security version numbers, etc., and is signed with a CPU-specific key. The untrusted host then passes this report to the TD Quoting Enclave (an SGX enclave running on the same host), which verifies the report (via a local attestation MAC) and then signs out an ECDSA *Quote* using the CPU's attestation key. This quote can be sent to a remote verifier; it contains (a) the measurement of the TD, (b) the TEE platform's TCB status (to indicate if firmware is updated), and (c) any user-provided data (like a nonce or public key). The quote is signed by an attestation key whose certificate chains to an Intel root of trust, allowing the verifier to validate that the quote is authentic. Modern attestation (Intel DCAP for SGX/TDX) no longer requires contacting Intel's online service for each quote; instead, Intel provides the necessary certificate chain and revocation lists (collectively called *attestation collateral*) so that anyone can locally verify a quote's signature. This is crucial for decentralization: t1 can perform quote verification in a self-contained manner, rather than relying on an Intel-run API (which could be a censorship or availability bottleneck).

For robust security, the attestation process must ensure *freshness*. This is why including a challenge or nonce in the attestation is a standard practice. In Intel's API, a verifier will send a 32-byte random nonce to the enclave, which the enclave includes in the `REPORT_DATA` field before asking for a quote. The resulting quote's hash binds that nonce, so it cannot be reused for a different session. t1 follows this by always using unique nonces (e.g., derived from the expected L1 block number or an incrementing counter) when requesting attestations from Executors. Additionally, the `REPORT_DATA` can carry other context: for instance, we include an identifier for which rollup instance and which network the attestation is for, to avoid any confusion or cross-use of quotes. The attestation quote verification involves checking the certificate chain (PCK certificate signed by Intel's CA), ensuring none of the certificates are revoked (via CRLs with statuses like `TDX TCB: out-of-date` or not) and verifying that the measurement matches the expected value for our software. If all checks pass, the verifier obtains confidence that the remote party is a legitimate TEE running our code on a platform at least as secure as a baseline (we might require, for example, TDX SVN >= X, meaning all known microcode patches are applied).

In summary, remote attestation in t1 provides a *root of trust* that anchors in hardware. Unlike a pure cryptographic proof of computation, attestation has a *trust assumption*: you trust Intel (and possibly the platform owner to some extent) that if the quote is valid, the code ran without tampering. We emphasize this difference: attestation proves the *integrity of the execution environment* but does not mathematically prove the *correctness of the output* – for the latter, a Remote Attestation proof of the computation itself is required. t1 actually combines both approaches: hardware attestation for immediate trust, and eventually ZKP checkpoint, via on-demand ZKP or periodic ZKP, for full validity proofs (as part of its multi-prover real-time proof architecture, though the ZKP checkpoint aspect for transactions is outside the scope of this TEE paper).

1.4 System Overview. At a high level *t1* operates as follows. A decentralised set of **Executor Trust Domains (TDs)**—Intel TDX confidential VMs—collectively maintain the L2 state. Users encrypt their transactions **with the published HPKE public key** and broadcast them; only TDs can decrypt these ciphertexts. A leader TD applies a deterministic ordering policy, executes the transactions, and derives a new L2 state root together with any auxiliary roots (e.g., a Proof-of-Read trie). Economic exposure on this fast TEE path is strictly capped by an on-chain insurance fund; see bullet 5 below.

1. **Block authentication without per-block attestation.**

   Each TD possesses an **enclave identity key** generated inside the enclave at onboarding.

   - The public half of this key is bound to a **zk-compressed attestation proof** that the TD submitted **once,** either at node registration, at least the most recent protocol/enclave upgrade, or after the latest TCB-revocation event.
   - The on-chain *Node Registry* contract stores this proof and flags the node as allowed.
   - For every subsequent L2 block, the TD simply signs `⟨state root ‖ block metadata⟩` with its identity key. Because the key is already anchored to a verified attestation, no fresh quote is required; the block producer includes only the signature, which other TDs and the L1 contract can check against the current allowed-set map.

2. **Event-triggered re-attestation.**

   A TD must regenerate an attestation quote and a new zk-SNARK **only when one of three events occurs**:

- **Onboarding / re-join.** A new or previously offline node is added to the validator set.
- **Protocol or enclave-code upgrade.** Governance publishes a new allowed measurement or protocol-version tag; all older measurements are invalidated in the *Node Registry*.
    - **Protocol Governance** is the on-chain authority whose actions are authorized by a threshold (e.g., ≥⅔ stake) of currently-allowed TEE validators (the "existing node set") signing an action slot with their enclave keys. Protocol Governance may bump `currentProtoVersion` and/or update `allowedMeasurements`. Upon activation, the Node Registry invalidates older measurements and marks validators not allowed until they re-attest.
    - A separately appointed k-of-n multisig (the **Security Council**) holds only time-boxed/scoped emergency powers intended to stop harm or censorship by a rogue TEE majority.
- **TCB-level revocation.** Intel or governance raises the minimum acceptable microcode SVN (or otherwise marks a vulnerability); the *Node Registry* enforces the higher `minTCB`.

Until a node posts a new proof that satisfies the updated policy, its signatures are rejected by both the consensus layer and the Canonical Bridge.

3. **Key provisioning gated by the latest proof.**

   Bridge-signing keys, threshold-signature shares, and the **root-secret epoch currently in force (from which Priv_HPKE is derived)** are handed to a TD only after its most recent zk-SNARK proof has been verified on-chain. If a validator falls out of compliance, or if governance toggles a `requireReattestation` flag in an emergency—key-distribution services withhold or revoke those secrets, instantly disabling the node's influence without affecting live peers.

4. **Real-time settlement preserved.**

   Although hardware proofs are not emitted per block, every new state root is still posted to Ethereum each second, together with a quorum of enclave signatures. The L1 contract finalises the block immediately, provided all signatures map to allowed TDs in the registry. Hence, the roll-up maintains **one-block settlement latency**—attestation latency is amortised over long intervals and does not appear on the critical path.

5. **Insurance-fund bounded risk.**

   Every block header reports two values—`blockValue`, the total amount transferred in that block, and `cumValueSinceZKP`, the running total of value moved since the last validity-proof checkpoint. A validator must refuse to sign any block whose `cumValueSinceZKP + blockValue` would exceed the current balance of an on-chain insurance fund maintained by governance. Once that ceiling is reached, validators pause value-bearing blocks until a zk-SNARK proof is submitted that validates all state transitions from the previous checkpoint forward, and resets `cumValueSinceZKP` to zero, and thereby reopens headroom under the insurance cap. The forthcoming **t1 ZKP-Fallback specification** will lay out the exact logic that makes this safeguard enforceable.

6. **Fallback to validity proofs.**

   Should a TDX vulnerability surface that cannot be mitigated in real time, governance may suspend TD signatures and revert to mandatory zk-SNARK validity proofs of the full state transition. This contingency preserves safety while the enclave software or Intel microcode undergoes urgent remediation.

The event-driven attestation cadence thus decouples block production from zk-proof generation costs, while ensuring that every validator's hardware-rooted trust is re-established exactly when the system's threat surface materially changes.

1.5 Early-Stage "Single-Node" Resilience. While the validator set is still forming, the roll-up may run on a single attested Executor Trust Domain. To avoid stalls and silent data loss during this phase we rely on the following lightweight guarantees.

- **I/O path.** Every outbound commit is signed inside the enclave; every inbound payload carries an ever-increasing nonce. Signatures stop the host from tampering, nonces stop it from replaying or re-ordering messages, and a simple heartbeat lets the enclave detect when the outside world stops responding.
- **Downtime handling.** If the TEE is shut down—gracefully or by crash—it seals its state, then on restart re-attests automatically and resumes from the last confirmed block. A configurable liveness fuse on-chain freezes deposits if no state-root appears within a few block intervals, unfreezing the moment a fresh attestation is posted or a standby node comes online.

These safeguards keep confidentiality, integrity and a bounded level of availability until a multi-node quorum takes over and ordinary consensus rules apply.

ℹ️ *The following sections assume familiarity with Ethereum rollup concepts including sequencers, state roots, and Merkle proofs. The focus is on TEE-related aspects. We aim to keep each section self-contained for clarity.*

2. TEE-Based Executor Architecture in t1

2.1 Trust Domains as Rollup Executors. In t1, each trusted Executor is an Intel TDX Trust Domain running a tailored Linux OS and the t1 rollup node software. The trust domain encapsulates the execution engine that processes L2 transactions and updates state. When a t1 node boots, it launches a TD via the TDX module: the CPU enters SEAM mode and measures the initial memory (which includes the OS kernel, t1 binary, and any supporting code). This measurement is essentially a cryptographic hash (SHA-384 in TDX's implementation) of the TD's initial contents. We denote this expected measurement as **M_e** (for example, the hash of the t1 enclave's boot image). Only if the measurement matches will the CPU allow the TD to fully initialize. Any tampering in the launch process would result in a different measurement, and the attestation later would reveal the mismatch, causing verifiers to reject it. Once running, the TD's memory is protected: all RAM pages assigned to it are encrypted with a hardware key unique to that TD. The memory controller ensures that if those pages are ever written to DRAM or accessed by the host, they remain encrypted. The encryption key is managed by the CPU and is not accessible to software. Even DMA (Direct Memory Access) from peripheral devices is blocked, or cannot read the TD memory. Furthermore, TDX provides integrity protection (optionally) – a cryptographic MAC is maintained for each memory line to detect any replay or bit-flip attacks on memory. The TD's CPU state (registers) is also saved and restored securely, such that when the TD is not executing, its registers are stored in an encrypted form, preventing the hypervisor from snooping on them.

In essence, once a t1 Executor TD is launched, it operates as if on a dedicated secure machine: the untrusted host can schedule it (start/stop the vCPU), but cannot see its internal state or influence its computation beyond denying service. This strong isolation is key to achieving what we refer to as *"confidential and integral execution"* – confidentiality meaning that the inputs (such as pending transactions) and intermediate states are hidden, and integrity meaning that no external tampering can alter the execution without detection. We avoid calling this "cryptographic isolation" in order not to confuse it with cryptographic proof systems; rather, it is **hardware-enforced isolation** using cryptography (memory encryption) under the hood.

2.2 Enclave Software Stack and Measurement. The software running inside the TD is composed of an OS, likely a minimal Linux distribution, plus the t1 node application. To reduce the TCB inside the enclave, we minimize unnecessary services and drivers in the TD's OS. Ideally, we use a slimmed-down kernel and possibly a *unikernel* or library OS approach for the rollup Executor. However, for practicality and compatibility with Ethereum clients (which often expect a standard OS environment), we currently use a standard Linux with only the required components. The entire stack is built reproducibly (see Section 7.4 on reproducible builds) allowing its measurement to be independently verified by developers. The measurement M_e includes the kernel, init system, and the t1 binary; if any of these change (even a single byte), the measurement will differ. We leverage this fact to enforce version control: when a new version of the rollup software is released (or a security patch is applied), it results in a new measurement, which will require updating the accepted values in the attestation verification process. This provides a form of binary authorization – only binaries that produce an approved hash will be recognized as valid Executor enclaves. A downside is that even benign changes (such as compiling the same source with a different compiler version) yield different measurements, hence the emphasis on reproducible builds so that a canonical binary can be agreed upon.

At runtime, the enclave stack includes a few important components: (a) a secure clock or trusted timestamp source (possibly using TDX's trusted time or synchronizing with the host but verifying it), since timing may be needed for time-based logic; (b) a source of randomness (the CPU's hardware RNG is accessible and can seed a DRBG inside the enclave); (c) networking and storage interfaces proxied via the untrusted host – data coming from outside is validated or decrypted as needed inside the enclave. For example, transactions arrive encrypted, so even if the host tries to feed corrupted data, the enclave will either detect decryption failure or malformed input. For cross-chain read data, as discussed, cryptographic proofs are provided and verified by the enclave; thus, the host cannot simply lie about an external account balance without being caught by a mismatched Merkle proof or signature.

2.3 Trust Domains vs. Multiple Enclaves. In earlier SGX-based systems, it was common to slice a roll-up node into several process-level enclaves: one enclave might hold the execution engine, another the key-management code, while external processes handled networking and storage. That approach kept individual binaries small, but it also forced every enclave to hand sensitive data across an untrusted operating system, reopening the very side-channels the TEE was meant to close. With Intel TDX, we place the entire roll-up stack—execution engine, encrypted mempool handler, and the light- or full-client logic that services cross-chain reads—inside a single confidential virtual machine. All code and data, therefore, live behind the same memory-encryption boundary, eliminating inter-enclave gaps and simplifying the threat model to "everything inside the TD is trusted; everything outside is not."

This monolithic layout also streamlines attestation. TDX reports and quotes cover the whole VM in one shot, so each validator needs only a single measurement, a single SVN vector, and—per security-relevant event (§ 3.6)—a single zk-compressed proof that `NodeRegistry` can record or revoke. Operational overhead falls sharply: operators no longer juggle a dozen enclave quotes, and the on-chain registry maintains just one allowed attestation record per node.

Finally, the resource cost is modest. Running a pruned Ethereum full client alongside the roll-up engine consumes on the order of 8 GB of TD memory, and any archival snapshots needed for edge-case proofs can reside on read-only, encrypted volumes outside the hot footprint. The result is a cleaner, fully self-contained Executor whose security and lifecycle management align neatly with the event-driven attestation policy defined in Section 3.

2.4 Secure Communication. Once the enclave is up, it needs to communicate with the outside world, such as sending the attestation quote to a verifier, submitting transactions to L1, and receiving user transactions, among other tasks. All network traffic leaving the enclave is encrypted at the application level. For instance, the enclave might establish a TLS connection out to a public endpoint.

However, since the host can tamper with or observe traffic, we typically treat it like a MITM adversary. Therefore, the enclave will use end-to-end secure channels whenever confidentiality or data integrity is required. A key example is the mempool: users will encrypt their transactions to the rollup's public key and send them over normal networks; the host just forwards the ciphertext to the enclave, which can decrypt it. When the enclave needs to output a result (like a new state root and proof), it often signs it with its enclave identity key, so any recipient can verify it originated from a valid enclave (by checking the signature against the enclave's public key that was included in an attestation on-chain).
In essence, digital signatures from the enclave serve as a second layer of attestation for individual messages, complementing the event-triggered attestation proofs that are submitted only upon validator onboarding, upgrade, or TCB revalidation. We ensure that each enclave has a unique keypair and that keypair's public key is either in the attestation's `report_data` or is it derivable from a known seed sealed in the enclave. In Section 5, we outline how these keys are managed and utilized for access control.

In summary, the architecture treats the trust domain as a black box that only interfaces with the external world through attested, authenticated channels. Any data coming out is signed or part of an attested payload; any data going in is encrypted or accompanied by proofs that the enclave will verify. This way, even though the host facilitates IO, it cannot violate the security or correctness of the enclave's operations. With the basic execution environment described, we now turn to the remote attestation process, which bootstraps trust in these Executors from the standpoint of the Ethereum blockchain.

3. Remote Attestation Pipeline

The remote attestation pipeline in our architecture comprises six stages (Sections 3.1–3.6) that connect trusted hardware attestations with on-chain verification and policy enforcement. This end-to-end process begins with the generation of a hardware-backed attestation quote inside a Trusted Execution Environment (TEE) and culminates in the on-chain validation of that attestation via zero-knowledge proofs, followed by event-driven policy actions. Each stage is designed to preserve security and integrity while minimizing on-chain overhead. In the following subsections, we detail each step: from quote generation and zk-SNARK proof production to on-chain verification and the enforcement of attestation policies.

3.1 Generating Attestation Quotes (TDX → Quote). A *t1* Executor Trust Domain (TD) produces a hardware quote **only at attestation rounds triggered by security-relevant events**—specifically, node onboarding or re-joining, enclave-code or protocol upgrades, and platform TCB changes mandated by governance (cf. Section 3.6). Upon receipt of the verifier's nonce, the enclave invokes the TDX report instruction (analogous to SGX's `EREPORT`; on TDX platforms, this is `TDG.MR.REPORT`) to obtain a local **TD-report** containing (i) the measurement of the TD's initial memory image, (ii) the current CPU/firmware security-version numbers, and (iii) the `REPORT_DATA` hash that binds the nonce, the validator's public key, and the current protocol-version tag.

Because a TD-report is not directly verifiable off-chip, it must be converted into a **TDX Quote** by Intel's quoting stack resident in the untrusted host. The enclave transmits the report to the host via the hypervisor-mediated shared-memory interface; the host then calls the TD Quoting Library, which delegates to the **TD Quoting Enclave (TDQE)**. TDQE—an Intel-signed SGX enclave—verifies the report's MAC, consults the **Provisioning Certification Enclave (PCE)** to obtain the CPU's Provisioning Certification Key (PCK) certificate, and signs the report with the CPU's attestation key.

Assuming Intel's root keys remain uncompromised, the Quote is cryptographically unforgeable. In *t1*, the immediate consumer of every Quote is an off-chain **zkVM attestation-verifier circuit** that re-executes the Data-Center Attestation Primitives (DCAP) checks and emits a succinct zk-SNARK attesting to the Quote's validity; the on-chain `NodeRegistry` accepts or rejects the validator based on that proof.

3.2 Off-Chain Quote Verification (zkVM Validation).

Each attestation round (Section 3.6) produces a single TDX Quote whose validity must be exhaustively checked before the validator's signing key is accepted for subsequent blocks. Because the full Intel DCAP verification procedure is far too heavy for an Ethereum contract, *t1* executes it inside a zero-knowledge virtual machine and publishes a compact zk-SNARK proof.

Verification routine executed inside the Risc0 zkVM

Step	Purpose	Inputs
1. Parse quote and collateral	Deserialize the Quote; extract the embedded PCK certificate, any intermediates, and all metadata fields.	Raw Quote bytes, Intel CRLs, hard-coded Intel root certificate
2. Certificate-chain validation	Confirm that the PCK certificate chains, via valid intermediates, to the hard-coded Intel root; reject if any certificate is expired or malformed.	Same X.509 chain
3. CRL and TCB checks	Ensure that neither the PCK certificate nor its issuer appear in the supplied CRLs; require that the CPU security-version	Intel CRLs, value of `minTCB`

	number (SVN) is at least the governance-defined minimum.	
4. Quote-signature check	Verify the ECDSA-P256 signature over the Quote body using the attestation public key contained in the certified PCK.	Quote body, PCK public key
5. REPORT_DATA validation	Recompute `SHA256(nonce	
6. Measurement admissibility	Check that the TD measurement in the Quote appears in the governance-supplied allow-list corresponding to the stated protocol version. The allow-list root is provided as an input, and the zkVM verifies a Merkle inclusion proof for the measurement hash.	TD measurement, allow-list root, inclusion proof

If all predicates hold, the program writes to its public journal:

```
validatorPubKey
measurementHash
protocolVersion
TCB_SVN
QuoteValid = 1
```

Risc0 then produces a STARK trace of the execution and compresses it into a Groth16 zk-SNARK; size is about 200 bytes and on-chain verification costs roughly 250 k gas.

On-chain acceptance logic

The `NodeRegistry` contract:

1. Verifies the zk-SNARK.
2. Confirms that `protocolVersion` equals the governance constant `currentProtoVersion`.
3. Stores `(validatorPubKey, measurementHash, attestedAt)` as an allowed validator record.
4. Emits `ValidatorActivated(validatorPubKey)`.

From that point forward, every L2 block header, withdrawal root, or cross-chain message signed by the recorded public key is accepted until the next attestation trigger (upgrade, TCB bump, emergency re-validation, or long-horizon expiry). A single zk-proof thus amortizes hardware-rooted trust over the entire epoch, while any Ethereum participant can independently audit the attestation by checking the proof on chain.

3.3 On-Chain Verification and Enforcement. With attestation now confined to discrete security-relevant rounds (Section 3.6), the zk-SNARK proving a validator's TDX Quote is carried **only in a dedicated attestation transaction**, not in routine block-submission calls. Two on-chain components share the enforcement logic.

NodeRegistry – admission of validator keys.

Whenever a Trust Domain completes an attestation round—whether at initial join, after a protocol/enclave upgrade, or following a TCB-revision—it submits to *NodeRegistry* the tuple `(zkProof, quoteHash, validatorPubKey, protoVersion, nonce)`

The contract invokes the Groth16 pre-compile to verify `zkProof`.

If the proof is sound and its public outputs satisfy:

- `protoVersion` equals the governance constant `currentProtoVersion`;
- the extracted measurement hash appears in the `allowedMeasurements` set for that protocol version; and
- the reported `TCB_SVN` is at least `minTCB`,

Then the pair `(validatorPubKey , measurementHash)` is recorded as allowed, together with the timestamp `attestedAt`. Any subsequent governance action that increments `currentProtoVersion`, raises `minTCB`, or flags a key for emergency re-attestation automatically renders the corresponding entry not allowed until a new proof is lodged.

CanonicalBridge – block-level acceptance.

Each L2 block header is furnished with a digital signature under the validator's enclave key. On reception, *CanonicalBridge* verifies that the signer's public key is listed as allowed in *NodeRegistry* and that the attestation timestamp is not older than the permitted `maxAge`. Only if these conditions hold does the contract accept the state-root tuple (execution, withdrawal, and proof-of-read roots) and finalise the block on Ethereum.

3.4 Replay Protection with Nonces: Remote-attestation rounds in *t1* are infrequent but critical to security. To guarantee that each round is fresh and cannot be replayed, the verifier issues a unique 256-bit **challenge nonce**, and the Trust Domain (TD) must bind that nonce to its quote as follows:

1. **Challenge acquisition.**

   The TD calls `NodeRegistry.requestNonce(nodeId)` and receives a Keccak-derived random value `nonce`.

2. **Nonce commitment.**

   Before invoking `SEAMREPORT`, the TD computes `REPORT_DATA = SHA256(nonce ∥ currentProtoVersion ∥ nodeId ∥ HashPubHPKE)` and embeds the 64-byte digest into the report. Because `REPORT_DATA` is covered by the TDX signature, the nonce becomes inseparable from the quote.

3. **zk-compressed verification.**

   The off-chain zkVM takes `nonce` as a public input; if the digest inside the quote does not match, the proof fails. On success, `NodeRegistry.register()` records the validator as allowed and forever burns the nonce to prevent reuse.

A single fresh nonce is sufficient per attestation cycle. Event-driven policy ensures that cycles occur only on onboarding, upgrades, or TCB events, not on every block.

3.5 Binding Attestation to Validator Credentials and Protocol Version: Because *t1* emits a hardware quote only when a validator **(re)attests**—not for every L2 block—the quote must establish a stable cryptographic anchor that remains valid across the many blocks the node will later produce. We bind the quote to two pieces of data that uniquely identify the validator's role in the current epoch:

Field encoded in `REPORT_DATA`	Purpose
`validatorPubKey` (32 / 33 bytes)	Long-lived ECDSA/EdDSA public key that the enclave will use to sign every block header, withdrawal root, and cross-chain message until the next attestation event.
`currentProtoVersion` (4–8 bytes)	Monotonically increasing tag set by governance each time enclave code or consensus logic changes. Ensures mixed-version execution cannot occur.
`HashPubHPKE` (32 bytes)	Binds the attestation to the mempool-encryption public key that wallets must use for this epoch.

`REPORT_DATA` is computed as `SHA256(nonce ∥ validatorPubKey ∥ currentProtoVersion ∥ HashPubHPKE)` and inserted into the `SEAMREPORT` before the quote is generated.

Verification path

1. *zkVM attestation verifier*
   - Confirms the quote signature and TCB status.
   - Exposes `(validatorPubKey, currentProtoVersion, HashPubHPKE)` as public outputs in the SNARK's journal.
2. *Node Registry contract*
   - Checks that `currentProtoVersion` equals the governance-defined constant.
   - Stores `(validatorPubKey, measurementHash, HashPubHPKE, attestedAt)` as an allowed validator record.
   - Rejects duplicates or proofs carrying stale `protoVersion` values.

Block-production phase

- Every roll-up block header is signed by `validatorPrivKey`.
- Peers and the Canonical Bridge verify the signature against the public key on file.
- If governance later bumps `currentProtoVersion` or raises the minimum TCB, the Registry marks all prior records not allowed; validators must produce a new quote and proof before their signatures regain validity.

By binding the validator's long-lived signing key and the protocol version directly into the attestation, *t1* ensures that:

- A rogue host cannot swap in a different private key after attestation.
- Mixed software versions cannot coexist in the allowed validator set.
- Each ordinary block remains lightweight—only a digital signature—while the heavy hardware proof appears precisely when the trust assumptions change.

- Wallets and peers can enforce a single, attested mempool encryption key per epoch, eliminating replay attacks with stale ciphertexts.

3.6 Attestation Frequency and Amortization: A trade-off exists in determining the frequency of attestations. Triggering a fresh proof for **every** roll-up block—potentially as often as once per second—would overload the proving pipeline and inflate on-chain costs, because even an optimized zk-proof for attestation still incurs measurable latency and gas. Conversely, attesting too infrequently weakens the real-time assurance that validator Trust Domains (TDs) remain compliant with the latest protocol and security posture.

In our design, we lean towards an *event-driven attestation policy* rather than a per-block cadence. A validator must generate a new zk-compressed attestation only when a security-relevant transition occurs; between such events, it may sign an arbitrary number of blocks without re-proving. The policy is summarised below.

Trigger event	Security purpose	Contractual gatekeeper	Effect on validator
Node onboarding / re-join	Establishes first-time authenticity (TDX platform, approved image, current TCB)	`NodeRegistry.register()` requires fresh zk-proof	Node listed as allowed only after proof verifies
Protocol or enclave-code upgrade (new image hash `M′`, new protocol-version tag)	Enforces homogeneous logic across all validators	Governance call updates `allowedMeasurement` and/or `currentProtoVersion`	All validators must re-attest with `M′`; prior proofs invalid
TCB-level revocation (e.g., microcode SVN bump after vulnerability disclosure)	Removes trust in unpatched hardware/firmware	Governance call updates `minTCB`; zk-proof must reveal `TCB_SVN ≥ minTCB`	Unpatched nodes excluded until re-attested
Emergency re-validation (suspected compromise or key misuse)	Provides rapid containment without waiting for scheduled upgrades	Security-council multisig sets `requireReattestation(node)` flag	Node's signatures rejected until new proof posted
REPORT_DATA validation	Tie the quote to a fresh nonce **and** the current `HashPubHPKE`, preventing replay and proving the enclave is using the authorised mempool-encryption key.	`NodeRegistry.verifyProof()`	If the hash matches, the validator is activated; if it does not, the registration transaction fails and the node stays not allowed.
Long-horizon expiry (e.g., `maxAge = 12 months`)	Slow-moving back-stop in case no explicit trigger occurs	`NodeRegistry.isAllowed(node)` checks `block.timestamp − attestedAt ≤ maxAge`	Validator must refresh proof before deadline

No per-block proofs are emitted. Gas expenditure for zk-proof verification is amortised over weeks or months; steady-state block time is decoupled from proving latency. Key provisioning (bridge-signing keys, mempool-decryption keys) remains strictly gated on the **most recent accepted attestation**, so a validator that fails to re-attest after a trigger event automatically loses the cryptographic capability to influence state roots or decrypt private transactions.

Operationally, this yields:

1. **Unconstrained throughput between events.** Blocks are produced continuously; attestation overhead appears only at onboarding or upgrade epochs.
2. **Predictable upgrade windows.** Governance can schedule protocol upgrades or TCB bumps, knowing that validators must re-attest exactly once per event.
3. **Bounded long-term risk.** A maximum-age parameter enforces periodic renewal if no explicit triggers occur, maintaining a hard upper bound on attestation staleness.

This event-driven cadence retains real-time settlement—state roots still commit every block—while ensuring that cryptographic proofs are generated precisely when the underlying trust assumptions change.

4. Encrypted Mempool and Key Provisioning

One of t1's distinguishing features is its encrypted mempool. The idea is that the privacy-sensitive payload of every pending transaction (recipient, value, calldata, etc.) is encrypted so that only the TEE-based Executors can decrypt and it. This provides strong MEV resistance: no outside entity (miners, validators, or even the sequencer operator's non-enclave self) can read the encrypted portion of the transaction payload to profit from ordering (e.g., no front-running or sandwich attacks by outsiders). The enclave will decrypt that payload in a secure environment and order them according to a fair policy (such as first-come, first-served or a verifiable auction mechanism) without leaking their content beforehand. This section explains how we implement the encrypted mempool and securely manage the keys associated with it.

4.1 Mempool Encryption Scheme. t1 secures its mempool with a hybrid-encryption hierarchy built on the IETF HPKE standard. At genesis, each Executor Trust Domain seals a 32-byte *root secret* inside its enclave and deterministically derives a single, static HPKE key-pair **(Pub_HPKE, Priv_HPKE)**. The public half, **Pub_HPKE**, is stored on-chain in the *Node Registry* next to the enclave's attestation hash, so a dApp can fetch the key straight from Ethereum without relying on any off-chain server. From the user's point of view, this is seamless. Before triggering the wallet's signing prompt, the *dApp's* helper code fetches `Pub_HPKE`, caches it, performs a sub-millisecond X25519/HPKE encryption of the payload, and embeds the ciphertext into the transaction's `data` field. *Only then* does it invoke the wallet, which signs *exactly* those bytes. It samples a fresh 256-bit session key *k_tx*, encrypts *k_tx* with HPKE to form **ciphertext$_1$**, encrypts the transaction payload with AES-GCM_{k_tx} to form **ciphertext$_2$**, concatenates the two, and submits the result as the transaction's calldata. Header fields—`from`, `nonce`, `gasLimit`, `maxFeePerGas`, `maxPriorityFeePerGas`, and the `v,r,s` signature—stay in plaintext so sequencers can rate-limit spammers, enforce fee floors, and drop malformed blobs without decryption. The entire sequence happens in the background; the user experiences the familiar single click and, if anything, enjoys lower slippage because their order is no longer exposed to the public mempool. Inside every Executor enclave, the matching **Priv_HPKE**—re-derived locally from the same sealed *root secret*, never shipped across machines—unwraps *k_tx*, decrypts ciphertext$_2$, and executes the transaction. Because all enclaves share the key pair by construction, any allowed node can process any pending transaction, yet a compromise of one enclave does **not** expose historical payloads once the network rotates the *root secret* (see Section 4.4). The scheme, therefore, keeps symmetric-cipher performance for bulk data while eliminating the single global secret and enabling clean, epoch-level forward secrecy, all without altering the everyday wallet workflow.

4.2 Root-Secret Generation and Storage: The encrypted-mempool scheme hinges on a 32-byte *root_secret* that must be born inside a TEE so that nobody—operator, cloud admin, or attackerever handles it in the clear. At genesis, the launch script boots a throw-away TDX Trust Domain whose only job is to call `RDRAND`, collect 32 bytes of entropy and seal the result to its own measurement. That TD then transmits the sealed blob to the first production Executor over an attested channel and terminates.

After its one-time creation the `root_secret` must occasionally migrate—such as new validators, disaster recovery, or region failover—without ever leaving a hardware boundary. We reuse the attestation machinery that already proves code identity:

- **Join request.** A fresh Executor TD comes online and posts a TDX quote proving it runs the authorised t1 image.
- **Mutual key exchange.** A live Executor verifies the quote, checks governance allow-lists, then performs an ECDH whose public key was embedded in the newcomer's `REPORT_DATA`.
- **Secret transfer.** Over the AES-GCM tunnel established by ECDH, the live node sends the *sealed* root_secret—or, equivalently, re-wraps the raw bytes under the newcomer's Pub_HPKE. No plaintext ever crosses the host kernel.
- **Local derivation.** The new TD unseals the blob, derives *(Pub_HPKE, Priv_HPKE)*, and joins block production.

Unlike SGX, the current TDX firmware does not offer a built-in sealing key. Instead, we rely on open-source projects, which expose a per-CPU secret rooted in the hardware's Chip-Endorsement Key and bound to the TD's `MRTD`. Every Executor links this provider and calls it to encrypt the `root_secret` before persisting to disk; on reboot the same code path unseals the blob if—and only if—the measurement still matches. For portability across minor software upgrades, we can switch the seal policy to *MRSIGNER*, trading a slightly larger trust circle (anything signed by the t1 release key) for a smoother operator experience.

This two-layer approach—remote-attested hand-off for cross-machine moves, sealed storage for local restarts—keeps the `root_secret` inside confidential memory from birth to retirement, while giving operators the flexibility they need to scale the validator set or recover from outages without touching any private HPKE material.

4.3 Enclave-Sealed Storage. While an Executor is live the `root_secret` (and any short-lived HPKE private keys retained during the grace window) reside only in the enclave RAM. A power loss or host reboot, however, must not strand the network by erasing the secret. TDX itself offers no built-in `EGETKEY` primitive like SGX, so we embed an open-source library that asks the TDX module for a CPU-unique "report key" that is cryptographically tied to the VM's measurement (`MRTD`). That key feeds an AES-GCM wrapper, which writes an encrypted blob to the node's SSD. On restart, the same path decrypts the file, but only if the measurement (or, when configured, the developer's *MRSIGNER* value) still matches, thereby blocking replay on alien hardware. For operators who prefer looser coupling across minor software bumps, we expose a switch that seals to *MRSIGNER* instead of `MRTD`; anything signed by the official t1 release key can then reopen the blob, simplifying blue-green upgrades while still preventing arbitrary binaries from unsealing it.

In practice, the secret is transmitted via two routes: live nodes pass it to newcomers through the attested ECDH channel described in Section 4.2, and every node also maintains its own sealed backup for crash recovery. The attested hand-off ensures that no plaintext ever touches storage, while the sealed file prevents re-asking peers after a routine reboot. Together, they give the network both safety and operational flexibility without diluting the guarantee that the `root_secret` never leaves confidential memory.

4.4 Key Rotation. At a fixed cadence or at any moment, the security governance of an emergency, each allowed Trust Domain hashes its sealed root secret, derives a fresh HPKE key-pair, and co-signs an `updateHPKE(pub, epoch)` call that places the new public key in the *Node Registry*. The update becomes canonical as soon as two-thirds of the stake-weighted Executors have signed. Wallet software that monitors the contract **sees no HPKE details; the dApp updates its cached `Pub_HPKE`, encrypts new payloads accordingly,** and the wallet continues to sign opaque bytes. To cover messages already in flight, every enclave retains the previous private key for a short, configurable grace period—three epochs by default—after which the old key is erased, preventing any later breach from revealing past traffic. If a compromise is suspected before the scheduled boundary, the governance can invoke `forceRotate`, which truncates the current epoch and triggers the same ratchet and publication sequence at once. Because all derivation occurs locally within the Trust Domains and only the public half is broadcast, the procedure transfers no private material across machines, requires no downtime, and preserves forward secrecy with a single deterministic step.

4.5 Interaction with MEV and Ordering Policies: With an encrypted mempool, users are protected from external frontrunning. The enclave could still theoretically reorder transactions arbitrarily since it decrypts them. To ensure fairness, we have to design the ordering policy inside the enclave code (which is open source, so anyone can audit how ordering is done). One policy is FIFO by arrival time (the host could delay delivering some tx to the enclave, but if the host is malicious in that way, that might be detectable via attestation of time or via multiple enclaves cross-checking the order of receipts). Another policy might involve a sealed-bid auction (but then the enclave would see bids – perhaps it could output a Merkle commitment to bids for transparency). These details are part of a broader rollup protocol design and are not specific to TEE; however, the key point is that the TEE *ensure*s the chosen policy is enforced, as the operator can't manipulate transactions outside the enclave's logic.

4.6 Protecting Key Confidentiality: If the root secret were exposed, an adversary could decrypt the encrypted payloads of all pending transactions and exploit MEV opportunities or user secrets. Therefore, we treat `root_secret` with the highest level of secrecy. It never leaves enclave memory unencrypted. When stored or transmitted, it's always under encryption tied to enclave identities. Also, note that even if someone later obtains a previously used `Priv_HPKE` key (or the `root_secret` from which it was derived), they cannot retroactively steal value from already executed transactions; however, they may decrypt any mempool ciphertexts still within the `retentionWindow`, potentially revealing historical user intent or sensitive payloads prior to rotation. The main threat is forward-looking: if an enclave's `Priv_HPKE` key or its underlying `root_secret` is compromised while allowed, the encrypted mempool becomes readable to the adversary for the duration of that key's validity window.

In the event of suspected leakage, the protocol should immediately rotate the key. One possible detection method is to notice strange MEV-like patterns that suggest someone can see inside the mempool; although that's not foolproof, it could trigger a precautionary rotation.

To complement encryption, the t1 enclave design might also incorporate *threshold encryption* or *multi-party enclaves* to decentralize the mempool control. For instance, multiple enclaves could each hold a share of the decryption key, requiring a quorum to decrypt the data. However, that moves into complex territory of distributed TEEs and is beyond our initial scope. Our current design assumes a single enclave (or a few in allowed-allowed redundancy) holds the key.

4.7 Summary of Key Provisioning. In summary, each enclave generates or receives a sealed `root_secret` inside a TEE, from which it derives an HPKE key pair used to decrypt incoming transactions. The `root_secret` is transferred only via attested channels and sealed to disk with TDX-based sealing mechanisms when not in use. Periodic key rotation—triggered either on schedule or by governance—ensures forward secrecy. This architecture ensures that an eavesdropper, sequencer host, or even an L1 miner cannot access rollup transactions before execution, thereby greatly reducing exploitable MEV. It also supports privacy-preserving applications: if a transaction contains sensitive data, keeping it encrypted until the moment of execution provides confidentiality similar to a layer-2 shielded poo,though, as designed, all data becomes public once executed.

With the mempool confidentiality handled, we now discuss how the enclaves control *who can execute and upgrade the protocol*, i.e., enclave-authenticated governance and operations.

5. Enclave-Keyed Access Control and Upgrades

In a decentralized protocol, we normally rely on cryptographic signatures from externally owned accounts or multisigs for administrative actions (like upgrading contracts or changing parameters). In t1, certain actions are instead gated by **enclave identities**. This section outlines how enclaves use their own keys to assert privileges and how this mechanism is used for secure upgrades and operation rights within the rollup.

5.1 Enclave Identity Keys: Every Executor enclave in t1 generates an internal asymmetric key pair upon initialization (if not already generated and sealed from a prior run). This key pair (let's call it *Enclave Signing Key*, with private part SK_enc and public part PK_enc) is used to sign important messages, such as rollup block proposals or votes on upgrades. We tie PK_enc to the enclave's identity through remote attestation: specifically, we include a hash of PK_enc in the attestation `REPORT_DATA` when the enclave attests. Thus, the attestation quote essentially states "This enclave with measurement M_e holds a key whose hash is H(PK_enc)". Given that the code inside the enclave is known, we can trust that this key was generated internally, and SK_enc never leaves the enclave. The verifier (via the zk-proof) will obtain H(PK_enc) as a public output. Now, on-chain or off-chain systems can recognize that any signature matching PK_enc (or H(PK_enc)) corresponds to an authentic enclave of the expected code.

In practice, we maintain an *allowlist* of enclave public keys that are currently allowed in the protocol. For example, the rollup's L1 contract can hold a list of approved enclave PKs (or their hashes) along with maybe an expiry or version. When a sequencer posts a new block, the contract verifies that the header is signed by a public key that is marked allowed in NodeRegistry; that allowed status derives from the most recent attestation round. This is a second layer of control: even if an attacker somehow got an attestation from some rogue enclave code (with a different measurement that might slip by if verification criteria were loosened), they wouldn't have an approved PK. Conversely, if we want to remove a node's privileges (say the node operator misbehaves), we could take their enclave's PK off the allowlist (by governance's decision), effectively preventing that node from participating further, even if their enclave is still technically valid. This allowlist is akin to a set of *authorized sequencers*, but authorization is tied to running the correct enclave. It could be used in a decentralized sequencer set or in a permissioned phase of the rollup.

5.2 Signing of Rollup Blocks: When an enclave produces a roll-up block, it signs the batch with SK_enc. The attestation evidence was supplied earlier, at the node's most recent attestation round, and therefore is **not** regenerated per block. This signature covers the block contents and maybe the L1 epoch or other context. Off-chain, this signature is used by other validators or mirrors of the rollup to validate that the block indeed came from an enclave (with identity PK_enc). On-chain verification relies on `ecrecover` of the block signature and a lookup in `NodeRegistry`; no fresh attestation proof accompanies the block.

In any case, the signature is useful for off-chain and audit logs, as well as for any interaction where the enclave needs to authenticate itself beyond the attestation. For example, if the enclave submits a transaction to Ethereum outside the normal channel, it could include a signature that a watchdog can verify against the known PK_enc.

5.3 Secure Protocol Upgrades. Upgrading t1 touches two trust anchors at once: the enclave *measurement* that attestation checks, and the mempool-encryption root secret whose hash is bound into every quote. The process, therefore, walks through five tightly sequenced steps:

1. **Publish the new image.** Developers tag the patched enclave build, reproduce it under deterministic CI, and disclose the resulting measurement *M′*. Auditors verify the hash, the changelog, and that the code will re-derive the root secret exactly as the previous version did.
2. **Governance whitelists *M′*.** A multisig (or token vote, once live) calls the L1 registry to add *M′* to `allowedMeasurements` and bumps `currentProtoVersion`. In the same transaction, it primes the registry to accept a fresh `HashPubHPKE`, signalling that wallets should expect a key rotation as soon as ≥ 66 % of the stake has moved to the new binary.
3. **Dual-running phase.** Operators boot the upgraded TDs. Each new enclave (a) fetches the latest root-secret blob from its predecessor over the attested ECDH channel, or—if migrating to new hardware—requests it from any allowed peer; (b) derives the identical *Pub_HPKE*; and (c) submits an attestation whose `REPORT_DATA` now contains `HashPubHPKE` and *protoVersion + 1*. During this overlap, both generations may co-produce blocks because the registry still treats the old measurement as allowed.
4. **Canonical key update.** When two-thirds of the upgraded TDs have attested, they co-sign `updateHPKE(pub, epoch)`; the contract records the new public key and emits the event that tells wallets to encrypt with it from the next block onward. Legacy enclaves retain the previous private key in RAM for only three epochs by default, allowing them to complete the decryption of transactions already in flight.
5. **Retire the old image.** Governance now removes *M_old* from the allow-list. Any enclave that has not upgraded will have its attestation flagged as "not allowed", and its signatures will be rejected by both the Node Registry and the Canonical Bridge. If the code change altered the state-transition function, the height at which *M′* becomes exclusive is chosen so that every honest node can replay and confirm the hand-off deterministically.

Because each step is authenticated either by the TD's attested key or by an on-chain vote, no operator can sneak in an unreviewed binary; conversely, honest nodes experience zero downtime and never move private material across an untrusted channel.

5.4 Enclave Governance and Admin Actions: Beyond code releases, several levers must change only with clear, cryptographically-verifiable consent from the operators who are actually running attested TDs. Each enclave, therefore, holds a long-lived *governance signing key,* and the Node Registry treats that key as a stake-weighted voter once its attestation is allowed. When a motion arises, the contract opens a *governance slot* (identified by a hash of the action's calldata and deadline). Enclaves that approve the action sign the slot hash and submit `vote(signature)`; the contract tallies compressed BLS signatures until the threshold—two-thirds of bonded stake by default—is reached, at which point it executes the queued call atomically.

Because every signature originates from a TD whose measurement and `HashPubHPKE` were verified in the most recent attestation, the mechanism guarantees that only nodes running the authorised software—and therefore subject to the same slashing and key-rotation logic—can steer the protocol. An operator who shuts down their enclave (or never upgrades) loses their vote automatically; an attacker who merely controls an EOA cannot vote unless they also compromise a TD and re-attest, which the allow-list rules would block. In practice, the system already uses this channel for routine root-secret rotations: the ratchet described in § 4.4 only fires once the `updateHPKE(pub, epoch)` the call itself has gathered the requisite enclave quorum and passed on-chain. The same pattern applies to pausing the bridge, raising `minTCB`, or switching to a new SNARK verifier—providing unified, hardware-backed governance without relying on off-chain coordination.

5.5 Administrator Key within Enclave. It's generally a good practice not to hard-code any secret keys (like an admin key) inside the enclave binary, as that would eventually leak. Instead, any admin or governance keys that need to control the enclave (e.g., tell it to upgrade, or emergency halt) **should** be injected at startup as configuration **that is measured and therefore covered by the enclave's attestation**. Specifically, the binary ships only with the *public* keys of the on-chain governance quorum described in Section 5.4; no private governance material ever resides in the TD.
For instance, if the community multisig wants to pause the roll-up after detecting an exploit, it signs a `PAUSE` command and submits it on-chain. The enclaves retrieve the call, verify the signature against the **embedded governance public key set**, and—if valid—stop processing new transactions and enter a safe mode. This gate ensures that an attacker controlling the host OS cannot inject unauthorised commands, and it keeps the enclave's critical-path secrets limited to its own identity key and the HPKE-derived **root-secret hierarchy**.

5.6 Open Source and Reproducibility Considerations. Because enclaves wield significant power—holding keys, enforcing upgrades, and ratcheting the root-secret-derived HPKE hierarchy—the community must be able to audit every line of code. t1 enclaves are therefore built from fully open-source repositories under a reproducible build pipeline. The key-management logic, encrypted-mempool implementation, root-secret derivation, and admin controls are all visible. Anyone can rebuild the binary, confirm that its hash (and thus the attested measurement and `HashPubHPKE`) matches the published reference, and rule out hidden back-doors or exfiltration paths. This transparency is what makes enclave-centric governance acceptable in a public network context: if an enclave misbehaves, it is either a code bug—addressed by patching and re-attestation—or a TEE breach, which immediately triggers the TCB-recovery measures outlined in Section 8.

5.7 Lifecycle of an Enclave Instance: To tie together sections 2–5, here's an example lifecycle for a single TD node:

- **Launch.** The operator boots the node; the TDX module measures and instantiates the TD. Inside it, the software either generates a fresh identity key pair `PK_enc / SK_enc` or unseals one from a prior run.
- **Initial attestation & registration.** The enclave constructs a quote whose `REPORT_DATA = SHA-256(nonce ∥ PK_enc ∥ currentProtoVersion ∥ HashPubHPKE)` and feeds that quote into the zk-attestation prover. The resulting proof is submitted to the L1 `NodeRegistry`, which records `PK_enc`, `measurementHash`, *and* `HashPubHPKE` as allowed once the proof is verified.
- **Root-secret provisioning.** If the TD was restarted locally, it simply unseals its previously stored 32-byte `root_secret`; otherwise, it opens an attested channel to a live peer, proves its measurement, and receives an encrypted transfer of `root_secret`. From that secret it deterministically derives `(Pub_HPKE, Priv_HPKE)` and immediately seals the secret back to disk using a Gramine-style TDX sealing key tied to the enclave signer.
- **Normal operation.** The enclave accepts ciphertexts $\text{ciphertext}_1 \parallel \text{ciphertext}_2$ from users, unwraps the session key with `Priv_HPKE`, decrypts the payload, executes the transaction batch, and signs the resulting L2 block with `SK_enc`. Only the lightweight block signature is posted on each L1 commit; no new quote is required.
- **Upgrade.** When governance approves a new binary, the operator halts the TD, spawns a fresh TD running the new measurement, and transfers `root_secret` (plus any other state) through the same attested channel. The new TD attests, publishes its proof, and becomes primary; the old TD retires once its final duties are complete.
- **Shutdown.** To exit, the enclave may send a signed *Deactivate* call to `NodeRegistry`, wipe `root_secret` and other secrets from RAM, and optionally destroy its sealed blobs. If future restarts are planned, it can instead keep the sealed files; TDX scrubs memory on power-off in either case.

Throughout this lifecycle, the *root_secret* never appears outside an attested channel or sealed blob, and every block's provenance can be traced via `PK_enc` and `HashPubHPKE`—back to the validator's most recent, on-chain-verified attestation.

At this juncture, we have a complete picture of how t1 uses TEEs for execution, attestation, mempool privacy, and control. We now address some advanced topics and concerns that have been raised in peer discussions to clarify our security posture and the limits of this approach.

6. Secure Cross-Chain interactions

ERC-7683 Intents serve here as a practical, near-term example of secure cross-chain transactions, providing a standard way to express and fulfill user requests across different blockchains. This is not unique to rollups or L2 networks—intents can bridge *any* disparate chains, from Ethereum mainnet to sidechains, rollups, or appchains. Under the ERC-7683 model, a user's desired outcome is encapsulated in an i**ntent** message (conforming to ERC-7683) that specifies *what* the user wants to achieve on a destination chain, without prescribing *how* to achieve it. This abstraction enables a seamless cross-chain experience by decoupling the user's intent from the underlying execution path.

Why single out intents? They are simply a convenient, standards-based subset of cross-chain calls that exists today. The very same TEE-observer pattern that we shall present generalizes to any verifiable x-chain transaction—asset transfers, oracle reads, governance votes—so long as the enclave co-locates a full node that can fetch state proofs.

6.1 Execution on the Destination Chain. When a cross-chain intent is opened, the user locks or escrows their assets on the source (origin) chain in a **Settlement Contract**. A third-party **Filler** then fulfills the intent on the **destination chain** by executing the required transactions there using their own liquidity. Note that **all intent execution occurs on-chain on the destination network**, not off-chain. The secure off-chain component in our architecture is limited to *verification*: a trusted domain observer confirms that the intent's fill **actually occurred on the destination chain** as specified. In other words, the trust domain's role is to validate the outcome (e.g. that the user received the expected funds or tokens on the destination chain) by checking on-chain state, **not** to perform the swap or action itself. The integrity of this verification step is ensured by running it within a secure trust domain, which guarantees that the observation cannot be tampered with and adheres to the intended logic.

6.2 Enclave Verification and Escrow Release. Once the trust domain confirms that the intent was filled on the destination chain (for example, detecting that the designated swap or contract call succeeded and the desired outcome was delivered), it produces an attestation of this event. This attestation is a signed, tamper-proof claim that *"Intent X was fulfilled on Chain B at block Y with result Z."* The trust domain appends this attestation to a Merkle trie on the origin chain's settlement contract, allowing anyone to post a Merkle inclusion proof to be verified against the trie, and if valid, releases the escrowed funds on the origin chain to the appropriate party. In a typical scenario, the filler who executed the intent on the destination chain is paid out from the user's deposit on the origin chain as a reward for their service. By structuring the payment as a release of the user's escrow, the system remains generalized – any origin chain can act as the escrow venue, not just a specific platform. The **Settlement System** on the origin chain holds the user's funds, verifies the fulfillment via the trust domain's attestation, and then unlocks the funds to compensate the filler or complete the intent's settlement. This general design makes the payment logic flexible: the origin chain contract handles payouts, allowing the model to be applied to various networks and avoiding hardcoding to a particular bridge or rollup.

6.3 Trust Domain of Observers. Our implementation employs secure trust domains as **rollup observers** (or more generally, cross-chain state observers) that monitor the destination chains for intent fulfillment. For simplicity and security, we chose to run all observer enclaves within a single trust domain – meaning they share a common attestation identity and trust root. This design choice enables a unified trust assumption for all chain observers and streamlines the management of enclave keys. **However, this is not a strict requirement of the architecture.** It is an implementation decision rather than a protocol mandate. Alternate deployments could isolate observers per chain or use multiple independent trust domains, as long as each trust domain's attestation can be verified on the origin chain. The ERC-7683 intent standard itself does not dictate how many trust domains must be used; it only requires that the mechanism observing and verifying the fill be trusted by the settlement contract. In summary, all observers *may* run under one trust domain for convenience; however, it is equally possible to have a distributed set of verifiers with separate trust roots, provided they meet the same security and authenticity criteria.

6.4 Deterministic Receipt Commitments. Each trust domain verification produces a **commitment** to the on-chain evidence of the intent's fulfillment. Specifically, the trust domain will perform a deterministic state query against the destination chain to gather proof of the fill. The trust domain then computes a hash over the full context and result of this query, including:

- the target contract address queried on the destination chain,
- the function signature (or specific event signature) and any input parameters used for the query,
- the block number (and/or block hash) at which the query was executed, and
- the resulting output or receipt data returned by that query.

This hashed commitment uniquely and deterministically represents the outcome of the intent at that point in the destination chain's state. Because the commitment covers **all relevant inputs and outputs of the query**, any external verifier with access to the destination chain can replicate the exact same query (using the same contract, function, parameters, and block number) and should obtain the identical result. They can then hash those results in the same way to confirm that it matches the enclave's reported commitment. In this manner, the enclave's cryptographic commitment to a block's receipts or state ensures **consistency and auditability**: if two independent parties check the same on-chain facts, they will derive the same commitment hash. This property enables transparency and trust, as anyone can verify that the enclave did not forge or alter the fill evidence – the attested result is objectively anchored to the destination chain's publicly verifiable state.

On-Chain Settlement Workflow: The origin-chain settlement contract is designed to securely accept the enclave's attestation and finalize the fulfillment of intent. The workflow is as follows:

1. **Attestation Submission:** The enclave (operating off-chain in the trusted environment) submits its signed attestation to the origin chain's settlement contract. This attestation contains the enclave's identity (e.g., an enclave public key or certificate), the hash of the query result, and the associated commitment data as described above.
2. **Enclave Verification:** The settlement contract checks the signature and identity against a registry of **recognized trust domains**. Each authorized domain is mapped to a known signing account or public key. The contract **verifies that the attestation's signer matches a valid enclave** and rejects any submission from unknown or untrusted sources. This ensures that only attested observations from approved secure enclaves can trigger a settlement.
3. **Recording Results in Merkle Trie:** Upon successful verification of the enclave signature, the contract records the reported result in an append-only log structure on-chain. In particular, the result (or its hash) is inserted as a new leaf in a **Merkle trie** maintained by the settlement contract. Each attested fill result thus becomes a part of an ever-growing Merkle tree of fulfilled intents. After insertion, the contract updates the stored **Merkle root** to reflect the new leaf.
4. **Public Root and Inclusion Proofs:** The updated Merkle root is exposed by the contract (e.g. stored in contract state or emitted in an event), allowing anyone external to obtain it. Because each file's data is now represented in the Merkle tree, any party can later prove the inclusion of a specific intent fulfillment by presenting a standard Merkle proof (a sequence of sibling hashes) against the publicly available root. In essence, the contract's Merkle trie serves as an auditable ledger of all verified cross-chain fills, allowing external auditors or other smart contracts to verify that a particular cross-chain outcome was indeed confirmed and recorded.

With this settlement process, the origin chain smart contract not only releases funds but also creates a tamper-evident record of the cross-chain intent fulfillment. The enclaves output the hash of the query results (plus the contextual commitment) as a signed package, and the on-chain logic ties this into an immutable log (via the Merkle root). This design makes the fulfillment hardware **verifiable** and **easily auditable**, allowing observers to independently confirm the destination chain outcome and its proper settlement on the origin chain. Overall, this flow underpins all cross-chain transactions we expect t1 to support: ERC-7683 intents today and the broader class of verifiable x-chain calls we foresee as new standards emerge.

Having covered cross-chain operations, we now turn to a broader discussion on security considerations, design rationale, and addressing potential academic critiques in Section 7.

7. Security Discussion and Design Refinements

In developing t1's TEE-based architecture, we carefully considered the feedback and concerns from the security research community. This section revisits some nuanced issues: the distinction between hardware attestation and full cryptographic verification, the exact guarantees of TDX's isolation (and misuse of terminology), the role of nonces in attestation, and the need for reproducible builds and transparency. We also address how our design balances trust and decentralization, acknowledging the inherent reliance on Intel and how we mitigate that.

7.1 Attestation vs. Verifiable Computation: A common point of confusion is the phrase "cryptographically verified execution" in the context of TEEs. It's crucial to clarify that **remote attestation is not equivalent to a zkSNARK proof of execution**. Remote attestation provides a *cryptographic identity check* proving that the output or message originated from a piece of software with a particular hash, running in a secure enclave on genuine hardware. It does *not* prove that the software's output is correct relative to some specification – we trust the software to be correct. In contrast, a system like a zk-rollup generates a proof that the new state is mathematically consistent with the old state and the transactions, independent of who computed it. In t1, we combine these approaches: attestation gives a faster but somewhat trust-based assurance (trusting the hardware and code), while eventually, a ZKP checkpoint, via on-demand or periodic ZKP, can give a trustless mathematical guarantee. By including both, we aim for the best of both worlds: immediate liveness and decent security from TEEs, with eventual correctness proofs from zk-provers as a backstop.

Insurance-Fund Guard-Rail.

To cap the economic risk that accrues between zk-proof checkpoints, governance escrows an on-chain **insurance fund**.

- **Insurance assumption:** the fund is sized to cover the worst-case loss if TEEs were to fail.
- **Cumulative-risk accounting:** every block header reports `epochValue` (value moved in that block) and `cumValueSinceZKP` (running total since the last zk checkpoint). Validators must pause value-bearing blocks—and require a zk-SNARK proof that resets `cumValueSinceZKP` to 0—before the running total can exceed the fund.
- **Further specification:** the detailed logic will be described in a follow-up document.

We refrain from stating that the enclaves provide "verifiable computation" in the strict sense. Instead, we say they provide *"verified environment execution."* The environment (the enclave identity) is verified, which implies (given we trust the code) that the computation was as intended. There is a subtle but important difference: if the code has a bug or the hardware is compromised, the enclave could produce a wrong result and still attest correctly (the attestation doesn't catch logical errors or malicious logic). This is why open source and auditing of the enclave code is vital, and why a fallback like fraud proofs or zk proofs can catch such errors. In academic terms, our approach is an instance of a hybrid model: not purely cryptographic truth, but hardware-rooted truth. It's akin to a Byzantine-fault-tolerant consensus protocol where the honest-majority assumption is replaced by an honest-hardware (or honest-majority-of-hardware) assumption for the short term.

7.2 "Cryptographic Isolation" vs. Hardware Isolation: In earlier drafts, we (and others) sometimes used the term *cryptographic isolation* to describe how enclaves protect data. To be precise, enclaves like TDX use encryption and integrity checks (which are cryptographic mechanisms) to isolate memory, but the overall isolation is enforced by hardware logic (the CPU will not execute certain instructions outside SEAM, etc.). It's not that there's a cryptographic proof exchanged at runtime between CPU and OS – rather, the CPU is built to refuse access. The only cryptographic element visible externally is the attestation signature and the encrypted memory. So, a more accurate description is **hardware-enforced isolation with cryptographic memory protection**. We have adjusted our terminology accordingly. Where the litepaper might have said "cryptographically isolated Executor", we now say "TDX-isolated Executor" or simply emphasize the hardware trust domain. This prevents readers from assuming we mean something like MPC or threshold cryptography, isolating the computation.

7.3 Role of `REPORT_DATA` Nonces in Attestation: We have implemented robust nonce usage in our RA protocol, as described in Section 3. The enclave's attestation code will always expect a 64-byte random challenge from the verifier (or derive one from context) and place a hash of it into the report before quoting. The importance of this cannot be overstated: without it, an attacker could record a quote from an enclave and later replay it to deceive the verifier into believing that a fresh operation had occurred. Intel SGX/TDX attestation itself does not include an implicit nonce – it relies on protocols to do so. The literature is clear on this point, and we abide by best practices. The attestation zk-circuit explicitly takes the expected nonce as an input and verifies it matches the quote's `report_data`. Moreover, we tie the nonce to the specific attestation round (node onboarding, protocol-version upgrade, or TCB update) so that even if someone replayed an old quote with the same nonce, the surrounding context would differ and the proof would be rejected. This provides **replay protection** for attestation.

7.4 Binary Measurements and Reproducible Builds: Because attestation works at the binary level (measuring the exact bytes loaded into the enclave), it raises the issue of how stakeholders can trust that those bytes correspond to the audited source code. If the project were closed-source, one would have to blindly trust the team about the measurement. However, t1's enclave code is fully open-source, and we commit to enabling **reproducible builds**. Reproducible (deterministic) builds mean that anyone can compile the source (with a specified compiler version, flags, etc.) and obtain the exact same binary bits and thus the same hash. This is non-trivial to achieve, especially for complex projects, but tools like Nix, Bazel, or Guix can help pin environments. As noted by *Trail of Bits* researchers, reproducible builds "complete the trust chain" for SGX enclaves; we apply the same principle for TDX enclaves. Our build pipeline uses a containerized environment with fixed dependencies to ensure deterministic output. We also provide the calculated measurement (hash) in the documentation for each release. During attestation verification, the expected measurement is checked; thus, if someone tampers with the code or compiles it differently, it won't match, and the attestation will fail. Community members are encouraged to rebuild the enclave from source to verify the official measurement. This approach is similar to how Etherscan verifies on-chain bytecode by recompiling Solidity – here we do it for off-chain enclave binaries. Reproducibility and open source together enable **open auditability** of the enclave. There should be no hidden code running that has not been reviewed.

7.5 Open-Source Requirement for Security: We want to stress that *the security of the t1 enclave model is only as good as the scrutiny it receives*. We choose Intel TDX for hardware security, but the software we run inside must be bug-free (to a high degree) and free of malicious logic. By open-sourcing it, we allow the community to inspect everything, from how keys are handled to how state transition logic is implemented. This contrasts with proprietary enclaves in some systems where users must trust the company's attestation blindly. In fact, one can argue that in a decentralized context, closed-source enclaves are unacceptable because they could attest to being "some allowed code" but that code might be doing something engage in malicious or undisclosed behaviour that isn't apparent. By open-sourcing and reproducibly building, we ensure that "allowed code" is exactly the code everyone agreed on.

7.6 Trust in Intel and Single-Source Dependency: Perhaps the most significant overarching concern is our reliance on Intel (or any single hardware vendor). The **trust model** of t1 includes trusting that Intel's implementation of TDX is secure and that Intel (and its manufacturing process) did not introduce backdoors. This is a *centralized trust assumption,* which blockchain purists rightly point out as a weakness. If Intel's secret key for signing attestation certificates were compromised, an attacker could potentially fake enclave quotes (though Intel would likely revoke and replace keys in that scenario). If a microcode bug allows an escape from the enclave or the extraction of secrets, the entire integrity and confidentiality could be compromised. These are not hypothetical – SGX saw numerous side-channel attacks and a few instances of key leakage (like the infamous SGX private key leak via *SGAxe*). Thus, we confront the reality: **TEEs are not invulnerable**, and their failure is a single point of failure for all relying relying on them.

We address this in several ways:

- **Multi-layer security:** As mentioned, a Remote Attestation proof can catch erroneous state updates even if an enclave acts maliciously *provided the code isn't colluding with the attacker*. In other words, if the hardware is compromised to just output a wrong state, a zk proof (if it recomputes the state) would not accept it. However, if the hardware is compromised in a way that it can also fake the zk proof or if we haven't integrated zk proofs yet, then that layer isn't there. But having the design slot for it means eventually we rely less on the TEE for correctness, using it mainly for liveness and data availability.
- **TCB recovery (see Section 8):** If Intel announces a vulnerability, we have a protocol in place to *recover trust*, which usually means patching the microcode or software and then re-attesting so everyone knows the new version is secure. In extreme cases, if TEEs are broken, the protocol could switch to an alternative security mode (like purely optimistic with fraud proofs, or halting until zk-proofs are ready).
- **Diversity of TEEs:** While currently we use Intel TDX, in the future the protocol could support multiple TEE implementations in parallel (e.g., AMD SEV-SNP, or ARM's Realms, or even different vendors like AWS Nitro enclaves). If we had two different TEEs, one could require attestation from both – this way it's less likely both get broken at the same time. This is complex but not impossible: it means writing our software to target multiple enclave types and then requiring a threshold of them to sign off on results (some sort of multi-TEE consensus). This was hinted at in multi-prover approaches.
- **Decentralization vs Trust Trade-off:** We openly acknowledge that introducing TEEs is a trade-off: we gain speed and privacy, but we introduce a trust dependency on hardware vendors and the security of complex CPUs. We mitigate it, and we believe the trade-off is worthwhile with safeguards, but it must be *evaluated continuously*. If at any point the risk outweighs the benefit, the community must be ready to fall back to a safer (if slower) fully trustless mode.
- **Transparency about TEE status:** We will maintain a transparent policy of tracking known TEE vulnerabilities and their impact on t1. If an issue like a new side-channel is discovered that affects our enclave's ability to keep secrets, we may temporarily disable certain features (like the partially encrypted mempool) until a patch or mitigation is in place. The system's design is flexible enough to degrade gracefully if confidentiality is lost (it would be unfortunate for MEV, but it wouldn't immediately break correctness). If integrity is lost (i.e., enclaves can be forged), that is more severe, likely leading to a fallback to the zero-knowledge proof-based protocol.

7.7 Side Channels and "Cryptographic Isolation" Limits: A subtle point: TEEs like SGX/TDX do not inherently protect against all side-channel leaks, especially timing or cache-based. Many SGX attacks exploited the shared CPU cache or branch predictors to infer enclave data. Intel TDX, by virtue of isolating a full VM, might be able to mitigate some of these by not sharing cores between TD and other VMs (one can pin the TD to cores, etc.). But some leakage might still occur (e.g., power analysis if one had physical access, or sophisticated memory access pattern analysis). In our context,
side-channel leakage could potentially reveal the `root_secret` or derived HPKE private key (`Priv_HPKE`) used to decrypt pending transactions, if the adversary can run spy processes on the same host. We assume our threat model does not include physical attacks (we assume the hosting environment is secure from physical tampering, focusing on remote software attacks). We also assume that major side channels will be patched by Intel when discovered (with microcode or guidance like disabling hyperthreading). We can incorporate best practices, such as ensuring the enclave VM is scheduled in a way that minimizes sharing with untrusted processes and using constant-time cryptography in enclave code. Nonetheless, it's an area where "isolation" is not absolute. We use the term **"isolated"** to mean logically and memory isolated, but not necessarily side-channel isolated. The phrase "cryptographic isolation" might mislead one to think that the isolation is as perfect as encryption, where nothing leaks; in reality, some information (such as access patterns) could still leak. Thus, we precisely define isolation as follows: *the enclave's memory contents and CPU state cannot be directly read or written by any other software; any attempt to do so would yield only encrypted data*. This is true, but indirect information could still be gleaned.

7.8 Reproducibility and Binary Transparency in Practice: A challenge with reproducible builds is making sure the *exact* environment can be replicated by third parties. We might use Nix with pinned dependencies, publishing a Docker image with a specific hash that builds the enclave. If multiple independent parties run the same measurement and obtain the same result, confidence increases that the binary is indeed from the published source. *Signal* and *MobileCoin* have done similar for their SGX enclaves. We'll also explore using attestation in the build process itself (for example, building inside an enclave to attest that the build was done on a clean system). That starts to overlap with concepts like "trustworthy build systems" but is beyond scope.

7.9 Summarizing the Trust Model: At the end of the day, the security of t1's TEE approach rests on:

- Trusting Intel (or any vendor providing the TEE) not to have included a hardware-level proprietary backdoor and to make best efforts to secure the hardware.
- Trusting our enclave code to be correct and secure (which we bolster via open audit and formal methods if possible).
- Trusting that ZKPs are mathematically sound and complete, that is, that the underlying algorithms are bug free.
- Accepting that in the interim between execution and zk-proof, the enclave results are *probabilistically* secure (based on these trusts) rather than *cryptographically guaranteed* in the absolute sense.
- Relying on on-chain enforcement (zk verification of attestation) to remove the need to trust any off-chain party's word about the enclave status.

This is a trusted-hardware assumption combined with a cryptographic proof for authentication and—potentially—periodic zero-knowledge proofs of computation. This is strictly weaker than a pure zk-rollup in security, but in practice stronger than an optimistic rollup on the dimensions of bounded risk and time-to-finality, especially due to the periodic ZKP checkpoints that cap interim exposure and remove challenge-period latency. We believe this hybrid can be practical and secure for many use-cases, at least until other advances allow comparable performance with full trustlessness.

With these considerations addressed, we proceed to Section 8, which outlines the steps taken when issues arise. Specifically, we outline how we recover from TEE vulnerabilities or need to patch the system.

8. TCB Recovery and Security Patching

No security system is static; over time, vulnerabilities may be discovered, and new attack vectors unveiled. In a blockchain context, responding to such events in a coordinated and secure manner is crucial to maintain user trust and asset safety. This section outlines how t1 handles **Trusted Computing Base (TCB) recovery**, i.e., updating and re-establishing trust in the TEE platform after a vulnerability, as well as how we manage key material during such events.

8.1 Understanding TCB and Vulnerabilities: For our purposes, the TCB (Trusted Computing Base) includes the CPU hardware, microcode/firmware, the TDX module, and the enclave software itself. A TEE vulnerability could exist at any of those layers:

- *Hardware/Microcode:* e.g., a CPU bug that allows an attack to extract enclave keys or bypass memory encryption.
- *TDX Module/Security Version Number (SVN):* if a flaw is found in the TDX implementation, Intel may issue a microcode update that raises the TDX SVN and fixes it.
- *Enclave Software:* a bug in our code that could be exploited (like a buffer overflow or a logical bug that allows someone to inject a fake attestation).

When such a vulnerability is discovered, especially in hardware or TDX, Intel will perform a **TCB recovery procedure**. This usually means:

- Releasing patches (microcode updates or new versions of the quoting enclave, etc.) that fix the vulnerability.
- Possibly revoking old attestation keys or certifying new ones if the old ones are considered compromised.

Intel's attestation infrastructure has a notion of TCB levels (as seen in the quote data and collateral). Verifiers can set a policy to reject attestations below a certain TCB level once an issue is public. For example, after a vulnerability disclosure, Intel's PCS (Provisioning Certification Service) might start marking quotes from unpatched platforms as "OutOfDate" in the TCB status, and eventually add the affected CPU's keys to a revocation list. Our attestation verification circuit will catch this (the quote verification includes checking that the `TDX TCB status` is OK). So in many cases, just running the verification as usual will start failing for unpatched enclaves after a vulnerability is announced.

8.2 Detection and Response: Suppose a vulnerability in TDX is announced. Immediately, the following happens:

- **Pause Accepting Attestations (if needed):** If the issue is severe (e.g., enclave secrets can be stolen), the safest immediate action might be for the L1 contract (or off-chain monitors) to stop accepting new rollup state updates from enclaves until we patch. This could be automated if the nature of the vulnerability is such that the quote verification would fail anyway (e.g., Intel revokes the old TCB, so the proofs would not verify). If not automated, the governance multisig may trigger an emergency brake (this is where having an admin capability to pause the contract via a privileged action is useful, though that introduces a point of centralized control – ideally, it's a multisig with community oversight).
- **Notify Node Operators:** All operators should update their systems. Intel may provide a microcode update; operators apply it (which may require rebooting the machine to load new microcode, or using BIOS updates).
- **Enclave Software Update:** If the vulnerability requires changes to enclave code (for example, to add a mitigation or rotate keys), we release a patch for the enclave code as well, with a new measurement M_new. This process involves the reproducible build.
- **Re-Attestation:** After patching, each node's enclave will have either a new measurement (if the software changed) or at least a new TCB SVN (due to microcode update). They generate new quotes. The L1 contract or verifying parties will now see quotes with the updated status. We likely need to update our allowlist to include the new measurement (for software changes), via a governance transaction.

This process essentially mirrors how cloud providers handle SGX/TDX updates but in a decentralized way. The *attestation evidence itself is used to prove that patching took place* – because the quote now shows an updated TCB version, which is cryptographically signed by the hardware proving the update was applied.

8.3 Rolling Back or Disabling Features: If a vulnerability specifically compromises confidentiality but not integrity (say a side-channel that leaks mempool contents), we might choose to temporarily disable the encrypted mempool feature rather than halt the whole rollup. The enclave could switch to publishing transactions in plaintext (losing MEV protection but maintaining liveness) until a fix is out. If integrity is compromised (attackers can fake enclaves), that's more severe as mentioned – likely requiring at least a short halt or moving to an alternative security model (perhaps manual validation of each block by a committee until TEEs are safe again). These contingency modes should be coded into the governance procedures.

8.4 Attestation Key Changes: In some cases, a TCB recovery involves changing the attestation keys (for instance, the quoting enclave's keys). Intel's whitepaper on SGX TCB recovery mentions that a new attestation key (and certificate) may be issued, and old ones invalidated. For us, this would mean our attestation verification code might need to be updated with collateral (new root CA or CRL). We designed our zkVM verification to be upgradable – the verifying program can accept updated Intel root certificates as input if needed. The on-chain verifier contract may also need an update if the SNARK's verification key changes (though ideally, we can avoid having to change the SNARK circuit – if the Intel root key changes, that could require a circuit update because the public key is hardcoded unless we allow it as input). We plan for some flexibility: e.g., support multiple root keys (the circuit can be configured to accept either the old or new Intel root by checking against both). In any event, such an update would be coordinated with an enclave software update in a hard fork-like maneuver for the rollup.

8.5 Sealed Data and Key Rotation: When upgrading enclaves (either routine or emergency), preserving or rotating the keys that were in the old enclave is delicate. Specifically:

- **Root-secret & HPKE hierarchy.** If analysis shows the 32-byte `root_secret` remained confidential, the retiring TD transfers it over the usual attested channel to the new TD, which then re-derives the same `(Pub_HPKE, Priv_HPKE)` pair. If the secret might have leaked (e.g., via a side-channel), the new enclave discards it, samples a fresh `root_secret'`, derives a new HPKE key-pair, and co-signs an on-chain `updateHPKE()` vote. Wallets read the event and immediately switch to the published `Pub_HPKE'`; Executors accept both old and new keys for `retentionWindow` epochs, then erase the compromised key to prevent retrospective decryption.
- **Enclave Identity Keys (PK_enc):** If we launch entirely new enclaves with new code, they'll have new identity keys. The allowlist mechanism on-chain needs updating to include the new PKs. If the set of operators is the same, one strategy is to derive the new PK from the old one in a way that can be verified. For example, the old enclave could sign a statement "I am going to instantiate a new enclave with measurement M_new and here is its new PK_new" – this is done right before upgrade, and then on-chain or off-chain validators can link the old and new identity (so that the operator's reputation or stake carries over). If that's too complex, just treat it as deploying new nodes and removing old ones.
- **Sealed State:** If the enclave had any other persistent state, such as cached cross-chain headers or other caches, those either can be discarded or exported. Generally, only `root_secret` (or its ratcheted successor) and the identity key deserves long-term sealing;

everything else is expendable or reproducible during boot-strap. We design the state such that an enclave can sync from scratch if needed (for example on startup, it can fetch necessary data from the L1 contract and from its follower nodes).

8.6 Testing Patches via Attestation: One advantage of attestation is we can even pre-test our network's resilience. For instance, we could simulate a TDX vulnerability by temporarily telling our verifier to reject a certain SVN (like mimicking what would happen) and see if nodes respond correctly (they should update and re-attest). We can run periodic attestation drills, e.g., require nodes to attest at least every X days even if they aren't producing blocks, to ensure their quotes are fresh and not stuck on an old microcode. Rather than requiring periodic re-attestation on a fixed schedule, we may enforce it *on-demand*—specifically, by requiring a validator to present a fresh attestation before it is permitted to post a new state root to L1. This policy ensures that enclave quotes are up-to-date at the exact moment they affect consensus, while avoiding unnecessary re-attestation from idle or standby nodes. This reduces the window of vulnerability if someone tries to conceal that they haven't patched.

8.7 Governance and Social Layer: Ultimately, handling major security events will involve human coordination and governance. While we automate what we can (via attestation checks and allowlist updates), something like deciding to halt the system or switch to a different mode may require multisig or community vote.... For transparency, we include the community in these decisions: e.g., if a pause is needed, a governance vote could be held if time permits, or a multisig of core contributors acts with the understanding that an ex-post-facto review by the community will happen.

8.8 Key Takeaways for TCB Recovery: The ability to *prove an update* via attestation is a powerful tool. Rather than relying solely on trust that operators updated their software, we get cryptographic evidence in the next quote. t1 leverages this: the moment a patch is applied, the new attestation shows a new SVN or measurement, which the contract can be set to accept while rejecting old ones. The design ensures that there is no silent failure: either an enclave meets the required TCB level and is allowed, or it doesn't and is rejected. This motivates operators to promptly apply critical patches or they will be unable to participate.

To conclude this section, t1's security strategy is not static; it is an evolving process that involves monitoring (for vulnerabilities), rapid response (through attestation enforcement and software updates), and community governance (for making hard calls like pausing or changing configurations). While we hope not to need these emergency measures often, they are specified and prepared for use. The ultimate goal is to maintain the integrity of the rollup even under adverse conditions, and to do so in a way that users and stakeholders can verify and trust through the same attestation and transparency mechanisms we've built.

9. Conclusion

This document presented a comprehensive design for integrating Trusted Execution Environments, specifically Intel TDX enclaves, into the t1 rollup protocol's architecture. We detailed how Executor Trust Domains provide a shielded environment for transaction processing and how remote attestation is employed to link the trust in those environments to on-chain verification via Remote Attestation proof. By doing so, t1 achieves **real-time finality** of rollup blocks on Ethereum with strong assurances: every state update accepted on L1 is backed by a hardware-rooted proof of correct execution.

Our design goes beyond a simple application of TEEs; it introduces a holistic system covering key management (partially encrypted mempool for MEV mitigation), cross-chain interoperability (enclaves as secure oracles for state reads and transaction relays), and a robust lifecycle for multiple enclaves (from deployment, through upgrades, to recovery from vulnerabilities). We have demonstrated how the traditionally opaque nature of TEEs can be ameliorated by open-source software development, reproducible builds, and on-chain attestation verification, thereby aligning with the transparency ethos of blockchain systems.

One of the core contributions of this work is the novel *attestation verification pipeline* which utilizes zero-knowledge proofs to compress and decentralize the verification of enclave quotes—so that new enclaves may contribute to real-time proving t1's state towards L1. This pipeline exemplifies how advanced cryptography (zkSNARKs) and hardware security can complement each other: hardware provides speed and integrity of computation, while cryptography provides public verifiability and enforcement. The result is a rollup that neither relies on optimistic assumptions with delayed fraud detection and challenge-based re-execution, nor incurs the overhead of generating expensive validity proofs for every block, yet also avoids blind trust in a single operator. Instead, trust is anchored in widely distributed silicon (Intel CPUs in many data centers) and can be monitored by anyone via on-chain proofs.

We incorporated multiple layers of fallback not by abandoning the TEE trust model, but by enforcing cryptographic and policy-driven controls around attestation and validator participation. In particular, t1 ensures that only nodes with allowed, governance-approved attestations are permitted to sign rollup state updates. If vulnerabilities arise—whether in specific enclave builds or broader hardware classes—those nodes can be revoked or suspended immediately, without halting the system. This model allows t1 to remain operational even under partial TEE compromise, while retaining the option to escalate to stronger verification guarantees, such as on-demand zk-proofs, when warranted.

From a performance and functionality standpoint, the described architecture enables t1 to offer unique features: instant t1 block finality on L1 (subject to attestation validation) and an MEV-resistant execution layer (through encrypted mempool and fair ordering in enclaves). Moreover, it allows cross-rollup composability by enabling secure cross-chain reads and writes. A cross-chain yield optimization dApp on t1 could, for example, read yield across different lending protocol(s) deployed on different rollups within an enclave and determine where to move t1 user funds—all while programmatically using authentic data only. This kind of fast cross-domain interaction is a stepping stone to unifying liquidity and state across rollups, which was one of t1's motivations.

The document has also outlined a blueprint for how to manage the enclave network operationally: how new nodes join (attest to get keys), how upgrades happen (with enclave cooperation and governance oversight), and how the system responds to the discovery of vulnerabilities (through attestation-driven TCB recovery). These operational guidelines ensure that the integration of TEEs does not become a maintenance nightmare or a central point of stagnation.

In conclusion, t1's TEE architecture aims to demonstrate that integrity and confidentiality provided by hardware can be combined with the auditability and trustlessness of Ethereum in a synergistic way. We transform traditionally private attestation statements into **transparent, on-chain verifications**, thus removing hidden trust relationships and replacing them with explicit, cryptographically verifiable ones. The approach can be seen as an intermediate on the spectrum of layer 2 designs – more decentralized and secure than a permissioned sidechain or state channel, but more performant (in latency terms) than fully zk-verified rollups.

We believe this design will be of interest not only to Ethereum researchers and rollup builders looking for practical scalability solutions, but also to the broader systems security community as a case study of applying TEEs in a decentralized context. By sharing this detailed specification, we invite peer review and collaboration. There are open questions and potential extensions: for example, exploring multi-TEE consensus, using formal verification for enclave code, or integrating post-quantum attestation signatures. The foundation laid here can serve as a basis for such future work.

Ultimately, the success of t1's approach will be measured by its security in production and the value it provides to the Ethereum ecosystem. We have aimed to cover all critical aspects around using TEEs towards this goal in this paper. As t1 progresses from design to implementation and deployment, this document will serve as the original reference for the community about how its TEE components function and how they will be governed. We look forward to realizing the potential of real-time, TEE-secured rollups, and will continue to update and refine this architecture in partnership with the community.

Acknowledgments

We are grateful for the insightful feedback and constructive critique provided by the following reviewers, whose comments have helped clarify and strengthen this whitepaper:

- **Jo Van Bulck** – DistriNet lab, Department of Computer Science, KU Leuven
- **Pedro Antonino** – The Blockhouse Technology Limited
- **Qian Ren** – Morphic Lab

Glossary

- **Trusted Execution Environment (TEE):** A secure area of a processor that guarantees code and data loaded within to be protected with respect to integrity and confidentiality. In a TEE, even a privileged OS cannot access the protected memory. Examples include Intel SGX, Intel TDX, AMD SEV-SNP, and ARM TrustZone. TEEs enable *remote attestation* to prove what code is running inside.
- **Enclave:** Common term for a protected container in a TEE. Originally used in context of Intel SGX to refer to a secure process enclave. In this paper, we sometimes use "enclave" generally to mean a TEE-protected execution context. Specifically, **SGX enclave** refers to a process-level enclave as in SGX, whereas **TDX enclave** (or **Trust Domain**) refers to a VM-level enclave in TDX. An enclave has a measured identity (like MRENCLAVE or similar hash) and can perform secure computations.
- **Trust Domain (TD):** In Intel TDX, a Trust Domain is an entire virtual machine that is cryptographically isolated from the host/hypervisor. It's analogous to an enclave but encompasses a full OS and application stack. Memory of a TD is encrypted and integrity-protected with a per-TD key. We use "Executor Trust Domain" to mean the TDX VM running the rollup node software.
- **Intel TDX (Trust Domain Extensions):** Intel's technology for confidential VMs. It creates Trust Domains on capable CPUs, isolating them via a special CPU mode (SEAM) and using a TDX module. TDX uses SGX technology internally for attestation (via TD Quoting Enclave). It aims to overcome SGX limitations by allowing legacy OS and bigger memory inside the TEE.
- **Remote Attestation (RA):** A process by which a TEE produces a signed statement (attestation) about what code is running and the security state of the platform. It usually involves a hardware-held key signing a hash of the enclave/TD's contents. The result is an attestation **Quote**, which can be verified by a remote party using the vendor's public keys. Attestation assures the remote party that they are interacting with a genuine enclave running expected code. In our context, attestation is used to convince Ethereum smart contracts of the enclave's identity.
- **Quote (Attestation Quote):** The data blob produced by the quoting enclave (QE/TDQE) that contains the enclave's measurement, TCB status, custom `report_data`, and a signature from the hardware's attestation key. It is the token of proof for attestation. A quote is accompanied by collateral like certs and revocation info so that a verifier can validate it fully.
- **REPORT_DATA:** A field in the Intel attestation report into which user-provided data is inserted before quoting. Typically used to include a nonce or other context, to prevent replay. It's 64 bytes (for SGX/TDX) and can carry arbitrary info chosen by the attesting software. We use report_data to embed nonces, state root hashes, or enclave public keys in the quote.

- **Measurement:** A cryptographic hash representing the identity of the enclave or trust domain. In SGX, this is called MRENCLAVE (and MRSIGNER for the signing key's hash). In TDX, an equivalent measurement (let's call it MRTD) is included in the quote. The measurement is calculated over the initial code/data loaded into the enclave; if the code changes, the measurement changes. Verifiers compare this against expected values to decide if the enclave is authorized.
- **Quoting Enclave (QE / TDQE):** An Intel-signed enclave that runs on the host and is responsible for signing attestation quotes on behalf of the CPU. For SGX, the quoting enclave uses the Provisioning Certification Key (PCK) of the CPU to sign. For TDX, the TD Quoting Enclave does similarly for a Trust Domain's report. It essentially bridges the gap between the TEE and an external verifiable signature.
- **DCAP (Data Center Attestation Primitives):** Intel's attestation model that allows offline verification. It provides the means to fetch the necessary certificates (PCK cert, CA certs, TCB info, CRLs) so that a quote can be verified without contacting Intel's online IAS service. DCAP is the method we use, enabling on-chain or off-chain trustless verification of quotes.
- **zkVM (Zero-Knowledge Virtual Machine):** A virtual machine that can execute programs and produce a zero-knowledge proof of their correct execution. In our design, we use the Risc0 zkVM to run the quote verification code, producing a SNARK. The zkVM ensures the verification was done correctly without relying on a third-party trust. It basically generates a proof of execution trace of the verification algorithm.
- **zkSNARK (Zero-Knowledge Succinct Non-interactive Argument of Knowledge):** A type of cryptographic proof that attests to the truth of a statement (e.g., "this attestation quote is valid") without revealing additional information.
- **Remote Attestation proof (via ZKP):** We use a Groth16 SNARK (which requires a one-time trusted setup) to verify attestation, because it yields small proofs (under 200 bytes) and fast verification. SNARKs enable our on-chain contract to confirm heavy computations (like ECDSA verifications) had proper outcomes by just checking a small proof.
- **RTP (Real-Time Proofs):** In *t1*, *real-time proof* denotes the cryptographic assurance that each newly proposed roll-up state root originates from an Executor Trust Domain whose most recent **event-driven attestation** has been accepted in `NodeRegistry`. Concretely, the block header is signed by the enclave's key, the on-chain contract verifies the signature with `ecrecover`, and it confirms the signer's key is marked *allowed*—meaning it passed a zk-compressed attestation at the latest onboarding, upgrade, or TCB-recovery event. This yields near-instant finality for every block without requiring a fresh attestation proof per block; attestations are generated only at security-relevant events while block-level confirmation remains real-time through signature checks.
- **Encrypted Mempool (partially-encrypted):** A mempool where only the privacy-sensitive payload of each pending transaction (recipient, value/amount, calldata, access-list, AA fields, intent metadata, etc.) is encrypted, while the header fields needed for spam filtering and fee-based prioritisation—`from`, `nonce`, `gasLimit`, `maxFeePerGas`, `maxPriorityFeePerGas`, `tx-type` tag, ciphertext length, and the `v,r,s` signature—remain in plaintext. This design lets sequencers drop invalid or zero-fee blobs and respect gas limits, avoiding the DoS risk of a fully opaque mempool, yet still hides actionable details from outside observers until the transaction is executed, thereby mitigating MEV exploits such as frontrunning and sandwich attacks. In t1, every Executor Trust Domain deterministically derives an HPKE key-pair (Pub_HPKE, Priv_HPKE) from a sealed `root_secret`; Pub_HPKE is published on-chain (its hash appears in the enclave's attestation as `HashPubHPKE`). Users encrypt the payload with Pub_HPKE; enclaves decrypt it internally with Priv_HPKE once the transaction is admitted to a block. Key rotation and on-chain attestation ensure that only up-to-date, authorised enclaves can ever read the encrypted portion.
- **Sealing (Enclave Sealing):** The process of an enclave encrypting data to disk such that it can be recovered later (by itself or another enclave, depending on policy). Intel SGX provides a seal key unique to an enclave's identity (or signing key) for this purpose. In TDX, similar sealing can be done either via SGX enclaves or by the application using a derived key. Sealed data is tied to the TEE, meaning outside entities cannot decrypt it. We utilize sealing for storing the `root_secret` (from which HPKE keys are derived) between reboots, etc.
- **Proof-of-Read Trie:** A Merkle trie constructed by the enclave to record all external data reads it performed during a rollup block. Each read (from an outside chain or source) is logged as a leaf, and the trie's root hash (Proof-of-Read root) is published. This acts as a commitment to those reads, enabling verification that the enclave didn't fabricate or omit external data. It's analogous to how a Merkle proof commits to a set of values. The term "Proof-of-Read" is specific to our design for cross-chain consistency.
- **TCB (Trusted Computing Base):** The set of components that must be trusted for the system's security. In our case, the TCB includes the CPU, its microcode, the TEE firmware (TDX module), the attestation enclaves (PCE, QE), and the rollup enclave code itself. When we talk about TCB recovery, we mean updating one or more of these components to eliminate a vulnerability. The attestation process conveys information about the TCB (e.g., CPU SVN, etc.) so verifiers can judge if the TCB is up-to-date.
- **TCB Recovery:** A procedure following discovery of a vulnerability whereby the platform's TCB is updated (patches applied) and the attestation mechanism uses new keys or reports to prove that the update happened. It often involves revoking old attestation keys so that only updated platforms can produce valid attestations. For t1, TCB recovery means all nodes must update and re-attest on a new TCB level, and the contracts will reject attestations from before the fix.
- **SVN (Security Version Number):** A number indicating the version of security-relevant firmware/microcode. Intel attestation quotes include SVNs for various components (e.g., the CPU microcode, the TDX module, etc.). If a vulnerability is fixed, the SVN increments.

Verifiers compare SVNs to a baseline to ensure the platform is patched. An attestation quote might be marked “OutOfDate” if SVNs are below expected.

- **Allowlist (of Measurements/Keys):** In our context, a list of enclave identities (measurements or public keys) that are considered authorized. The on-chain verifier contract uses an allowlist of acceptable enclave measurements and may also track allowed enclave signing keys. This prevents unauthorized code from being accepted even if attested (since only known good measurements are valid). It’s a governance tool to manage which enclave versions are allowed.
- **Multisig (Multisignature Wallet):** A wallet or contract that requires multiple parties to sign off on an action. Mentioned here as part of governance – e.g., a multisig might control upgrades or emergency pauses in t1. Not TEE-specific, but relevant for who controls allowlists and upgrades.
- **Groth16:** A specific zkSNARK proving system that yields 3-element proofs and verifies quickly (with pairing checks). We use it for attestation verification, referencing that verifying a Groth16 proof costs ~250k gas on Ethereum. It requires a trusted setup, but many such setups have been performed (including universal ones like Powers of Tau).
- **Risc0:** A particular zkVM implementation based on RISC-V architecture. Risc0 uses a STARK (FRI-based) to prove execution and then can output a SNARK. We cited it as our zkVM of choice because it can run Rust code (like the quote verifier) easily. It’s developed by the Risc0 team and used in several projects needing general-purpose ZKP computation.
- **MEV (Maximal Extractable Value):** The value that a block producer can extract by reordering, including, or censoring transactions. We mention MEV because encrypted mempool and fair ordering aim to reduce MEV extraction on t1. TEEs help here by hiding transaction info until after ordering is decided.
- **Side-Channel Attack:** An attack that derives secrets from indirect information like timing, power consumption, or memory access patterns, rather than breaking cryptography head-on. TEEs are known to be vulnerable to certain side channels (e.g., cache timing attacks) if countermeasures aren’t in place. We acknowledge this in Section 7. Side-channel defenses often require both hardware and software strategies.
- **MRSIGNER:** In SGX, the hash of the public key that signed an enclave. Enclaves signed by the same key can be given certain privileges like accessing common sealed data. While TDX attestation focuses on measurement, one can use signing keys to allow a range of code hashes. We use a concept akin to MRSIGNER to allow new enclave versions to unseal data from old versions (as long as the signing key is the same). Essentially, this trusts the developer signature instead of the exact code identity for certain operations.
- **PCS (Provisioning Certification Service):** Intel’s online service that provides attestation collateral – like PCK certificates, revocation lists, TCB info. In DCAP, this is often an API to fetch the latest JSON of revocations and such. We input this collateral to our verifier. It’s part of the external dependencies for verifying quotes, but once fetched it can be reused many times (hence suitable for on-chain after proof generation).
- **QvE (Quote Verification Enclave):** An optional Intel-provided SGX enclave that can perform quote verification and output a signed report of the result. We don’t use QvE in our design (we do verification in zkVM instead), but it’s a component in Intel’s attestation flow for cases where the verifier is on a TEE platform. We mention it only insofar as it appears in some references.
- **Epoch (Rollup Epoch):** A period or sequence number in the rollup. Not a TEE term, but mentioned when discussing sequencer selection or staking. An epoch might correspond to a set of blocks after which something rotates (like keys or duties). If we had an enclave set performing a consensus, they might have epochs for which ones active.

The above glossary defines key terms used in this document, aiming to clarify their specific meanings in the context of t1’s design. Familiarity with these concepts is assumed in the main text, but we collate them here for reference and to ensure precision of language given the overlapping terminology in hardware security and blockchain domains.