Arc Privacy Sector: Keeping Blockchain State, Transactions, and Accounts Private and Quantum Safe

Mira Belenkiy, Guy Garcia, Sergey Gorbunov, Christian Gorenflo, Jay Logelin, Christopher Ries, Dragos Rotaru, Abinesh Sarvepalli, Milap Sheth

June 8, 2026

Abstract

Most blockchain platforms expose all transaction data and smart-contract state publicly, creating a fundamental barrier to enterprise adoption in regulated industries where confidentiality is required. Existing privacy-preserving designs face significant adoption challenges: developers must learn new smart-contract languages and execution models, cryptographic overhead from techniques such as homomorphic encryption substantially degrades performance, and the resulting systems break composability with existing contracts and developer tooling.

We present Arc Privacy Sector (APS), a practical confidential-computing system that lets developers deploy existing smart contracts into a private execution environment with minimal modifications. At its core, APS runs a private Ethereum Virtual Machine (pEVM)—a separate execution environment that operates synchronously alongside Arc's public execution environment, with both ledgers advancing in lockstep within the same consensus rounds. The pEVM produces an encrypted state-root commitment that is included in every block of the public Arc ledger. Private transactions are encrypted to a network public key shared across validators running inside hardware enclaves, and they are finalized atomically with public state in the same block. Private contracts are isolated by default and exposed only through configurable access policies.

The architecture preserves strong composability: both execution environments commit within the same block, enabling controlled transitions between public and private execution through bridge precompiles. On an AWS Nitro Enclave deployment, APS sustains roughly 1070 TPS for private transfers. The system uses post-quantum hybrid cryptography on the wire and distributes the master secret key across validators via threshold secret sharing.

1 Introduction

Privacy is fundamental to financial services. Lack of confidentiality remains one of the few technical barriers to broad enterprise adoption of blockchain infrastructure: trade secrets and regulation alike require companies to safeguard financial data. The primary obstacle is no longer pure cryptography. Prior privacy-preserving smart-contract systems [KMS+16, CZK+19, KGM19] have struggled to gain traction because they impose a steep learning curve, offer weak composability, require deep modification of existing contracts, and integrate poorly with standard tooling.

In this work, we design Arc Privacy Sector (APS), a system that simplifies the developer and ecosystem requirements for running privacy-preserving smart contracts while preserving strong composability with public state. At its core, APS runs a private Ethereum Virtual Machine (pEVM)—a separate execution environment that operates synchronously alongside the public Arc ledger. The pEVM executes private transactions and produces an encrypted state-root commitment that is included in every block of the main public ledger. Developers can deploy existing Solidity contracts to APS with minimal modifications.

The design must address a key challenge: a contract written for a public ledger does not become private merely because it is deployed inside an enclave. Without platform intervention, functions such as `balanceOf(account)` would remain externally queryable, leaking exactly the state the contract is meant to

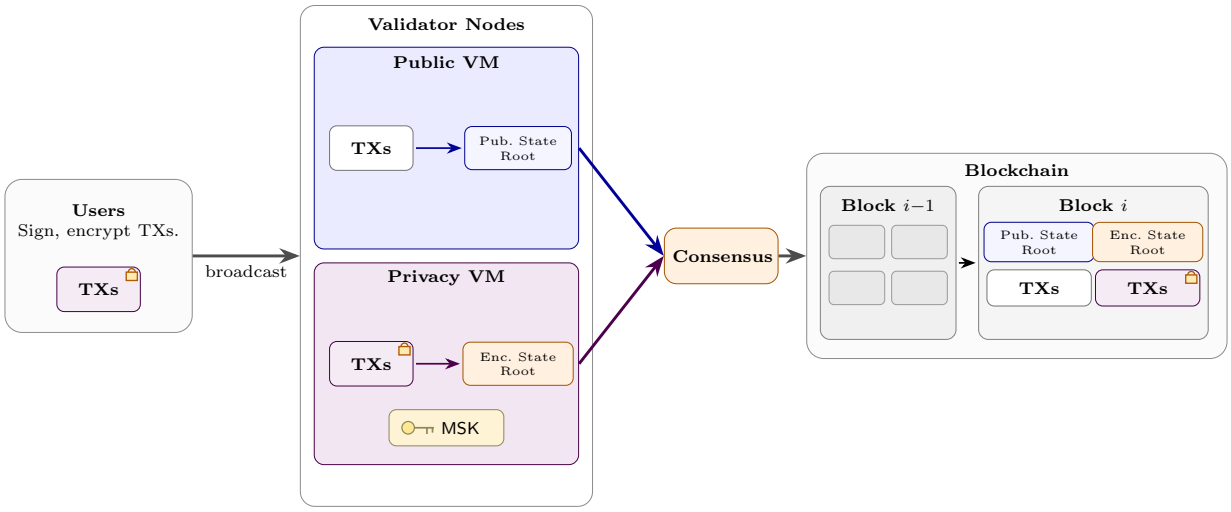


Figure 1: Private transaction lifecycle in APS. Users sign and encrypt private transactions, broadcasting the resulting ciphertext to the validator nodes. Each node runs a Public VM on plaintext public transactions and a Privacy VM that ingests the encrypted transactions and applies them to private state inside the enclave under MSK; both VMs produce state roots that flow into consensus. Every finalized block records the public state root, the encrypted state root, the public transactions, and the encrypted private transactions.

protect [JLLJ+24]. APS therefore implements a sandbox execution environment that enforces isolation by default. Developers opt in to cross-contract interactions through configurable trust domains and function-level access policies, ensuring strong isolation with controlled interoperability where the deployer explicitly allows it. We now outline our design goals and the technical architecture that achieves them.

1.1 Design Goals and Non-Goals

APS is designed to protect the confidentiality of smart contract state and transaction data while preserving the usability and composability of the EVM ecosystem. The core challenge is to enable privacy without forcing developers to abandon existing contracts, tooling, or programming models.

Our first goal is **state and transaction confidentiality**. All contract storage and transaction payloads are encrypted, and only validators running the pEVM inside hardware enclaves can decrypt and process private state. This ensures that transaction contents, intermediate execution state, and persistent storage remain hidden from external observers and from the validator hosts themselves.

Second, we prioritize **reusability of existing contracts**. Developers should be able to deploy existing Solidity contracts with minimal modifications. The vast majority of contract logic—including core business rules, state management, and function implementations—remains unchanged. The primary adaptation required is how contracts expose their interfaces to external callers or other contracts.

Third, APS provides **isolation by default**. Contracts remain isolated even if developers forget to add explicit protections. By default, all contract functions and storage are inaccessible to external callers and other contracts. Exposure of functions or state to the outside world must be explicitly declared. This design prevents accidental data leakage and ensures that privacy guarantees hold even when developers make mistakes.

Fourth, the system is designed to remain secure against **a single or a colluding subset of validators**, below the threshold. The master secret key is distributed across validators and can only be reconstructed inside attested enclaves.

Fifth, APS guarantees **finality-based security**. No rewind of the ledger or compromise of a single validator can leak private data or transactions, provided the consensus safety threshold is maintained. Private

state is committed alongside the Arc consensus as it finalizes each block, and validators verify the committed state before allowing queries against it.

Finally, the system is designed for **post-quantum security** against harvest-now-decrypt-later attacks. All public-key encryption uses hybrid constructions [BCD+24] combining classical and post-quantum algorithms , and symmetric encryption primitives [DR99, Nat23] remain secure against quantum adversaries This protects the confidentiality of today's private transactions even if a sufficiently powerful quantum computer becomes available in the future.

There are also explicit *non-goals*. We do not attempt to hide smart contract code itself: contract bytecode and deployment transactions are visible on the public ledger. We protect state and execution, not the logic visibility itself. We also do not protect against side-channel attacks that may arise from memory access patterns observable by the enclave host or through timing channels. These are orthogonal challenges that can be addressed through techniques such as oblivious RAM at application layer, but we leave them to future work [SvDS+13, SGF18, AKSL18, MPC+18, MIS+22]. Also, more work needs to be put into learning how to design smart contracts that do not expose the private state via its normal functionalities. For instance, AMM that simply allows the user to learn the slippage, inherently reveals the depth of each pool of assets. We provide sample smart contracts for contracts like ERC20s, and, with the community, will continue investigating the broader design space.

1.2 Technical Overview

We now describe the architecture that realizes the goals outlined above. One blockchain runs two execution environments: public state executes on Arc, while private state executes inside hardware enclaves on the APS pEVM. Both environments produce state roots included in the consensus layer.

Users prepare transactions according to standard EVM semantics [Woo25] and sign them with their regular signature keys. To submit a private transaction, users encrypt the signed transaction using the pEVM public key and send the resulting ciphertext as the calldata argument to one of the Arc precompile functions. From the perspective of the public Arc ledger, this looks like an ordinary transaction calling a precompile—the encrypted payload is opaque.

Every APS validator runs the pEVM inside a trusted execution environment. When a validator processes a block containing calls to the privacy precompile, it forwards the encrypted payload to its enclave. The enclave decrypts the transaction using the shared network key, verifies the user's signature, and evaluates the transaction against the private state using the standard EVM execution rules. The privacy precompile returns only an acknowledgement and a predefined gas cost to the public execution layer. No execution results, return values, or event logs are exposed to the host during block processing. This constant-gas design prevents timing side-channels from leaking information about the control flow of private transactions.

Arc queries and includes the pEVM encrypted state root in every block. All validators—both enclaved and non-enclaved—participate in consensus over the same block header, which commits to both the public state root and the encrypted private state root. After consensus finalizes the block, the pEVM light client running inside each enclave verifies the certified block header, checking that more than two-thirds of validators signed the block and that the encrypted state root matches what the enclave computed. Only once that verification succeeds does the pEVM make the previous block's private state queryable by authorized users.

The light client maintains synchronization between Arc and the pEVM. It is a stateless node that receives the block header and the commit certificate from the host. Enclaved validators run an instance of the light client inside the enclave; once Arc finalizes a block, the pEVM receives the Arc block header from its light client and finalizes the corresponding private block.

This design ensures that private and public state advance in lockstep: both are finalized in the same consensus round, both are committed to by the same validator set, and both reflect execution within the same block. This synchronous finalization enables strong composability between the two execution environments. Users can bridge assets between public and private contracts through controlled precompile operations, and the system guarantees that both sides of the bridge see consistent state at each block height.

The pEVM currently runs inside AWS Nitro Enclaves; but nothing in the design limits it to one TEE or another, or restricts to a specific cloud. Any cloud-based TEE inherits residual risks from the hosting

environment: for example, an attacker with physical access to infrastructure could, in principle, observe patterns of encrypted state as it is written to persistent storage. We treat this as a bounded, operational risk that can be further reduced with stronger at-rest encryption and key-management practices.

1.3 Use Cases

APS aims to make privacy-preserving computation practical for general-purpose blockchain applications. The design allows developers to retain the standard Ethereum Virtual Machine programming model, tooling, and composability while introducing private state and execution where required. This broadens the class of applications that can be deployed on-chain, particularly those that depend on hidden state, selective disclosure, or private user interaction. We outline four representative use cases that illustrate the kinds of applications APS enables.

Payroll. Payroll systems require correctness, auditability, and predictable settlement, but they often involve salary, bonus, and employee-identity information that should not be globally visible. On a public blockchain, these systems are difficult to deploy directly because every payment amount, recipient pattern, and treasury outflow can be observed by competitors, counterparties, or other employees. APS enables payroll contracts in which compensation logic executes on-chain while salary amounts, employee balances, and payment schedules remain confidential. Organizations can fund the payroll contract in APS, where individual salary disbursements remain encrypted. This allows organizations to use programmable payroll rails without revealing internal compensation structures, while still benefiting from deterministic execution and composability with other on-chain financial applications.

Lending Markets. Borrowing and lending protocols rely on information about deposits, outstanding debt, collateralization, and liquidation thresholds. On public blockchains, the full transparency of vault positions exposes users to strategy imitation, targeted liquidations, and adverse trading behavior that can arise once market participants observe stressed or highly leveraged positions. APS enables vault state, borrowing activity, and position composition to remain confidential while still allowing protocol rules to execute deterministically. Users deposit collateral into private vaults, borrow against those positions, and manage their risk without broadcasting their portfolio composition or risk profile to the market. Liquidation logic can still execute correctly based on encrypted price feeds and collateral ratios, but the details of individual positions remain hidden until a liquidation actually occurs.

Asset Issuance. Asset issuance protocols manage the lifecycle of digital assets, including minting, burning, distribution, and ownership records. In many institutional settings, the identity of holders, allocation amounts, transfer restrictions, or issuance schedules may be commercially sensitive or subject to confidentiality requirements. On a public blockchain, those properties are typically exposed by default, which limits the kinds of assets that can be issued directly on-chain. APS enables issuers to manage confidential balances and transfer activity while preserving the programmability needed for issuance controls, compliance logic, and settlement workflows. An issuer can deploy a token contract in APS that enforces transfer restrictions, vesting schedules, or allowlist requirements without revealing which addresses hold the asset or how much they own. This opens the door to asset models in which confidentiality is treated as a first-class feature.

Intraday Repo Markets. Intraday repo markets allow institutions to borrow cash or securities against high-quality collateral for very short durations, often to manage liquidity, settlement timing, or balance sheet constraints. These transactions are highly sensitive because the terms of the trade, collateral composition, funding needs, and counterparty relationships can reveal a firm's liquidity position or trading strategy. On a public blockchain, exposing this information in real time would make the venue unattractive for many institutional participants, since even temporary financing activity could leak commercially important signals to the market. APS enables repo terms, collateral movements, and participant balances to remain confidential while still allowing settlement, margining, and repayment logic to execute on-chain. Institutions can post collateral, negotiate terms, and settle transactions programmatically without disclosing the information that gives rise to their financing needs. The synchronous finalization of public and private state ensures that repo transactions can be atomically settled against public market prices or other on-chain state.

These examples illustrate that the value of APS lies not in privacy alone, but in privacy combined with programmability, synchronous composability, and compatibility with existing developer workflows. The

ability to deploy existing contracts with minimal modifications lowers the barrier to adoption, while the synchronous execution model preserves the composability that makes smart contract platforms valuable.

1.4 Evaluation

On an AWS Nitro Enclave deployment, APS achieves a private transfer throughput of 1070 TPS, which is a significant performance improvement over other approaches. Section 6 gives more details of our evaluation.

1.5 Other Approaches

Privacy-preserving blockchain systems face a fundamental trade-off between generality, performance, and composability. Existing approaches can be broadly categorized by their underlying cryptographic technique and the resulting constraints on usability and integration with existing ecosystems.

Cryptography-based approaches. There are blockchain systems based on complex zero-knowledge proof systems such as Zerocash [BCG+14], Zexe [BCG+20], and VeriZexe [XCZ+23] enable private transactions with strong cryptographic guarantees. These systems typically focus on specific use cases such as private transfers; extending them to support general smart-contract composability remains challenging. Systems like Zether [BAZB20], deployed by Solana [Sol26] and Aptos [Apt26], combine zero-knowledge proofs with homomorphic encryption to enable private asset transfers, but are limited to token operations and do not support arbitrary smart-contract interactions.

Fully homomorphic encryption (FHE) offers the potential for generic computation over encrypted data [Fhe26, Sun26, CGGI20]. While FHE-based systems can support programmable smart contracts without trusted hardware, the computational overhead of homomorphic operations is substantial. Current FHE implementations are orders of magnitude slower than plaintext execution, making them suitable for specific high-value use cases but not for general-purpose transaction processing at scale. Hybrid approaches that combine FHE with multiparty computation for key management [DDE+23] or with secure enclaves for key shares add complexity and still inherit the performance limitations of the underlying FHE operations.

MPC-based solutions [Avi25, TAC25] distribute trust across multiple parties and can support general computation, but require developers to reason about secret-shared execution models that differ substantially from standard smart-contract programming. The need to coordinate multiple parties also introduces latency and availability challenges that complicate integration with existing blockchain infrastructure.

Enclave-based approaches. Trusted execution environments offer an alternative path to privacy that avoids the performance overhead of cryptographic techniques. Systems such as Secret Network [Net25], Oasis Sapphire [Pro26, CZK+19], Phala [Net26], and Ten Protocol [Ten26a] encrypt transactions and execute them inside hardware enclaves, maintaining encrypted state that can only be accessed within the TEE.

These systems support general smart-contract execution and achieve performance closer to plaintext execution than FHE-based approaches. However, they typically operate as standalone chains or as separate execution layers. Secret Network and Oasis Sapphire run independent consensus protocols, which limits composability with existing public blockchain ecosystems. Ten Protocol provides an encrypted Layer 2 for Ethereum but does not synchronize private and public state within the same consensus round, requiring asynchronous bridging for cross-domain interactions.

The key distinguishing feature of APS is synchronous execution: both public and private state are finalized in the same block by the same validator set. This enables atomic composability between public and private contracts without cross-chain bridging delays or additional trust assumptions. Users can move assets between execution environments through controlled precompile operations that execute deterministically within a single block, preserving the composability properties that make smart-contract platforms valuable while adding privacy where needed.

Kaptchuk, Green, and Miers also use a ledger to give state to otherwise stateless enclave computations [KGM19]. Their construction prevents rollback and forking by requiring the host to commit inputs to the ledger and by deriving the enclave's randomness deterministically from the committed execution history. APS uses the same high-level idea where the ledger fixes the execution trace but places the binding inside Arc consensus. Encrypted private transactions are included in the same block that use the certified encrypted

private state root, and enclaved validators re-execute the deterministic pEVM transition before voting. As a result, APS does not need a prior input-commitment round.

Like other enclave-based systems, APS inherits the trust assumptions of the underlying TEE. We use AWS Nitro Enclaves, which provide isolation from the host operating system and from end users without physical access to the hardware. Side-channel attacks based on memory access patterns [JLLJ+24] remain an orthogonal concern that can be addressed through techniques such as oblivious RAM [MIS+22, JLLJ+24, Dis24, SMCM25], but we leave this to future work.

1.6 Organization

The rest of this document is organized as follows. Section 2 sets up the network and threat model, walks through the private-transaction lifecycle, and argues consensus security. Section 3 details how APS generates, rotates, and recovers the shared master key. Section 4 describes the APS sandbox, the contract isolation model, and access control for cross-contract interaction. Section 5 specifies the concrete ciphersuites and parameters used for key derivation, signing, and encryption.

2 One Blockchain, Two Execution Environments

2.1 Network Setup and Assumptions

APS reuses Arc's validator management. The initial validator set is configured in the genesis file, and a public smart contract on Arc maintains the live set thereafter. The same contract also publishes the APS KMS attestation policy—the rules each organization's KMS uses to recognize valid dealer, seed, and validator enclave builds. Each organization independently verifies the published policy and loads it into its own KMS before participating.

- **Dealer.** A single trusted enclave dealer that generates MSK at bootstrap, creates Shamir shares inside the enclave, encrypts those shares through direct attested access to each participating organization's KMS, and publishes the encrypted shares with a dealer attestation.
- **Seed node.** A specialized enclave that reconstructs MSK from the published encrypted shares by connecting directly to the participating organizations' KMSs, then holds MSK in memory and serves it over attested TLS to any enclave matching its attestation policy. The policy is part of the seed node's launch configuration. Seed nodes share MSK with validators, and full nodes whose enclaves match the attestation policy.
- **Validator.** A pEVM instance running inside an enclave alongside an Arc node. Validators verify the dealer attestation and fetch MSK from approved seed nodes.
- **KMS.** A key management service (e.g., AWS KMS) that releases secrets only to enclaves matching its attestation policy. Each organization runs an independent KMS, loaded with the policy that Arc publishes.

pEVM enclaves communicate among themselves and with their organization KMS over post-quantum TLS (Section 3). AWS currently supports only classical signatures for attestation; this does not compromise the secrecy of MSK or its shares MSK_i, because all key material travels over the post-quantum-secure outer channel.

Alongside the enclaved nodes above, the network also supports non-enclaved standard nodes, archive nodes for historical data, and relay nodes for transaction propagation. These additional node types are not essential to the core construction but support operational and usability needs.

Threat model. APS runs on a network of N validators, of which up to $t < N/3$ may be Byzantine. The Byzantine nodes may collude to learn information about transactions or corrupt state. We assume:

- **Consensus security.** The Arc consensus protocol is safe against Byzantine adversaries controlling up to $t < N/3$ validators.
- **Enclave security.** Validators run the pEVM inside trusted execution environments (AWS Nitro Enclaves in the current deployment) that provide authenticated execution and confidentiality. The adversary cannot extract secrets from an honest validator's enclave or tamper with code running inside it.
- **Untrusted host.** The machine hosting the enclave is untrusted from the enclave's perspective. A malicious host may replay old blocks, supply forged or partial data (tampered receipts, fake certificates), inject fabricated private transactions, or kill and restart the enclave at will. We do not address side-channel attacks (timing, traffic analysis, access patterns) in this paper.
- **Threshold cryptography.** MSK is secret-shared among the N validators using a threshold scheme. Reconstruction requires at least $T_{\mathsf{Recon}} = 2t + 1$ shares—matching Arc's BFT safety threshold—and only happens inside an enclave during initialization or recovery (Section 3).

2.2 Construction

Arc runs two execution environments under a single consensus layer. The public environment L maintains plaintext, publicly visible state, while the private environment $\mathsf{L_p}$ maintains private state that is accessible only inside enclaves. Both environments produce state roots that are committed together in each block, so the two ledgers advance in lockstep. This synchronous finalization enables atomic composability: public and private contracts can interact within the same block without cross-domain bridging delays.

Users interact with APS by calling Arc precompiles with encrypted payloads; the precompiles handle deployment (Section 4.5), execution (Section 4.6), and query authorization (Section 4.8). The pEVM treats each precompile call as a private transaction and updates its state accordingly:

$$\mathsf{pEVM}(\mathsf{st}_i, \bar{\mathsf{tx}}) \;\rightarrow\; \mathsf{st}_{i+1},$$

where st_{i+1} is the private state after executing the encrypted transaction $\bar{\mathsf{tx}}$ against the previous state st_i. At the end of each block the pEVM outputs an encrypted state root; Arc includes that root in the block header during consensus. After consensus finalizes the block, the pEVM light client running inside each enclaved node verifies the Arc commit certificate (more than $\frac{2}{3}$ of validator voting power) and authenticates the header before the enclave commits private state.

The privacy precompile returns only an acknowledgement carrying a predefined gas cost, so no execution results or state changes are visible to the public ledger during block processing. Users retrieve transaction outcomes from an enclave-backed RPC node once the block containing the transaction finalizes; with sub-second block times, query latency is on the order of a single block. RPC queries return state at the latest finalized block.

APS uses a single shared master secret key MSK distributed across validators using threshold secret sharing (Section 3). Validators reconstruct MSK only inside their enclaves. All keys derive from MSK, which serves three roles: deriving the network public key for encrypting user transactions, deriving per-block keys for encrypting each block's state-root commitment in the header, and deriving the keys for encrypting the snapshots and write-ahead logs the enclave persists to its untrusted host.

Users sign transactions with standard ECDSA keys and encrypt them under the APS network public key before submitting to Arc precompiles. No key registration or persistent client-side secret is required. A user may optionally include an ephemeral public key with the submission so that query results returned later can be encrypted back to that key.

2.3 Private Transaction Lifecycle

We now describe how a private transaction is constructed, executed, and committed to a block.

1. The user prepares a `PrivateTx` that is valid under standard EVM semantics and signs the EIP-712 typed data with their usual signature key.

2. The user wraps the signed `PrivateTx` into a payload tx' targeting an Arc privacy precompile. To receive an encrypted result later, the user may include an ephemeral public key EPK_u as one of the precompile arguments; otherwise the pEVM returns no result.

3. The user encrypts the wrapped payload as $\mathsf{ct} \leftarrow \mathsf{HPKE.Encrypt}(\mathsf{EPK}, \mathsf{tx}')$ under the APS global public key [BBLW22].

4. The ciphertext ct is encoded as bytes and submitted to the public ledger as the calldata of a transaction that calls the Arc privacy precompile $\mathsf{executePrivateTx}$. Any account may submit this transaction and pay its gas; the submitter need not be the originating user. The transaction enters Arc's mempool.

5. When a validator is selected as proposer for block height $i{+}1$, it picks transactions from the mempool and constructs block B_{i+1}. The proposer must run an enclave. It executes each selected transaction: public transactions advance public state under ordinary EVM rules, and each privacy-precompile call $\mathsf{executePrivateTx}(\mathtt{bytes}(\mathsf{ct}))$ is dispatched to the proposer's local pEVM. The enclave decrypts the payload $\mathsf{tx}' \leftarrow \mathsf{HPKE.Decrypt}(\mathsf{ESK}, \mathsf{ct})$, verifies the EIP-712 signature on the enclosed `PrivateTx` to recover `msg.sender`, and executes the transaction, advancing the private state to $\bar{\mathsf{st}}_{i+1}$. The precompile returns a predefined gas cost to the public execution layer, independent of the actual private execution cost. After processing all transactions in the block, the proposer queries the pEVM for the encrypted state root (Section A.5) and assembles the block header, which contains the public state root ρ_{i+1}, the encrypted private state root $\mathsf{EncRoot}_{i+1}$ (`private_state_root` in the header), a *validator-set hash*—a hash of the active validator set that signed this block's certificate, taken from the public consensus state—and, when the validator set changes this block, a *validator-set diff* listing the registrations, activations, removals, and voting-power changes that take effect for the next block. The validator-set hash serves as a per-block fail-safe the enclave can use to verify its local copy of the set. The proposer then broadcasts B_{i+1} to the network.

6. When a node receives the proposed block, its behavior depends on whether it runs an enclave.

   A **non-enclave node** executes only the public portion of the block. Each privacy-precompile call is a no-op that returns the predefined gas cost and has no public side effects. The node validates the resulting public state root against ρ_{i+1} and treats $\mathsf{EncRoot}_{i+1}$, the validator-set hash, and the validator-set diff (when present) as opaque bytes. It performs no private-state verification.

   An **enclave node** re-runs the proposer's work end-to-end. For each privacy-precompile call, its local pEVM performs the same decrypt–verify–execute pipeline and produces its own $\bar{\mathsf{st}}_{i+1}$. After processing all transactions, the node queries the pEVM for the encrypted state root and checks that it matches $\mathsf{EncRoot}_{i+1}$ in the proposed header. A mismatch invalidates the block.

7. Because validating a proposed block requires an enclave, and validators vote only on blocks they have validated, the APS protocol requires every consensus validator to run an enclave. Once more than $\frac{2}{3}$ of voting power signs B_{i+1}, the block is finalized; the BFT commit certificate on the block hash transitively certifies both the public state root and the private-state header fields. The validators thus reach consensus on $B_{i+1} = (\{\mathsf{tx}\}, \rho_{i+1}, \mathsf{EncRoot}_{i+1})$.

8. Once the block is finalized, the pEVM light client running inside each enclaved node authenticates the finalized block via its commit certificate and atomically commits the private state the enclave has already produced. Non-enclave nodes skip this step—they have no private state to commit. The light client is *stateless*: all persistent state (current validator set, last committed block header, encrypted state root) is read from the pEVM on each invocation. It receives the block header and the commit certificate from the host, which is untrusted (Section 2.1). It performs the following checks:

(a) **Chain continuity.** The block number must be exactly one greater than the last committed height, and the parent hash must match the previous block's hash.

(b) **Validator-set hash check.** The light client hashes its maintained validator set and compares it to the validator-set hash in the header. A mismatch indicates the maintained set has drifted from what the rest of the network signed with, and the block is rejected before the suspect set is trusted to verify anything else.

(c) **Certificate verification.** The BFT commit certificate is verified against the maintained validator set: more than $\frac{2}{3}$ of the total voting power must have signed the block hash.

(d) **Encrypted state root.** The light client reads the current encrypted state root from the pEVM and compares it to $\mathsf{EncRoot}_{i+1}$ in the certified header. A mismatch means the pEVM's internal state has diverged from what the honest validators computed.

(e) **Atomic commit.** If all checks pass, the light client instructs the pEVM to atomically commit the private state, the block header, and—when the header carries a validator-set diff—the maintained validator set updated by applying that diff. The block header is persisted so that on restart the enclave can verify its snapshot against the block it belongs to. Until this call succeeds, private state is uncommitted, and the enclaved validator cannot begin processing transactions for the next block.

Processing is strictly sequential: blocks arriving faster than the light client can verify are buffered and processed in order.

Encrypted state root. The pEVM encrypts the private state root under a per-block key derived from MSK and the block height. The AEAD nonce is fresh randomness drawn per encryption; the proposer commits it in the block header alongside the ciphertext and authentication tag. A validator that computed the same plaintext state root re-encrypts it under the transmitted nonce inside its own enclave and compares the result against the header bundle: a match certifies that the proposer's plaintext root matched its own, while a mismatch invalidates the block. Section A.5 gives the full construction.

Persistence to disk. Inside the enclave, the pEVM holds private state in plaintext; encryption applies only when state crosses the enclave/host boundary. The host stores two kinds of artifacts: periodic full snapshots of the private database, and per-block write-ahead logs that capture each block's state changes. We call each such artifact an *emission*. Each emission is encrypted with AEAD-AES-256-GCM-SIV under a fresh key derived from MSK at the start of the emission; the key is never persisted. Each validator persists independently; only the encrypted state root in the block header is shared across the network. Section A.4 gives the construction.

Querying private transaction results. A user retrieves a private transaction's result by identifying the transaction (e.g., by transaction hash) and presenting a signature from the same key that signed the original transaction. The query is directed at an enclaved RPC node, which returns the encrypted result only when both of the following hold:

1. The block B_i containing the transaction has been finalized by BFT consensus, and the light client has verified and committed the private state for that block.
2. The pEVM is operating on the authenticated state corresponding to the committed state root for block B_i.

This mirrors the public execution layer, where queries are served only against canonical state (i.e., after the consensus layer promotes the block via `forkchoiceUpdated`). For private state, the light client's atomic commit plays the analogous role: it is the gate after which private state becomes queryable. Restricting reads to post-finalization state also defends against host-extraction probes: no speculative state from fabricated host inputs ever becomes queryable, so a malicious host cannot use the enclave as an oracle.

Startup and catch-up. The enclave bootstraps from either the hardcoded genesis state (at block 0) or a pre-authenticated encrypted snapshot at a later height. Once initialized, the light client accepts only the next sequential block and requests any missing blocks from the host, processing them in order until it reaches the current finalized height.

2.4 APS Consensus Security

We summarize how APS inherits the safety properties of Arc's BFT layer (Malachite-style, Tendermint-like [BKM18, Cir26]): with N validators, the usual guarantees hold when fewer than $N/3$ are Byzantine—equivalently, when more than two-thirds of validators are honest. The argument has three pieces: honest validators agree on the encrypted state root, Arc's commit certificate binds that root to its block header, and no information about a block leaks to the public until that block is finalized.

Agreement on the encrypted state root. Honest validators start each block from the same finalized Arc inputs (height, parent commitment, ordered transactions). Deterministic execution yields the same plaintext state root. Each honest validator re-encrypts its local root under the nonce committed by the proposer (Section A.5); honest validators that agree on the plaintext therefore produce matching ciphertexts and accept the proposer's header bundle.

Binding of Arc to pEVM state. With fewer than $N/3$ faulty validators, more than two-thirds of replicas re-encrypt to the proposer's ciphertext and vote to finalize the block. Standard BFT safety implies that only one value can be committed per height, and that value must be the one accepted by the honest supermajority. The encrypted state root carried in Arc's block header therefore matches what honest pEVM execution produced for that height.

Privacy of pEVM blocks until finalization. Before executing block $i{+}1$, each validator's enclave verifies Arc's commit certificate for block i through the light client (Step 7 in the transaction lifecycle). A malicious host cannot bypass this check; it may waste enclave cycles by replaying or fabricating inputs that fail verification, but it cannot cause an honest validator to treat uncommitted pEVM state as final. Hence the pEVM cannot finalize block i until Arc finalizes it, and no information about block i is released to the public until that point.

3 Key Management

APS uses a single shared master secret key MSK from which enclaved validators derive every other key they need. To survive validator restarts and partial outages without re-running setup, the system persists one Shamir share MSK_i of MSK per validator organization (threshold T_{Recon}, distributed via Shamir secret sharing [Sha79]). The share is itself encrypted under an organization-managed key k_i so that the stored ciphertext $\mathsf{Enc}_{k_i}(\mathsf{MSK}_i)$ is useless without an attested unwrap through the organization's KMS. The master key MSK can be reconstructed only when an approved enclave decrypts at least T_{Recon} shares and runs Shamir reconstruction. Validators belong to multiple organizations, and each organization manages its own KMS with an attestation policy that regulates which enclave profiles it accepts.

Bootstrap. To start the system, a single enclaved *dealer* generates MSK, derives the Shamir shares, connects directly to each validator organization's KMS from inside the enclave, and publishes the resulting encrypted shares on Arc together with an attestation of the dealer image and share-generation transcript. Validators verify the dealer attestation and the encrypted share records on the chain. Any validator that joins the system later must run with a previously approved attestation profile, or a quorum of T_{Agree} existing validators must approve the new profile.

Seed nodes. Alongside validators, the system runs one or more *seed nodes* [Ten26b]. A seed node is itself an enclave whose profile is approved through the same process as a validator. Seed nodes, are responsible for reconstructing and serving MSK after bootstrap: a seed node reads the encrypted share records from Arc, connects directly to the participating organizations' KMSs under attestation, reconstructs MSK from at least T_{Recon} decrypted shares, and then serves MSK over attested TLS to whitelisted pEVM enclaves. New seed nodes must be approved by a threshold of T_{Agree} validators, and each organization's KMS independently decides which seed nodes can retrieve the encryption key for the encrypted MSK_i.

Each organization's KMS binds key material to a trusted execution environment: keys are released or exported under wrapping only after the KMS verifies an attestation that satisfies the organization's policy. Concrete APIs differ by platform; the figures and protocol below describe the logical flow shared by any major cloud's TEE-aware KMS (Azure, AWS, Google, and similar).

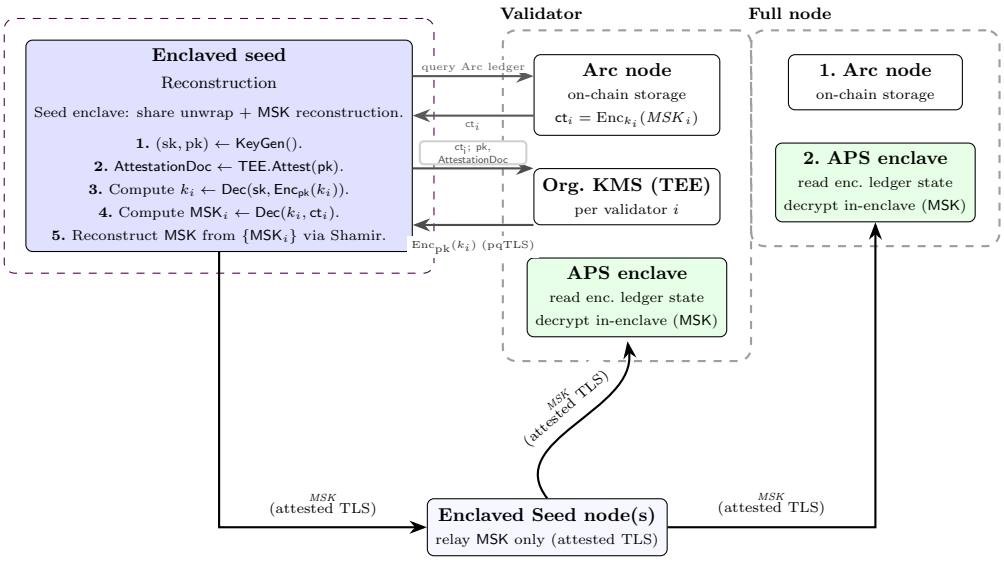


Figure 2: Key reconstruction. Each validator has an organization-scoped KMS integrated with TEE attestation (separate regions or accounts may still isolate tenants). A *Validator* (dotted) comprises an Arc node, that organization's KMS, and a pEVM enclave that reads encrypted ledger state and decrypts in-enclave under MSK. A *Full node* (dotted) comprises only an Arc node and a pEVM enclave. The **dealer enclave** generates MSK and publishes encrypted Shamir shares created through direct attested access to each organization's KMS. Only dealer- and seed-class enclaves can perform an attested unwrap against the organization's KMS; seed enclaves query Arc and the organization KMS, reconstruct MSK from shares, and relay MSK to whitelisted pEVM enclaves over attested TLS.

3.1 Key Generation

The dealer enclave samples a random master key MSK and a random polynomial F of degree $T_{\text{Recon}} - 1$ with $F(0) = \text{MSK}$, producing one share $\text{MSK}_i = F(i)$ per validator. We set $T_{\text{Recon}} = 2t + 1$ so that reconstruction requires the same supermajority as Arc's BFT safety threshold (Section 2.1); any T_{Recon} shares suffice to reconstruct MSK via Shamir reconstruction, and any $T_{\text{Recon}} - 1$ or fewer shares reveal nothing about it. For each validator organization, the dealer uses that organization's KMS from inside the dealer enclave to encrypt MSK_i for persistent storage, following the attested-unwrap pattern detailed in Section 3.2.

The dealer publishes the encrypted shares and its attestation on Arc. Seed nodes then reconstruct MSK by unwrapping enough posted shares through the corresponding organization KMSs, and serve MSK to whitelisted pEVM enclaves over attested TLS [WGSH22]; the pEVM enclaves use it to decrypt ledger state.

3.2 Key Shard Storage

Each encrypted share MSK_i is persisted under a key derived from the organization KMS. The procedure below mirrors the attested-unwrap pattern used in similar systems [Zam25]:

1. The validator organization configures the organization KMS with a symmetric key under identifier $\text{RootKey}_{\text{Id}}$ and an attestation policy listing approved dealer- and seed-class enclave builds.

2. The dealer enclave generates an asymmetric keypair (PK, SK) and produces an attestation binding the public key to the enclave's measured software state: AttestationDoc ← TEE.Attest(PK).

3. The dealer enclave prepares a recipient header containing AttestationDoc and PK and requests a fresh data key wrapped to that recipient (conceptually $\text{WrapKey}(\text{RootKey}_{\text{Id}})$ or an equivalent attested-unwrap API). On successful verification the KMS returns $\text{ct} \leftarrow \text{Enc}_{\text{PK}}(K)$.

4. The dealer enclave decrypts $K \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ and seals the organization's share under K: $\mathsf{ct}' \leftarrow \mathsf{Enc}(K, \mathsf{MSK}_i)$.

5. The dealer enclave publishes ct' on Arc with the associated identifier $\mathsf{RootKey}_{\mathsf{Id}}$ and the dealer attestation; the encrypted blob may also be mirrored to persistent object storage.

Retrieval reverses the flow: a seed enclave reads ct' from Arc or persistent storage, asks the organization KMS to unwrap K under $\mathsf{RootKey}_{\mathsf{Id}}$ subject to the same attestation policy, and decrypts ct' to recover MSK_i. After recovering at least T_{Recon} shares, the seed enclave reconstructs MSK and serves it only to whitelisted enclaves over attested TLS. Enclave–KMS traffic runs over a post-quantum-ready TLS variant. Some KMS products still wrap data keys with RSA today; keeping the outer channel post-quantum mitigates harvest-now–decrypt-later risk for the wrapping traffic itself.

3.3 Key Rotation

APS rotates MSK either on a scheduled cadence—bounding the exposure of any single key over time—or in response to suspected compromise. The mechanism below replaces MSK with a fresh MSK' and coordinates the cutover so that every component that derives keys from MSK rotates with it.

At block height X, the dealer produces a fresh master key MSK' via $\mathsf{RotateKey}$, builds new Shamir shares MSK'_i, wraps them through each organization's KMS using the same attested-unwrap pattern as in Section 3.2, and publishes the resulting ciphertexts ct'_i on Arc with its rotation attestation. Seed nodes reconstruct MSK' from the posted encrypted shares and relay MSK' to pEVM enclaves over attested TLS while consensus continues to execute under the old MSK. From block X onward, enclaves use MSK' to derive the keys that operate on *new* block data; by block $X+d$ the cutover for those new-data keys is complete network-wide. Persisted artifacts already on disk remain encrypted under MSK and continue to be decrypted under MSK until they age past the host's retention window.

A master-key rotation also rotates three derived secrets that other components of the system depend on; each is recovered deterministically from MSK' but requires a brief coordination window across the rotation boundary.

Network public key. Private transactions are submitted to APS encrypted under the network public key. The post-rotation validator set derives a new EPK from MSK' and publishes it in advance of the rotation boundary. Transactions encrypted under the old key that miss inclusion in the old epoch must be resubmitted under the new key.

State-root encryption key. The post-rotation validators derive a new per-block state-root key from MSK' (Section A.5). The derivation is fully determined by MSK' and the block height, so all enclaved validators converge on the same key without an explicit synchronization step.

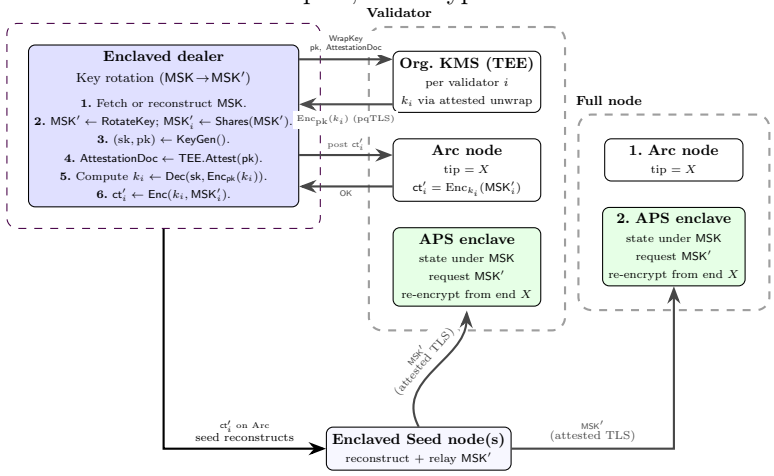
Persistence keys. Persisted artifacts—per-block write-ahead logs and periodic snapshots (Section A.4)—are encrypted under fresh per-block keys derived from the MSK active when the artifact is written. After rotation, new artifacts are produced under MSK'; the previous MSK remains in use for decrypting pre-rotation artifacts and is destroyed only once those artifacts have aged past the configured retention window.

Forward and backward security. Rotation is the lever APS provides against MSK compromise. Once enclaves switch to MSK', every subsequent network-key decryption, state-root encryption, and persistence write uses keys derived from MSK'; a holder of the old MSK cannot decrypt anything produced after the cutover (forward security). Conversely, data produced under a prior MSK becomes undecryptable once that MSK is destroyed; APS destroys an old MSK only after retention has aged out the last persisted artifact still encrypted under it (backward security). The master key itself never leaves the enclave in plaintext; an adversary aiming to recover it must compromise the underlying KMS-side share custody, which is the trust assumption inherited from Section 3.2.

3.4 Key Recovery

If a seed node crashes, it initiates the key recovery protocol on reboot. The seed establishes attested TLS connections to each validator's organization KMS, and each KMS checks that the seed's attestation evidence

(a) Block X — rotation at tip X; re-encryption starts at end of block X.



(b) Block $X+d$ — tip $X+d$; re-encryption complete by here; decrypt with MSK' only.

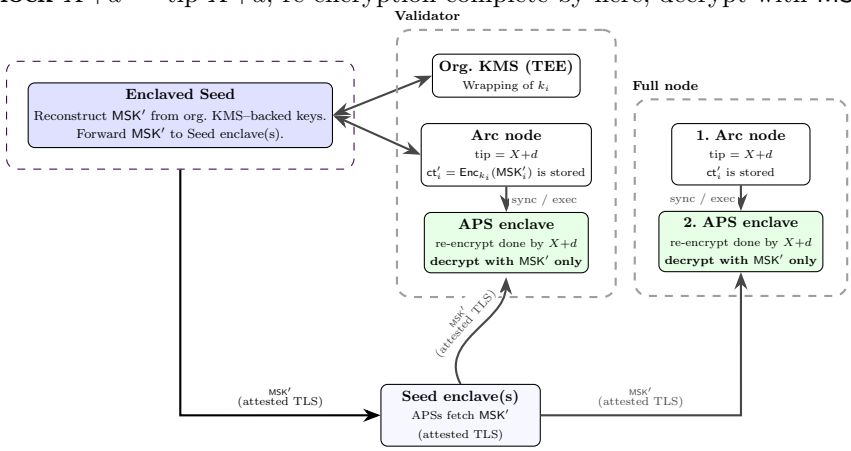


Figure 3: Master-key rotation from MSK to MSK'. **(a)** Rotation triggered at block X. **(b)** Rotation finished at block $X+d$.

matches its organization's policy. On successful verification, the validator's KMS returns to the seed the unwrapping of the share-wrapping key k_i needed to decrypt $\mathsf{Enc}_{k_i}(\mathsf{MSK}_i)$ from persistent storage. Once the seed has decrypted at least T_{Recon} shares, it runs Shamir reconstruction to recover MSK and resumes serving whitelisted enclaves as in Section 3.1.

4 APS Sandbox

Private execution in the pEVM reuses standard EVM bytecode and tooling: contracts run under a modified EVM engine inside the confidential environment described in Section 2, but every state transition is mediated by a sandbox that isolates private state by default. The goal of this design is to let developers port existing smart contracts while limiting accidental leakage: private state never leaves the enclave in plaintext—artifacts crossing the enclave/host boundary are encrypted under keys derived from MSK—and an execution *inspector* intercepts calls, storage accesses, and selected opcodes so policy is enforced at runtime, not only at compile time. Fungible value crosses the boundary between the public Arc ledger and APS exclusively through a bridge precompile (`shield` / `unshield`); ordinary private contracts never read or write public ERC20 balances directly. The remainder of this section describes the sandbox, the bridge, and the access-control

mechanisms a deployer uses to expose state and entry points to other callers.

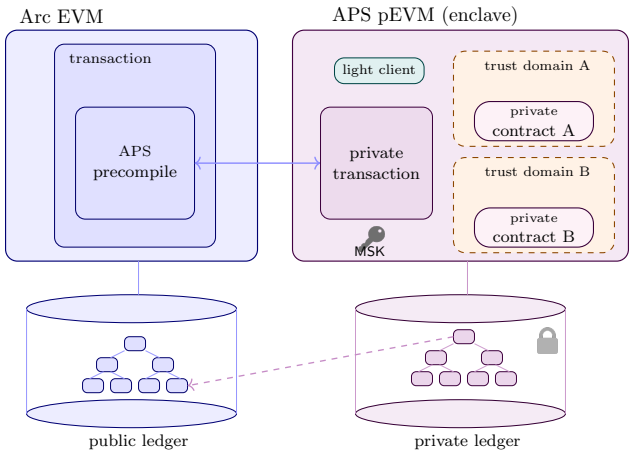


Figure 4: Public Arc EVM composability with the APS pEVM running inside an enclave. Transactions enter through the APS precompile path; distinct trust domains isolate private contracts; during finalization the encrypted private state root is committed to the public Arc ledger so that both ledgers advance in lockstep.

Figure 4 sketches the public Arc EVM stack alongside the pEVM enclave, the precompile-mediated transaction path, and the trust domains that isolate private contracts. Each side maintains its own ledger; finalization commits the encrypted private state root into the public Arc header so the two ledgers reflect a single world state per block. The bridge, deployment, execution, and result-query flows are summarized inline in the subsections below.

At a high level, the pEVM maintains a call stack of frames that separate the contract whose *code* is executing from the contract whose *storage* is active—a distinction essential for `DELEGATECALL` and library patterns. Policy checks compare the active storage context and the function's visibility before allowing loads, stores, or outbound calls. Together with encrypted persistence and filtered outputs, the environment behaves as a contained private EVM whose only sanctioned interface to public Arc custody is the bridge precompile; all other observable touchpoints with the outer chain are precompile-defined entry points and commitments.

4.1 Contract Isolation Model

The sandbox is *deny by default*. Every cross-frame storage access, cross-contract call, and outward observation is checked before it is allowed to take effect. The mechanisms below combine to keep a private contract's state, control flow, and gas usage from leaking to other contracts on APS or to observers on the public Arc chain.

Default deny. Storage reads and writes succeed only when the active storage context and sandbox rules permit them: another contract's slots are not addressable unless the call graph and policy allow that frame. Fungible value crosses the Arc ↔ APS boundary only through `shield` / `unshield` on the bridge precompile (Section 4.3); ordinary `CALL`s never link private state to public Arc ERC20 positions. Static calls cannot mutate storage.

Runtime visibility. Solidity visibility (`public`, `external`, `internal`, `private`) is resolved from deployed metadata and enforced on every `CALL` / `DELEGATECALL`: `internal` and `private` functions cannot be invoked from arbitrary external code unless the call pattern mirrors Solidity's rules (including `DELEGATECALL` for library-style composition). This closes a large class of unintended entry points when third-party bytecode is linked into a confidential application.

Logging and introspection. Contract logs are disabled by default; emitting an event requires an explicit APS precompile so accidental data exfiltration via `LOG` opcodes is avoided. Similarly, `BALANCE`,

`EXTCODE*`, and related probes are masked across trust boundaries (e.g., zero or empty results) so untrusted code cannot fingerprint neighbors. Return data exposed to untrusted observers is encrypted or otherwise sanitized; detailed revert reasons are not surfaced.

Constant-time gas. For all private APS contracts, gas observable outside the confidential execution boundary is constant-time: callers and chain observers see a *fixed* gas figure per invocation, independent of branch-to-branch path variation, eliminating gas-based inference of control flow. The EVM still meters real consumption internally: if execution exceeds the contract's configured allowance, the transaction reverts. Each deployer supplies a per-contract constant gas allowance at deploy time (e.g., via constructor or registration metadata), subject to admin-enforced caps such as the block gas limit and global policy, so deployers tune costs for their workload while the platform bounds DoS and misconfiguration.

4.2 Public State Access Control

The isolation model above governs how a contract relates to its co-resident neighbors. This subsection describes how the same machinery keeps APS separated from the public Arc state and how a deployer opts in to cross-contract reads inside APS.

Isolated execution environment. The pEVM runs as a fully isolated EVM instance backed by its own encrypted database, with no read path into the public Arc ledger. A private contract therefore never silently ingests public state that might leak through storage patterns, gas variation, or control-flow side channels.

Cross-contract access within APS. Access between co-resident private contracts is deny by default and enforced at three complementary layers:

Function visibility. Contracts embed bytecode markers that map each function selector to a Solidity visibility level (`public`, `external`, `internal`, `private`). The pEVM Inspector (implemented as a `revm` inspector hook) intercepts all `CALL`-family opcodes and reverts any invocation that violates the declared visibility. This mirrors Solidity's compile-time model but enforces it at runtime, so crafted calldata cannot reach an entry point the author did not intend to expose.

Trust domains. A contract's admin (initially its deployer) may grant trust to other contracts via `addTrustees`. Without an active trust relationship, opcodes that inspect another contract—`EXTCODESIZE`, `EXTCODEHASH`, `EXTCODECOPY`, `BALANCE`—return zero. Trust is unidirectional: A trusting B does not imply B trusts A. Only the admin can modify trust relationships, and they are revocable at any time. The per-function granularity of the trust grant is described in Section 4.9.

Security rationale. The model enforces *consent-based disclosure*. No contract can observe another's code, balance, or storage unless the target's admin has explicitly opted in via the trust domain, and an admin can revoke that access unilaterally. This preserves deployer sovereignty: the party who created a contract retains exclusive control over who may read its state.

4.3 Cross-Contract Interactions

Cross-environment composability for standard ERC20 value between the public Arc ledger and APS is mediated exclusively by the APS bridge precompile, which exposes two entry points—`shield` and `unshield`—each taking the Arc token address and an amount. Ordinary private contracts do not pull public balances directly; instead, a user (or a transaction bundle on their behalf) invokes the precompile, which moves value through a controlled bridge cycle: the Arc-side asset is burned or locked on shield and later minted or unlocked on unshield, while the APS-side mirror is minted or burned accordingly. Nothing in the architecture prevents richer cross-environment interactions—public and private execution share the same block—but designing private contracts whose ordinary query and execution paths do not leak the very state they are meant to protect is an open research problem. We continue to develop reference contracts for common cases (e.g., ERC20) and to explore the broader design space with the community.

Shielding (Arc → APS). A call to `shield` consumes the user's ERC20 on Arc by burning it or locking it in precompile-controlled custody or a dedicated lock contract, depending on the token's bridge mode, and ensures a corresponding Private ERC20 mirror exists for that Arc token address. The precompile then

instructs the mirror to mint the same amount to the user's APS-side address, increasing private supply in lockstep with the Arc-side burn or lock.

Unshielding (APS → Arc). A call to `unshield` burns the Private ERC20 amount from the user on APS (reducing mirror supply), then restores the corresponding ERC20 to the user's Arc address by minting it back or unlocking collateral on Arc.

Mirror security. The Private ERC20 `mint` path is restricted (e.g., `onlyOwner`) so that only the bridge precompile can issue supply; users burn through the precompile-driven `unshield` path. This keeps APS-side circulation tied to Arc-side burns or locks and preserves the fungibility of the represented asset.

Porting contracts. Existing token logic moved into APS should expose `mint` / `burn` hooks compatible with precompile-controlled supply if it is to participate in this bridge model; bespoke assets may follow the same Private ERC20 pattern. Mappings between Arc token addresses and APS mirror instances are fixed by deployment policy so that linkability is unambiguous and verifiable.

Intra-APS calls. Standard `CALL` / `DELEGATECALL` semantics still apply *among* private contracts inside APS (e.g., transferring Private ERC20 tokens or composing local logic). The design constraint is solely that integration with the public Arc ERC20 currency layer for generic tokens is centralized on `shield` and `unshield`, not on ad-hoc cross-environment calls into custody.

4.4 Shield and Unshield

Figure 5 renders the bridge as two accounting flows. The same bridge precompile coordinates both directions so that Arc-side accounting and Private ERC20 supply stay synchronized, while only the `unshield` public release is delayed by one block.

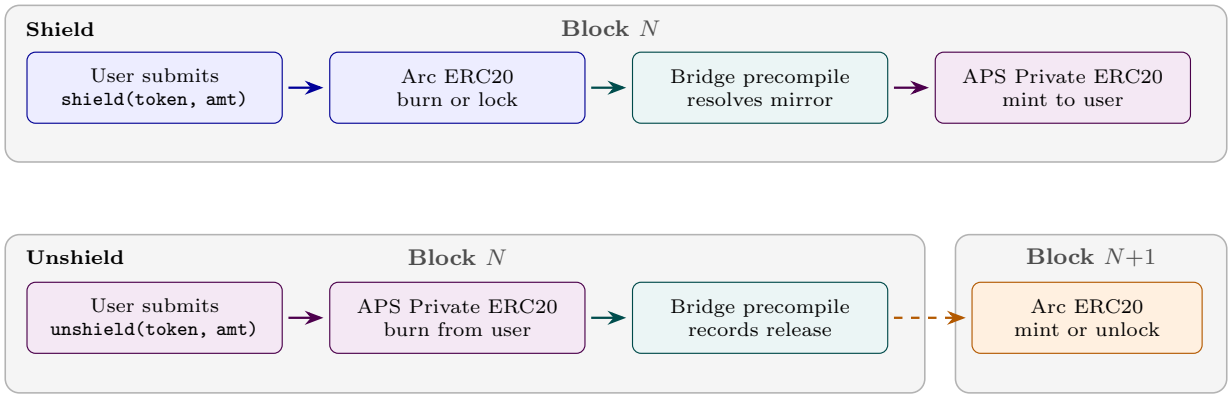


Figure 5: `shield` and `unshield` bridge flows. Shielding consumes public Arc ERC20 value and mints the Private ERC20 in the same block. Unshielding burns the Private ERC20 in block N, and mint-or-unlocks public ERC20 value in block $N+1$ to ensure block N is finalized.

Shield. The `shield` path is atomic: the Arc-side burn or lock amount is already publicly observable on Arc, so the corresponding APS-side mint reveals nothing beyond what an external observer can read from the public ledger anyway. Burn-or-lock and mint can therefore safely finalize in the same block N without leaking additional information.

Unshield. In the `unshield` path, the APS-side burn occurs in block N, but the Arc-side ERC20 mint-or-unlock only occurs after block N is finalized, i.e in block $N+1$. Without finalization, a validator host can learn information about other private transactions by reordering and replaying the `unshield` tx.

4.5 Private Contract Deployment

Deploying a private contract follows a short request–execute–register path. The client supplies deployment parameters in plaintext to a APS-facing RPC or enclave endpoint, which drives a confidential `CREATE` inside

APS and registers the new deployment with the contract registry. Any private deployment result is retrieved through the signed results query path described below. Figure 6 summarizes the deployment flow.

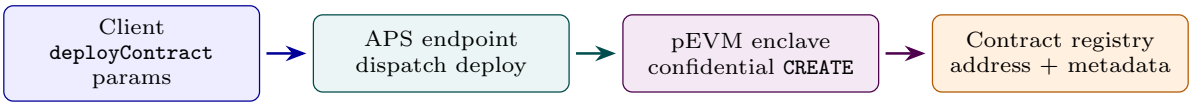


Figure 6: Private contract deployment. The deploy endpoint drives `CREATE` inside the pEVM enclave and records the new contract in the registry; deployment outputs follow the same signed result-query path as private transaction outputs.

4.6 Private Contract Execution and Result Retrieval

A private execution separates into two flows: the block-time execution path in Figure 7 and the signed post-finalization result query in Figure 8. Both are mediated by the Arc-facing APS precompile or an enclaved RPC endpoint and use the same RFC 9180 Base-mode hybrid suite (X-Wing KEM, HKDF-SHA256, AES-256-GCM) for confidentiality on the wire.

Encrypt the call. The client first requests `globalPublicKey` through the APS precompile, which forwards into the pEVM and returns the global X-Wing KEM public key. The wallet authorizes the `PrivateTx` as EIP-712 typed data and hybrid-encrypts the resulting bundle to that key. The ciphertext is ABI-encoded as bytes for `executePrivateTx`.

Execute in block N. The precompile verifies the encrypted payload and dispatches the ciphertext bytes through `executePrivateTx` into the pEVM, where the enclave decapsulates the bundle, decrypts the calldata with AES-256-GCM, verifies the EIP-712 signature on `PrivateTx` to recover `msg.sender`, and runs the confidential `CALL`. Before persisting the receipt association `(txHash, APSTxOutput)`, the pEVM re-seals `APSTxOutput` under the same hybrid suite to the user's receiver X-Wing public key.

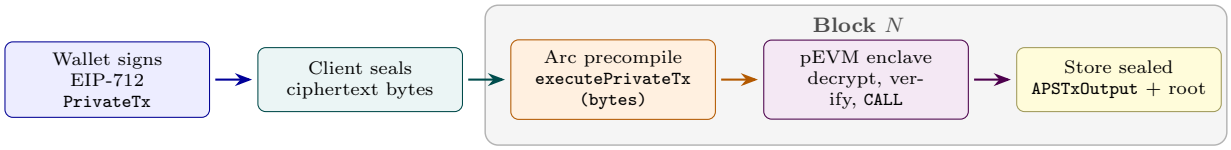


Figure 7: Private transaction execution. The EIP-712-signed `PrivateTx` is sealed to the APS global key, submitted as ciphertext bytes to `executePrivateTx`, executed inside the pEVM in block N, and retained only as sealed output plus the encrypted private-state commitment.

Retrieve on block N finalization. The client (or wallet) fetches `results(txHash, sig)` from an enclaved RPC node once block N, the block containing the transaction, has finalized and the light client has committed the authenticated private state. The signature comes from the same key that signed the original transaction; the RPC enclave verifies that signature before looking up the sealed receipt. The wallet receives `APSTxOutput` ciphertext and completes X-Wing decapsulation followed by AES-256-GCM decryption with the receiver secret key. The same payload can be re-fetched by `txHash` for as long as the enclave retains the receipt.

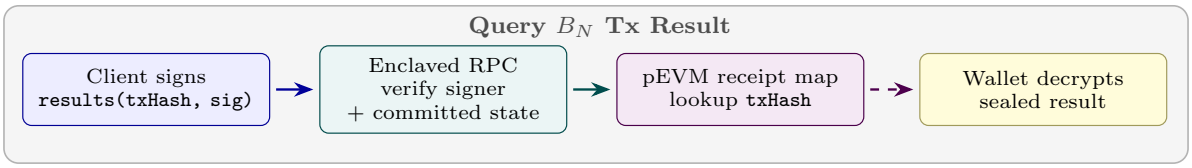


Figure 8: Signed results query. On block N finalization, the client authorizes `results(txHash, sig)` with the original signing key; the enclaved RPC serves the sealed receipt only from committed private state, and the wallet decrypts it locally.

4.7 Security Properties

We summarize the privacy guarantees the sandbox is intended to provide:

- **Enclave Execution.** APS depends on the Enclave technology for attested private execution.
- **Storage confidentiality.** Persisted artifacts are encrypted using MSK under randomized nonces before crossing the enclave/host boundary; without valid execution of finalized block and the policy checks above, plaintext state is never exposed to the public.
- **Cross-EVM interactions.** Public Arc and APS interactions are confined to the `shield` / `unshield` bridge.
- **Access Control.** Function entry points respect enforced visibility, limiting the information flow that would otherwise arise from calling functions across isolation boundaries.
- **Side Channel Leakage.** Fixed execution gas is used for private transactions to reduce public leakage. Furthermore, private txs cannot be simulated, and results can only be retrieved on block finality. However, the sandbox does not claim hardware-level constant resource usage (time, memory etc.), which may still be visible to the host.

4.8 Authorization Mechanisms

All external entry points into APS are routed through a single APS precompile. Recall from Section 4.2 that trust domains govern *which* contracts may interact; the precompile additionally governs *how* external callers reach APS in the first place. This subsection describes the query authorization endpoints that govern read access, and the trust domain API that governs cross-contract interaction.

Query authorization. Three read-only query endpoints provide graduated caller-identity guarantees. Each is exposed via the APS precompile; an EOA-only gate at the precompile level prevents contracts on the public chain from invoking them directly.

`ethCall`**.** The simplest query mode. The caller inside the pEVM is set to `address(0)`, signaling to the target contract that the query is unauthenticated. The result is returned as plaintext, and neither the request nor the response is encrypted. This mode is suitable for `public` or `external` view functions that do not require caller identity; contracts that gate sensitive data on `msg.sender` will see the zero address and can deny access accordingly.

`ethCallEncrypted`**.** Semantically identical to `ethCall`—the caller remains `address(0)` and the query is unauthenticated to the target—but the request payload is a hybrid ciphertext in RFC 9180 Base mode (X-Wing KEM, HKDF-SHA256, AES-256-GCM) to the node's global X-Wing public key. The encrypted payload decrypts to a structure containing a *receiver public key*, the target address, and the calldata. The response is sealed again with the same hybrid suite to the receiver public key before being returned. This mode protects query contents and results on the wire while leaving the caller anonymous to the target contract.

`ethCallAuthorized`**.** The authenticated query mode. The caller's identity is cryptographically verified via EIP-712, and the pEVM executes the call with `msg.sender` set to the verified address. This allows

the target contract to enforce `msg.sender`-based access control on view functions, and the pEVM Inspector evaluates function visibility against the real caller.

The encrypted payload decrypts to an `AuthQueryData` structure comprising three parts:

- `InnerAuthQueryData`: the receiver public key (for response encryption), the target contract address, the asserted caller address, the ABI-encoded calldata, and an `AuthExpiration` specifying a valid block range (`validAfter`, `validBefore`).
- `AuthDomain`: the EIP-712 domain separator fields—name, version, chain ID, verifying contract, and salt.
- `signature`: an ECDSA signature over the EIP-712 typed-data hash of `InnerAuthQueryData` under the `AuthDomain`.

Verification proceeds through four checks, any of which causes the call to revert:

1. *Chain ID.* The domain's `chainId` must match the current chain, preventing cross-chain replay.
2. *Block range.* The current block height must satisfy `validAfter` $<$ block_height $<$ `validBefore` (both exclusive), bounding the authorization's temporal validity and preventing indefinite replay of stale signatures.
3. *Target binding.* The domain's `verifyingContract` must equal the target address in the inner payload, preventing an authorization signed for contract A from being replayed against contract B.
4. *ECDSA recovery.* The EIP-712 signing hash is computed and the signer is recovered; it must match the asserted caller address.

On success, the query executes with the verified caller and the output is encrypted to the receiver public key.

Query mode summary.

Mode	Caller identity	Mutates state	Output
`ethCall`	Anonymous	No	Plaintext
`ethCallEncrypted`	Anonymous	No	Encrypted
`ethCallAuthorized`	EIP-712 verified	No	Encrypted

Trust domain management. Trust relationships (introduced in Section 4.2) are managed through five primary precompile functions:

Function	Returns	Mutates
`setFunctionPolicy(address truster, bytes4[] selectors, Policy policy)`	`bool`	Yes
`addTrustees(address truster, address[] trustees, bytes4[] selectors)`	`bool`	Yes
`removeTrustees(address truster, address[] trustees, bytes4[] selectors)`	`bool`	Yes
`isTrusted(address truster, address trustee, bytes4 selector)`	`bool`	No
`listTrustees(address truster, uint256 page, uint256 pageSize)`	`address[]`	No

Admin authorization. The mutating operations `setFunctionPolicy`, `addTrustees`, and `removeTrustees` require the transaction sender to be the *admin* of the truster contract. The admin is initially set to the deployer address at contract creation time and is recorded in the contract's on-chain metadata. Admin transfer follows a two-step handshake (`proposeAdmin` → `acceptAdmin`, with `cancelProposedAdmin` to abort), ensuring that the receiving party actively consents. If the caller is not the current admin, the precompile reverts.

Storage. Trust state is maintained as a directed graph: each truster address maps to a set of trustee addresses, each carrying a bitmask that defines which selectors are trusted. The relationship is asymmetric—if A trusts B, B does not automatically trust A. A convenience operation for mutual trust adds both directions explicitly.

Runtime enforcement of trust domains, function visibility, introspection masking, and storage isolation is described in Sections 4.1 and 4.2.

4.9 Trust Domain Function Access Policy

Trust domains (Section 4.2) govern *which* contracts may interact; per-function access policies govern *how* each function in a contract may be reached. A call must satisfy both the function's visibility rules and its access policy before the pEVM Inspector permits execution.

Policy states. Every function selector registered in a contract's on-chain metadata carries an `AccessPolicy` drawn from three values:

Policy	Semantics
`Open`	Any caller may invoke the function (after visibility checks).
`Restricted`	Only callers granted access to this specific selector may invoke it.
`Locked`	The function is unconditionally disabled; all calls revert.

Trust is bound to a specific selector, not to the contract as a whole: a contract may grant address A access to `balanceOf` without granting access to `transfer`. The trust domain API exposes this granularity through `addTrustees` and `removeTrustees` (Section 4.8).

Access grants are stored as a bitmask per (contract, grantee) pair. Each contract's function selectors occupy fixed positions determined at deployment; bit i in a grantee's bitmask corresponds to the i-th selector. `addTrustees` sets bits; `removeTrustees` clears them. This representation keeps bulk operations compact—granting access to all selectors is a single bitmask write—and keeps per-trustee storage constant regardless of how many selectors are authorized.

Safety. When a contract is deployed, every function is `Restricted` and no grantee bitmasks exist. The contract is effectively locked until the admin explicitly grants access or sets a selector to `Open`.

Granularity. Policies are set per 4-byte selector or in bulk across all selectors on a contract. Either operation requires admin authority over the target contract (Section 4.8). Because policies bind to individual selectors rather than to the contract as a whole, an admin can open a subset of the interface while keeping sensitive operations `Restricted` or `Locked`—without modifying or redeploying the source contract.

Enforcement. The pEVM Inspector evaluates access policy on every `CALL`-family opcode, after visibility checks pass. For `Open`, no further check is required. For `Restricted`, the Inspector looks up the caller's bitmask for the target contract and checks the bit corresponding to the invoked selector; if the bit is unset or no bitmask exists, the call reverts. For `Locked`, the call reverts unconditionally. This two-layer evaluation—visibility first, then access policy—means that a function marked `internal` at the Solidity level and `Open` at the policy level remains unreachable from an external call, because visibility blocks it before the policy is consulted.

UX. To simplify deployment UX, the trust domains could be generated from the source code of the smart contracts by a static analyzer tool that gathers the cross-contract invocations, and allows the Trust Domain admin to easily set the right permissions on deployment.

Example ERC20 with segregated query access. Consider an admin who deploys a standard OpenZeppelin ERC20 token into APS without source modifications.

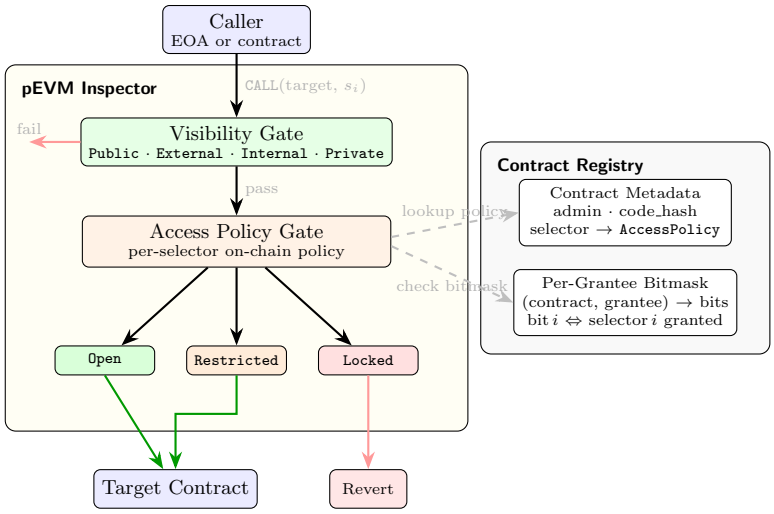


Figure 9: Trust domain function access policy architecture. On every `CALL`-family opcode the pEVM Inspector evaluates two sequential enforcement layers: a *visibility gate* (Solidity-level modifiers recovered from per-contract bytecode markers) followed by an *access policy gate* (per-selector on-chain policy stored in the contract registry). `Open` selectors proceed without further check (green path); `Restricted` selectors require a set bit in the caller's per-contract bitmask (green if granted, otherwise reverts); `Locked` selectors revert unconditionally (red path). Dashed arrows denote data lookups against the contract registry.

The result is a token whose transfers are permissionless, whose raw balance reads are restricted to explicitly trusted contracts, and whose authorized query path requires caller authentication—all achieved without editing the ERC20 bytecode. Any contract can be wrapped with an independent query-auth facade that mediates read access on behalf of authenticated callers.

5 APS Ciphersuite

This section specifies the ciphersuites and parameters used by APS. The concrete key-derivation tables (HPKE seeds and keypair, persistence keys, per-block state-root key) are collected in Appendix A to keep the body focused on the design choices.

5.1 Ciphersuite Selection Criteria

We applied four criteria when selecting our ciphersuites:

- **NIST approved.** Many Arc customers are financial organizations subject to government regulation. We use NIST-approved ciphersuites wherever possible.
- **Post-quantum privacy.** APS must protect customers against harvest-now–decrypt-later attacks.
- **Battle tested.** We adopt a hybrid approach: whenever a newer post-quantum algorithm is used, we pair it with a classical ciphersuite so that a vulnerability in the post-quantum component does not on its own compromise security.
- **EVM compatibility.** We want our customers to use standard SDKs and tooling, and to simplify the experience of cross-chain applications.

The goal of using NIST-approved cryptographic algorithms occasionally conflicts with our commitment to hybrid constructions when introducing post-quantum algorithms. The existing hybrid algorithms

X-Wing [CSW25, BCD+24] and X25519MLKEM768 [KKWS25] both use the X25519 elliptic curve for the classical half of key agreement. As of this writing, NIST has chosen not to standardize key agreements based on X25519, regarding "new schemes as secondary in the ongoing transition to post-quantum cryptography" [oSN25]. Both X-Wing and X25519MLKEM768 have been widely vetted by industry and are in the process of being standardized by the IETF; we adopt them in APS as the best option available today.

5.2 Signatures

Arc and APS rely on the same signature schemes for transactions and consensus.

Transaction signatures. Arc and APS use the Ethereum ECDSA signature scheme for transactions. The blockchain also exposes a post-quantum signature verification function for SLH-DSA-SHA2-128s; a full post-quantum migration is planned.

Consensus signatures. Arc manages consensus on behalf of the pEVM by including its encrypted state root in each block. The consensus layer currently uses Ed25519 signatures, with a planned migration to BLS aggregate signatures for efficiency. Because both Ed25519 and BLS are elliptic-curve based, the consensus layer will eventually need to migrate to post-quantum signatures as well. That migration is not trivial because of the performance impact and is therefore deferred until a clearer quantum threat materializes: consensus is the layer least vulnerable to early-stage quantum attacks, since the window during which an attacker could inject a malicious block is short. Hash-based post-quantum aggregate signatures such as XMSS [CAD+20] and related hash-based constructions [DKKW25] are the most promising candidates today.

Enclave attestation. APS relies on AWS Nitro Enclave attestation to identify enclaves to one another and to each organization's KMS. AWS currently supports only elliptic-curve–based attestation. Once an enclave identifies itself, however, it communicates over post-quantum TLS 1.3, protecting all data in transit against harvest-now–decrypt-later attacks.

5.3 Secure Communication

All inter-component communication runs over TLS 1.3 with X25519MLKEM768 [KKWS25] wherever supported. As of this writing, NIST has standardized ML-KEM768 [oSNA+24], and the hybrid algorithm X25519MLKEM768 is in the process of being standardized by the IETF and has been widely adopted. Arc tracks the standardization progress and updates its TLS configuration to match industry best practice.

6 Evaluation

We evaluate APS on an AWS Nitro Enclave deployment. Each validator allocates half of its 32 virtual CPUs to the public execution layer and consensus, and the other half to the privacy enclave. We use the same load-testing suite that benchmarks the public Arc network, which lets us compare performance directly.

Public transactions execute outside the enclave and achieve the same throughput as the public Arc baseline. Since private transactions are executed through the precompile, and can't directly interact with public state, they can be executed in parallel to public transactions. The full batch of private transactions are sent to the enclave, where APS parallelly decrypts and verifies their signatures. Then, they are sequentially executed against the private state. Finally, the updated encrypted private state root is returned to the host.

We achieve a private transfer throughput of 1070 TPS. With a 500 ms block time, APS executes a batch of 535 private transfers with end-to-end $p99$ batch execution latency of about 50 ms. Table 1 breaks down this batch execution latency. For comparison, the public Arc network achieves a 2800 TPS in the same setting. This demonstrates that APS achieves a performance for private transactions that is comparable to public transactions. Given that the decryption and verification can be parallelized as needed, the sequential EVM execution and consensus network propagation is the bottleneck in the steady state. Our design supports replacing EVM with other runtime that might support parallelization at the core.

Stage	Latency (ms)
HPKE/X-Wing Decryption (parallel)	20.26
ECDSA Signature Verification (parallel)	15.21
Batch Transfer Transport over vsock	11.37
EVM Execution (sequential)	4.19
Other	< 1
End-to-End Batch Total	51.04

Table 1: Breakdown of the batch execution latency within a block time of 500 ms for a batch of 535 private transfers.

7 Conclusion

Arc Privacy Sector (APS) shows that confidential smart-contract execution can be integrated into a public blockchain without forcing developers to abandon the EVM programming model or accept asynchronous bridge assumptions. By running a pEVM inside attested enclaves, committing an encrypted private state root in every Arc block, and exposing private execution through controlled precompiles and sandbox policies, APS lets public and private state finalize together while keeping transaction payloads and contract storage confidential. Its cryptographic design provides hybrid post-quantum security for private transactions. Overall, APS provides a pragmatic path to a synchronous public-private EVM.

References

[AKSL18] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *NDSS 2018*. The Internet Society, February 2018.

[Apt26] Aptos. Confidential Asset, 2026. https://aptos.dev/build/smart-contracts/confidential-asset.

[Avi25] Avishay Yanai, Meital Levy, Hila Dahari-Garbian, Mike Rosulek. gcevm. https://docsend.com/view/edxd4srjfv37z762, 2025. Online; accessed 2025.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443. Springer, Cham, February 2020.

[BBLW22] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[BC25] Richard Barnes and Deirdre Connolly. Post-quantum and post-quantum/traditional hybrid algorithms for hpke. https://datatracker.ietf.org/doc/draft-ietf-hpke-pq/, 2025. Online; accessed 2025.

[BCD+24] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, and Bas Westerbaan. X-Wing: The hybrid KEM you've been looking for. Cryptology ePrint Archive, Report 2024/039, 2024.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

[BKM18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[CAD+20] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. Recommendation for stateful hash-based signature schemes. https://csrc.nist.gov/pubs/sp/800/208/final, 2020. Online; accessed 2025.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[Cir26] Circle Internet Financial. Malachite: Flexible BFT Consensus Engine in Rust. https://github.com/circlefin/malachite, 2026. GitHub repository; accessed 2026.

[CSW25] Deirdre Connolly, Peter Schwabe, and Bas Westerbaan. X-wing: general-purpose hybrid post-quantum kem. https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/06/, 2025. Online; accessed 2025.

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy*, pages 185–200. IEEE Computer Society Press, June 2019.

[DDE+23] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-FHE using noise flooding. Cryptology ePrint Archive, Report 2023/815, 2023.

[Dis24] Distributed Systems and Storage Laboratory at UIUC. TEE-Rollups: Fixing Access Patterns in TEE-based Smart Contracts with Off-chain Computing, 2024. medium.com/@dsl_uiuc/tee-rollups.

[DKKW25] Justin Drake, Dmitry Khovratovich, Mikhail Kudinov, and Benedikt Wagner. Hash-based multi-signatures for post-quantum ethereum. https://eprint.iacr.org/2025/055, 2025. Online; accessed 2025.

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[Fhe26] Fhenix. Fhenix Documentation, 2026. https://cofhe-docs.fhenix.zone/.

[JLLJ+24] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. SGXonerate:finding (and partially fixing) privacy flaws in TEE-based smart contract platforms without breaking the TEE. *PoPETs*, 2024(1):617–634, January 2024.

[KGM19] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *NDSS 2019*. The Internet Society, February 2019.

[KKWS25] Kris Kwiatkowski, Panos Kampanakis, Bas Westerbaan, and Douglas Stebila. Post-quantum hybrid ecdhe-mlkem key agreement for tlsv1.3. https://datatracker.ietf.org/doc/draft-kwiatkowski-tls-ecdhe-mlkem/, 2025. Online; accessed 2025.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.

[MIS+22] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. QuORAM: A quorum-replicated fault tolerant ORAM datastore. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3665–3682. USENIX Association, August 2022.

[MPC+18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy*, pages 279–296. IEEE Computer Society Press, May 2018.

[Nat23] National Institute of Standards and Technology. Advanced Encryption Standard (AES). FIPS PUB 197, 2023.

[Net25] Secret Network. Secret Network, 2025. https://scrt.network/.

[Net26] Phala Network. dStack: Open Source Confidential Computing, 2026. https://phala.com/dstack.

[oSN25] National Institute of Standards and Technology (NIST). Nist proposes to update sp 800-56a and revise sp 800-56c. https://csrc.nist.gov/news/2025/proposal-for-sp-800-56-reports, 2025. Online; accessed 2025.

[oSNA+24] National Institute of Standards, Technology (NIST), Gorjan Alagic, Quynh Dang, Dustin Moody, Angela Robinson, Hamilton Silberg, and Daniel Smith-Tone. Module-lattice-based key-encapsulation mechanism standard, 2024-08-13 04:08:00 2024.

[Pro26] Oasis Protocol. Sapphire ParaTime, 2026. https://docs.oasis.io/build/sapphire/.

[SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious memory primitives from intel SGX. In *NDSS 2018*. The Internet Society, February 2018.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SMCM25] Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya. Treebeard: A scalable and fault tolerant ORAM datastore. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security 2025*, pages 7583–7602. USENIX Association, August 2025.

[Sol26] Solana. Confidential Transfer, 2026. https://solana.com/docs/tokens/extensions/confidential-transfer.

[Sun26] Sunscreen. Sunscreen Blog, 2026. https://blog.sunscreen.tech/.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[TAC25] TACEO. Presenting Merces: A Confidential Stablecoin Payments Demo, 2025. https://core.taceo.io/articles/confidential-token-transfers-demo/.

[Ten26a] Ten. Ten: Encrypted Ethereum Layer 2, 2026. https://github.com/ten-protocol.

[Ten26b] Tendermint. Tendermint Core, 2026. https://docs.tendermint.com/master/nodes.

[WGSH22] Philipp Winter, Ralph Giles, Moritz Schafhuber, and Hamed Haddadi. Nitriding: A tool kit for building scalable, networked, secure enclaves. *arXiv preprint arXiv:2206.04123*, 2022.

[Woo25] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Yellow Paper, 2025.

[XCZ+23] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VeriZexe: Decentralized private computation with universal setup. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 4445–4462. USENIX Association, August 2023.

[Zam25] Zama. Zama KMS: A Pure Rust Implementation of a Threshold Key Management System for TFHE, 2025. https://github.com/zama-ai/kms.

A Key Derivation and Ciphersuite Parameters

This appendix specifies the concrete key-derivation parameters that the APS ciphersuite (Section 5) uses to instantiate HPKE for user transactions, persisted-state encryption, and per-block state-root encryption committed to each block.

A.1 Keys Derived from MSK

APS uses a 32-byte master secret key MSK to derive all other keys as well as pseudorandom seeds and values. Table 2 lists the derived keys and their purpose.

Name	Type	Purpose
(EPK, ESK)	X-Wing HPKE	Encrypt transactions to APS
Persistence key	AEAD-AES-256-GCM-SIV	Encrypt persisted state (one key per emission)
State-root key	AEAD-AES-256-GCM-SIV	Encrypt the state root in the block header (one key per block)

Table 2: Keys derived from MSK and their purpose.

A.2 Hybrid Public Key Encryption

Users send encrypted transactions to APS using the global HPKE key pair (EPK, ESK). After the block is finalized, users can query an enclaved RPC node for their transaction results; the response is encrypted back to the user's ephemeral public key under HPKE. APS uses a post-quantum-secure configuration of HPKE [BC25] with context-specific info and aad string constants—GLOBAL_KEY_INFO and GLOBAL_KEY_AAD when receiving encrypted transactions from users, USER_INFO and USER_AAD when sending encrypted responses back. Parameters and constants are summarized in Tables 3 and 4.

Parameter	Value
MODE	Base
KEM	X-Wing KEM
KDF	HKDF-SHA256
AEAD	AES-256-GCM

Table 3: HPKE parameters.

Constant	String value
GLOBAL_KEY_INFO	APS global public key info
GLOBAL_KEY_AAD	APS global public key aad
USER_INFO	APS user info
USER_AAD	APS user aad

Table 4: HPKE info and aad string constants.

A.3 HPKE Key Derivation

The global HPKE key pair (EPK, ESK) is derived from MSK in two steps: HKDF-SHA256 produces a 96-byte X-Wing seed (Table 5), and the X-Wing key-generation routine produces the keypair from that seed (Table 6). The Version parameter allows an unambiguous representation for changing the key derivation in the future.

A.4 Encrypting Persisted State

Inside the enclave, the pEVM holds private contract state and the private state root in plaintext. Encryption applies only when state crosses the enclave/host boundary for persistence on the untrusted host. The artifacts that cross are periodic full snapshots of the private database and per-block write-ahead logs that allow recovery after an enclave restart. The pEVM treats each such artifact as a single *emission* and encrypts it with AEAD-AES-256-GCM-SIV under a key derived from MSK via HKDF-SHA256, using a 16-byte salt drawn fresh from the enclave's CSPRNG at the start of the emission (Table 7). The AEAD nonce is 12 bytes of fresh randomness drawn for each AEAD operation. The salt and the per-operation nonce travel alongside the ciphertext, so any validator can re-derive the key from its own copy of MSK and decrypt the artifact

Parameter	Value
Version	4-byte `uint32(0)`
Input key material	32-byte MSK
Info	`HPKE XWing: contract-key-seed-derivation`

Table 5: HKDF-SHA256 parameters for deriving the X-Wing seed.

Parameter	Value
Seed	96-byte X-Wing seed
Info	`HPKE XWing: contract-keypair-derivation`

Table 6: X-Wing HPKE key-pair derivation parameters.

when restoring from host-supplied data. Tampering, truncation, or fabrication by the host fails the AEAD's authentication check on decryption.

Parameter	Value
Input key material	32-byte MSK
Salt	16 random bytes per emission
Info	`symmetric-key: persistence emission`

Table 7: HKDF-SHA256 parameters for the per-emission persistence key.

Because the salt is drawn fresh per emission and the AEAD nonces are fresh per operation, identical plaintexts encrypted at different times produce unrelated ciphertexts. The host cannot infer plaintext equality, structure, or recurrence across persistence events.

A.5 Encrypting the State Root

The pEVM's state root commits to the private state after block execution and must remain confidential to non-enclave nodes. During block proposal, the enclave encrypts it under a per-block key, and the resulting ciphertext, authentication tag, and nonce together form the header's `private_state_root` field.

A node validating the block independently computes its local plaintext root inside its own enclave, re-encrypts it under the nonce from the header bundle, and compares the result against the header's ciphertext and tag. A mismatch indicates that the validator's plaintext root differs from the proposer's.

The pEVM encrypts the state root with AEAD-AES-256-GCM-SIV. The 32-byte key is derived per block from MSK via HKDF-SHA256, using the 8-byte block height as salt (Table 8). The AEAD nonce is 12 bytes of fresh randomness drawn per encryption and travels alongside the ciphertext and authentication tag in the header bundle. The AAD is a fixed algorithm-version constant known by convention to both producer and consumer; it is not transmitted.

Parameter	Value
Input key material	32-byte MSK
Salt	8-byte block height
Info	`symmetric-key: state-root-encryption-key-derivation`

Table 8: HKDF-SHA256 parameters for the per-block state-root encryption key.

Fresh randomness in the nonce decouples encryptions from one another, so the proposer can query the state root as often as needed during block construction without leaking equality information across queries. The per-block key decouples blocks from one another, so unchanged private state across consecutive blocks produces no observable signature in the header.