Monad Private Settlement Domains.

White paper.

Jan Camenisch, Ariq Chowdhury, Andrei Constantinescu, Sourav Das, Kai Jun Eer, Aashish Welling

ABSTRACT

Public blockchains offer a global, highly resilient infrastructure for settling financial and other transactions that is available to anyone at any time. Institutions and enterprises have recognized this and have started to make substantial use of this infrastructure. However, the lack of privacy, sovereignty, and control severely limits the applicability of public blockchains to a narrower set of use cases and has driven institutions to private permissioned chains. Permissioned chains unfortunately sacrifice composability, liquidity, financial efficiency, and ease of operation. Monad Private Settlement Domains bring the two worlds together: they are rooted in the Monad blockchain and operated by designated operators, with state and transactions being private.

01 Introduction

Institutions and enterprises are already using public blockchains, but only where nothing needs to be private. Regulated stablecoins and tokenized funds settle at scale because their issuers have no onchain confidentiality requirement: reserves are attested publicly, supply is meant to be verifiable, and what is genuinely sensitive never leaves the issuer's own systems. The cost of lacking privacy falls on holders, a cost the first wave of users tolerated but enterprises can only afford in cases where confidentiality is not critical. Almost everything else a bank, fund, or asset manager does (such as holding client balances, paying staff and suppliers, quoting prices, managing treasury, maintaining correspondent relationships, and deciding compliance outcomes), consists of information they are obliged or determined to withhold from competitors, customers, and frequently the platform itself. A public chain will by nature of its construction publish all of it. Confronted with this, institutions and enterprises typically opt for a permissioned ledger run by themselves or by a consortium. This departure is expensive: liquidity fragments across ledgers, and every connection to the outside world or to other blockchains has to be built as a bespoke bridge, recreating the very problem blockchains set out to solve. Worse, someone needs to operate and maintain the shared infrastructure: a party with power over settlement whose remit, governance, and liability no project has yet managed to define.

To address this gap between private and public chains, we introduce Monad Private Settlement Domains (PSD), a sovereign execution space embedded in the Monad blockchain. Each PSD provides a private, permissioned, customizable execution environment that is operated by a set of parties selected by the creator of the domain. Execution, state, and transactions are fully confidential and, unless explicitly approved by specified policies, cannot be read by any party other than the operators. Furthermore, PSDs can be implemented so that the operators have no access either. State settlement, sequencing, and (confidential) cross-domain messaging, however, remain on the Monad network. All PSDs thus share a single ordering system provided by the public Monad blockchain, enabling interaction between smart contracts running in any domain. Confidentiality, per-participant and per-jurisdiction control, and composability with everything else can now be achieved on a public blockchain.

Monad Private Settlement Domains are enabled by a *hub* smart contract deployed on the Monad blockchain and corresponding *spoke* contracts on each PSD; our design therefore requires no protocol change. (The contract could be replaced by a precompile, e.g., to increase efficiency.) A PSD is created by sending a creation transaction to the hub specifying the PSD's configuration, including the operators who will operate it, a public encryption key, and any required access restrictions. The design is agnostic to whether these operators are also validators of the Monad blockchain. The configuration can be changed at any time by the creator (or by other parties as specified in the domain's configuration). For instance, one can swap out an operator to address a hardware failure. Transactions for a Monad Private Settlement domain are encrypted under its public key and sent to the hub as ordinary transactions. After a Monad Private Settlement domain is created, smart contracts can be deployed, where the deployment can be gated by an allowlist specified at creation. Smart contracts on Monad Private Settlement domains can be addressed globally, i.e., they can be called by users (externally owned accounts) and by smart contracts running on the public Monad blockchain or in other Monad Private Settlement domains.

All transactions for PSDs are sequenced on the Monad blockchain. The user encrypts the transaction for the PSD and wraps it in an ordinary Monad transaction for the hub contract. When called with such a wrapping transaction, the hub retrieves the encrypted transaction and adds it to the input queue of the recipient PSD. The hub can also apply backpressure if a domain is processing transactions too slowly. This backpressure can have different forms such as failing all transactions for a PSD or temporarily raising the domain's transaction fees (similar to Monad's base fee). The operators of a Monad Private Settlement domain periodically pull from the hub the transactions destined for the Monad Private Settlement domain they operate. The pulling interval can be in sync with the Monad block rate but

does not have to be. As all domain transactions are sequenced on the Monad network, PSD operators do not run consensus themselves. They do, however, produce a block that their execution layer handles. In particular, they decrypt the transactions pulled from the hub and form a block ready for execution. Upon executing the block, the operators obtain the new state and a list of messages that the executed contracts on the PSD seek to send to contracts on the main domain and on other PSDs (in the second case, the messages also get encrypted for the destination domain). Messages are how calls between contracts on different domains are made possible. The operators attest that these results are correct while keeping the results themselves secret. To this end, they salt and Merklize the state and the list of messages into two different Merkle trees, and attest to their new roots in the hub. The exact attestation procedure depends on how the Monad Private Settlement domain is realized, e.g., a multi-signature by the domain operators in a correctness-by-committee design, or a zero-knowledge proof in a zkVM-based one. The attestation, the new state root, and the new message root are sent by one or more of the operators as a transaction to the hub, which verifies the attestation and, if correct, accepts the new state and message roots. This acceptance allows the operators to then submit each of the domain's outbound messages, together with a proof that it is contained in the message tree, as a transaction to the hub. The hub verifies the inclusion proof, checks for replays and routes the message to the recipient's inbox (either on a Monad Private Settlement domain or on the main network). This messaging infrastructure enables the composition of smart contracts across Monad Private Settlement domains and the main network (cf. Figure 1), thereby tapping into liquidity on the Monad network and avoiding fragmentation of liquidity across different domains.

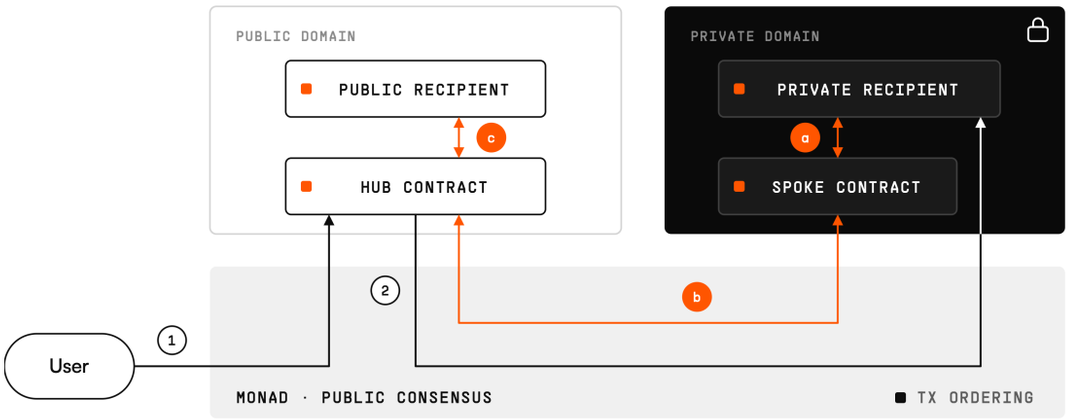


FIG. 1 Transaction flows involving the public domain and a private domain. Steps 1 and 2 depict a flow (black), where a user sends a transaction to a contract on a private domain. Steps a to c depict flows of cross-domain contract calls. Because these calls are necessarily asynchronous, they are realized through message-sending and -receiving calls to and from the hub and spoke contracts. The messages get passed between the hub and spoke contracts by our protocol.

Because state is private and both domain transactions and cross-domain messages are encrypted, flows that stay within a single Monad Private Settlement Domain or move between Domains remain private. However, when transactions tap into Monad network liquidity or move assets on the main network, transferred amounts are visible, and the sending and receiving Monad Private Settlement Domains are in the clear. This can be addressed by known privacy techniques such as shielded pools and confidential transactions, which will be available in the next release of Monad Private Settlement Domains. This, together with suitable address white- and blacklisting by PSDs, provide a powerful basis for a private and compliant infrastructure.

As transactions for a PSD are submitted and sequenced on the main domain, all fees are paid on this domain. Every PSD has an account in the main domain to which received fees (minus the main domain fees) are credited;

authorization for this account is specified in the initial configuration. Transactions generated by a PSD will typically be submitted, and paid for, by one of its operators; a PSD can implement a reimbursement scheme for these payments. Since each Monad Private Settlement domain has an account on the main network, it can also own smart contracts in the main domain and own ERC-20 tokens. Furthermore, a PSD can specify which tokens are available inside it, in which case they are bridged in. The design supports atomic transfers of a token between a PSD and other domains (i.e., the main domain and other Private Settlement domains); for more complex transactions, atomicity can be achieved with coordination and locking contracts, as we show later by example.

Although our design aims to enable private EVM networks and our implementation consists of smart contracts and PSD client software, it also allows for different realizations, including non-EVM PSDs. Different realizations offer different security guarantees. If a PSD is realized by a committee of operators who multi-sign the state and message roots, security and liveness rest on a threshold assumption (since sequencing happens on the main network, one could even require *n*-out-of-*n* valid signatures, at the cost of no liveness resilience). If a PSD can afford lower throughput, its operators can run a zkVM or run the client inside a TEE for additional security. Alternatively, a PSD could also be an MPC network computing a specific functionality, e.g., a private payment or an auction, or oracles that communicate with arbitrary Internet servers, thereby allowing main domain smart contracts to interact with the outside world.

We next discuss the guarantees a PSD obtains by being embedded in the Monad network and our design. First is sender identity: contracts receiving messages from other contracts on another PSD or on the main domain will receive the verified identity of the chain and smart contract sending the message. Of course, the strength of this guarantee depends on how a PSD is realized, as it relies on the trustworthiness of the sending PSD. Second is robust message routing: contracts running on a PSD can rely on their messages being sent properly. The operators of the PSD and the hub contract will appropriately route these messages.

Monad Private Settlement domains benefit from the existing and forthcoming properties of the Monad blockchain, many of which are essential for addressing the requirements of institutions and enterprises. The former include fast and deterministic finality (which is essential for robust settlement), high throughput at low fees, composability with public liquidity, and EVM-compatibility. Three important forthcoming features will greatly strengthen PSDs. The first is an encrypted mempool [1] that will make DeFi on Monad fairer and more efficient. The second is multiple concurrent proposers (MCP) [5, 9], which will allow PSDs to submit transactions to a proposer in their jurisdiction, addressing the need to interact with known and vetted counterparties. The third is scheduled transactions [7], which make it possible to perform recurring tasks directly onchain, for instance accruing interest or checking whether positions still have sufficient collateral.

02 Related Work

There are three relevant kinds of blockchains that aim to provide privacy. The first ones are blockchains such as rollups or subnets that have their own sequencing, transaction execution, and state, but rely on a public blockchain to settle their state. Their security ultimately depends on the public blockchain. The second one is subnets. Subnets are essentially independent blockchains but rely on a public blockchain for registration and exchanging information. Finally, there are private blockchains that are fully independent and do not interact with any public blockchain.

Rollups. Rollups are blockchains that typically run a centralized operator who is responsible for ordering transactions and executing them locally [4, 11, 12, 13]. These rollups rely on a public blockchain's consensus protocol (their L1) to

periodically checkpoint their locally sequenced blocks and the corresponding execution state. Contrary to this, in Monad Private Settlement Domains we sequence every transaction on the Monad network itself. Some rollups such as Arbitrum One and OP Mainnet, are *optimistic*. Here, the operator commits to the execution state without any proof of correctness and one relies on a fraud-proof mechanism to detect the operator's misbehavior. The fraud proofs can be submitted by any party who re-executes all transactions of the rollup and, if the committed state deviates from what the operator had submitted, create and submit a fraud proof invalidating the state. The drawback with this approach is that settlement is final only after a long period of time and, crucially, it lacks privacy as it forces optimistic rollups to make their state and transactions public.

Privacy-centric rollups such as Prividium, Aztec, and Ten address the latter problem with zero-knowledge proofs of correct execution. For instance, in Prividium, only the rollup operator is privy to the state and uses STARK to prove the correctness of execution. Aztec is a decentralized rollup where users execute and prove private functions on their own devices and publish only commitments and encrypted notes. The operators of Aztec verify these proofs, execute functions on public data and submit the proofs and state updates to Ethereum [4].

Recently, Tempo has introduced *Tempo Zones*, which are private rollups that rely on the Tempo blockchain for settlements [14]. Zones are the closest design to ours, but have the following two critical differences: First, each zone runs its own sequencer, and zone transactions are never posted onchain (not even in encrypted form); second, zones only support a very limited pre-deployed set of ERC-20-like contracts.

Rollups with private state are the closest design to Monad Private Settlement Domains, the difference being that sequencing and message routing happens on Monad rather than independently. This is what makes it possible to seamlessly communicate between different domains (something that is very hard for public rollups and virtually impossible for private rollups). Similar to private rollups and Tempo Zones, Private Settlement Domains adopt a privacy-first approach, in which the state is visible only to the operators running the Monad Private Settlement domain. Regarding correctness proofs, Private Settlement domains do not prescribe a specific method for ensuring correctness of execution but allow for various approaches which are described in Section 6.

Interoperable subnets. Another way to isolate an application's state is to give it its own private blockchain within a multi-chain ecosystem. Often these blockchains are called subnets and how these subnets interact with a public blockchain varies. Some popular subnet instances are Avalanche L1s, Cosmos zones, or ICP subnets [3, 6, 10]. In these systems, each blockchain in a subnet runs its own consensus protocol and maintains its own state. Additionally, these blockchains have protocol-level support for interoperating with each other. For example, blockchains (zones) in the Cosmos ecosystem use an inter-blockchain communication protocol. Not all of them have privacy by default. However, there are a number of Cosmos chains that prevent public access to the state and do provide privacy, including Secret Network, Penumbra, and Namada. In ICP, all subnets are private by default as execution happens inside TEEs and the state is generally not accessible externally. Avalanche L1s are public by default, but one can add access control by closing off RPC access and using allow-lists to restrict who can send transactions and deploy contracts. Unlike these systems, Monad Private Settlement domains rely on the Monad network for ordering transactions and interoperability, keep their state private, and provide mechanisms to control access to state for authorized parties and provide access control policies between contracts deployed within the same domain, main domain, and other domains.

Institutional ledgers. Institutional ledgers such as Canton and Hyperledger Fabric achieve privacy by running on permissioned ledgers and partitioning the state in a way that a stakeholder only gets access to their own private state [2, 8]. For example, Canton splits a transaction into sub-transactions and encrypts each sub-transaction for the intended recipients. This ensures that different participants in the same atomic transaction see different parts of it. Hyperledger Fabric sends the private payload only to authorized organizations and writes its hash to the ledger of

every peer on the channel. Participants utilizing Canton and Hyperledger Fabric may achieve privacy, but at the cost of being cut off from the liquidity of the L1. Unlike these, Monad Private Settlement domain integrates with the Monad network enabling access to its liquidity while restricting state access to only operators, contracts, and users as specified by access control policies.

03 Use Case Examples

By combining confidentiality of Private Settlement domains with the other features of Monad such as fast execution speed and deterministic settlement finality, legacy financial institutions gain the requisite primitives to replicate existing operational models onchain while adhering to local regulatory frameworks. Although the platform supports a range of business architectures, we highlight some application scenarios below that can be addressed with Private Settlement domains. Once we have presented how PSDs work, we describe how solutions for two more specific scenarios can be built.

Cross-Border Wholesale Central Bank Digital Currency

Representing a tokenized variant of conventional central bank reserves, wholesale CBDCs enable high-value financial transfers among institutional participants. While initiatives led by the Bank for International Settlements (BIS), such as Mariana, Agorá, mBridge, Jura, and Dunbar, have explored these mechanics, nearly all relied on permissioned ledgers. This introduced fragmented liquidity pools alongside operational overheads associated with running dedicated infrastructure. Conversely, experiments on public chains like Mariana exposed transaction data completely. Monad Private Settlement domains eliminate the previous compromises by allowing central monetary authorities to create dedicated Private Settlement domains, as shown in Figure 2. Commercial banking entities can deploy contracts on the domain of their respective central banks to manage and transfer their tokenized liabilities with confidentiality and jurisdictional authority. Cross-border settlement is achieved via cross-domain messaging interacting with mainnet foreign-exchange liquidity, guaranteeing atomic, deterministic execution across both currency legs without relying on correspondent networks or pre-funded accounts. Transaction metadata and account balances remain strictly confidential within participating domains, regulatory entities preserve required oversight, and settlement anchors directly to the Monad network.

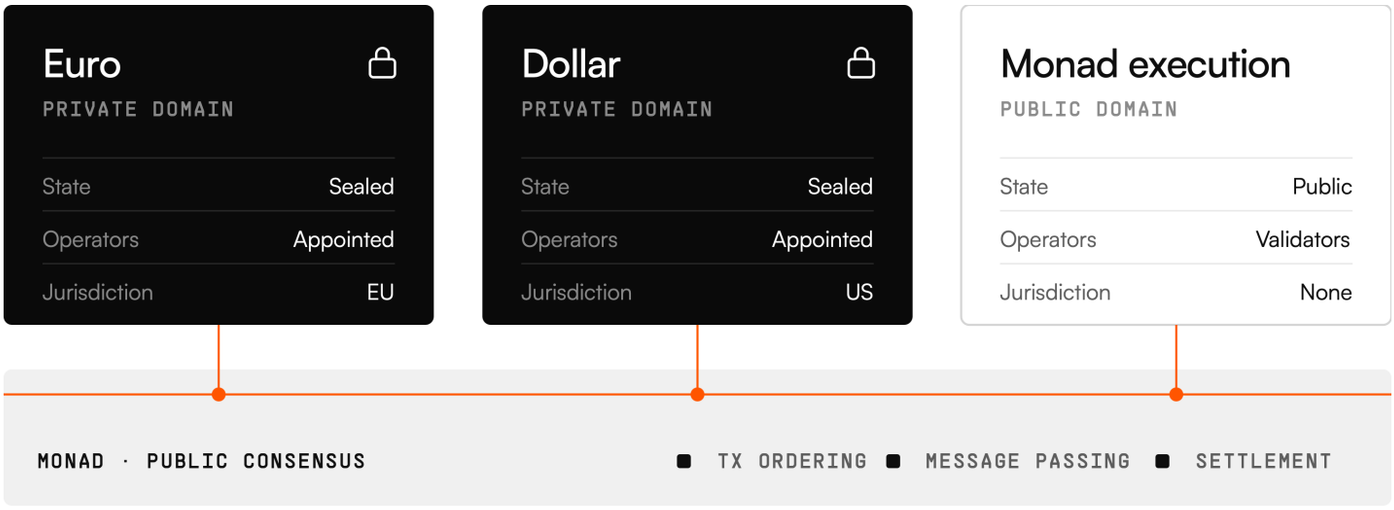


FIG. 2 Central banks can issue wholesale CBDC on Monad and have Private Settlement domains operated by parties in their jurisdiction. Commercial banks can deploy contracts in these domains to manage and transfer tokenized liability. Cross-border transactions are facilitated by cross-domain messages and by the liquidity available on the Monad.

Commercial Banking

Monad Private Settlement provides the configurable infrastructure for domain operators to support processing tokenized assets and commercial transactions in a private, regulatory-compliant environment. Customer ledger data stays hidden from the public, while supervisors and auditors maintain explicit supervisory access required for legal compliance. State transitions and value transfers settle deterministically and rapidly both within a single domain and across domains.

Enterprise Treasury Management

Payroll, vendor settlements, and intra-group liquidity adjustments among corporate entities execute deterministically and confidentially within a dedicated Monad Private Settlement domain. This prevents disclosure of transaction amounts, participant identities, or overall treasury reserves to market rivals. Granular access control parameters enabled by operators can provide regulatory bodies and tax inspectors with tailored oversight permissions, whereas governance constraints including dual-control sign-offs are directly mandated by the underlying smart contracts.

Institutional Asset Management

Fund shares and cap tables live in an asset manager's Monad Private Settlement Domain. Subscriptions, redemptions, and distributions settle atomically against stablecoins or tokenized deposits, without revealing investor identities, positions, or fund flows to the market. Investors see only their own holdings, while administrators, auditors, and regulators get the access they need, and eligibility and transfer restrictions are enforced by the fund contracts themselves.

Institutional Liquidity & OTC Markets

Proprietary desks, off-exchange market makers, and institutional clients manage request-for-quote workflows, order books, and block transactions within a dedicated Monad Private Settlement domain. Because orders are encrypted and sequenced directly on the Monad network, trading strategies, inventory allocations, quote sizes, and counterparty

identities remain fully shielded from market rivals and front-running risks. Executing the domain within TEEs ensures these details remain hidden even from a domain's operators. Transactions settle deterministically and atomically against tokenized deposits or stablecoins, while regulatory bodies and clearing entities can retain oversight access required for compliance.

04 Requirements

This section discusses typical application requirements that enterprises and institutions have when applying blockchain technology to their applications. Naturally, the exact requirements depend on the application itself; after all, addressing privacy and compliance requires a holistic design approach that considers all data and access to it by all parties. Sometimes requirements are conflicting, and a compromise needs to be found. Therefore, we summarize here the most important and most common requirements.

4.1 Application requirements

Sovereignty & control. Parties to a transaction need to implement compliance controls to comply with certain intra- and extra-jurisdictional regulatory requirements. Domestic-only transactions need to be performed under compliance with the domestic controls and cross-jurisdiction transactions need to be in compliance with the respective jurisdictions.

Counterparty risk. The parties interacted with must be identified and able to be held responsible. Interaction with sanctioned entities must be avoidable, and suitable parties such as operators and block proposers must be available and/or designated in the respective jurisdiction. Atomicity must be enforceable by the protocol (together with dedicated smart contracts), and recourse must be possible.

Privacy & compliance. The contents of transactions and the transaction metadata need to be private. Privacy needs to be conditional, so that parties such as auditors and law enforcement are able to retrieve the information they are allowed to see and require. The protocol and platform need to allow the implementation of data flow controls as imposed by regulation such as GDPR, SOX, and the travel rule. Similarly, access control mechanisms such as white- and blacklisting need to be implementable.

Mutability. Applications must be able to support controlled mutability, including freezes, state reverts, and administrative interventions. These capabilities are ultimately enabled through suitable smart contract implementations, but must be supported by the underlying protocol and data availability layer, and by suitable access control mechanisms.

Fast cross-boundary settlement. Execution of transactions across organizational boundaries and jurisdictional borders requires a fast settlement layer that anyone can easily join and transact on. The platform must ensure liveness, the highest resilience, censorship resistance, and be continuously available with sufficient capacity.

Deterministic finality. Settlement is final by construction, not probabilistic, not revertible.

Predictable fees and no native token dependency. Operating costs need to be forecastable and non-volatile. Fees should be payable in stablecoins or be sponsored. Ownership of a native token should not be required to interact with the chain.

Cross-boundary composability. The design must make composability across different jurisdictions, organizations, and chains easy and painless. Cross-boundary transactions must not depend on the availability of message routing parties or similar external parties. Liquidity must not be segregated, and markets must be automated and efficient.

Interoperability. The platform must be integrable with legacy systems, Internet services, and other chains.

No lock-in. Open source platform and built on common standards.

4.2 Technical requirements

Blockchain provides powerful security guarantees such as correct execution, consistency, transparency, public verifiability, authenticity and authorization of transactions, non-repudiation, intra-transaction atomicity, censorship resistance, liveness, availability, resilience, and tamper-proofness. The main drawback is the lack of privacy and data control; both are in conflict with permissionlessness, transparency, and public verifiability. Achieving all these elements together requires a different approach for many of the security properties, in particular those that were originally granted by all state and transactions being public and accessible to anyone. Some of the properties can be achieved at the protocol level, whereas others inherently depend on the application and need to be realized at the application level.

For instance, when transactions are encrypted and data is private, public verifiability can no longer be achieved by replaying the transactions. Instead, one could employ zero-knowledge proofs or attestation mechanisms. While such mechanisms are application-specific, they still require support at the protocol level and thus need to be considered in the design. In the following, we list requirements that must be implemented at the protocol level, laying the foundation for our design.

Confidentiality. Transactions to a PSD and messages between PSDs must be confidential, including sender and receiver addresses when they refer to an address on a PSD. All states of a PSD including the code of contracts are private. We note that, depending on the implementation of a PSD, designated operators might be privy to the state.

Authenticity. In normal EVM execution, smart contracts get information about the caller and the originator of the transaction causing the call. Albeit this is limited information, it can be trusted as the smart contracts are executed in the same environment that validated the transaction and orchestrates the calls. In our setting, this is no longer the case, contracts can call each other from different execution environments. The platform must therefore ensure that contracts also learn from which domain the call originates.

Access Control. State and transaction data in PSDs are confidential by default. However, designated parties will still need access to the state and transactional data and therefore access control mechanisms are required. These mechanisms include control for access by externally owned accounts as well as for access by other smart contracts. Moreover, these access control mechanisms need to be enforced for both RPC-calls and inside transaction execution.

Non-Repudiation/Accountability. As transactions are not public and can involve other PSDs, authenticity of a sender and a transaction can no longer be established by looking at the state and the blocks. The protocol must therefore provide alternative means that allow involved parties to verify who sent a transaction and that it was executed, and to prove this to authorized third parties such as auditors or regulators.

05 Design of Monad Private Settlement Domains

In this section, we describe the high-level design of the Monad PSD, as implemented for first release. We explain how PSDs communicate with each other and with the main domain. Because PSDs have separate execution environments, contracts in one PSD cannot simply make synchronous calls to contracts in other PSDs, get the result back and proceed with execution as usual as part of the same transaction. Rather, an asynchronous messaging infrastructure is required to enable cross-domain contract calls. The design hence distinguishes two kinds of transmission objects: transactions and messages. A *transaction* is how a user gets something executed on a PSD. The user signs it, encrypts it, wraps it in a second transaction that is then sent to the hub contract on the Monad network, paying the gas for its execution there. The Monad network defines the order of received transactions, and the domain's operators then execute them as part of the domain's next block. A *message* is how smart contracts on different domains communicate with each other. To send a message on the main domain, a contract calls the hub contract, and to send one on a PSD, a contract calls the spoke contract. The PSD messaging protocol will then deliver the message to their respective recipients. Users therefore only ever submit transactions; messages are sent and received only by contracts. We introduce the hub and spoke contracts that orchestrate PSDs in Section 5.1. Section 5.2 follows a transaction from submission to attested result, and Section 5.3 describes cross-domain messaging in more detail.

The defining difference between PSDs and the main domain is that execution and state are private. That means that many existing interaction flows for EVM blockchains have to be revisited to ensure that they cannot be used to access private data. We discuss our approach to access control to state and contracts in Section 5.4. Finally, in Section 5.5, we describe how tokens flow between different domains.

5.1 The Hub and Spoke Contracts enable Domains and Communication

Monad PSDs and communication between them are enabled by two protocol contracts at fixed addresses: the *hub* on the main domain and a *spoke* on each PSD. A PSD is created by sending a transaction to the hub contract. The transaction pays for the domain's creation and specifies its configuration, including operators, the method to use for attesting to its state and outbound messages, a public encryption key, optional access restrictions such as an allowlist for contract deployment, transaction sending and access to private data, the address of its token bridge contract on the main domain, and the root of the domain's genesis state. The hub stores this configuration, so anyone can look it up. The default genesis state already contains the spoke contract. The creator will also make the secret decryption key available to the operators. There are a number of ways this key can be generated and made available. Once a domain is registered with the hub and the decryption key distributed, the operators can start running the Monad PSD client (an extension of the Monad client), making the domain live.

The hub has five responsibilities:

1. **Sequencing.** All transactions for all PSDs are submitted to the hub. The hub orders them as part of the Monad network's block execution. The hub rejects sequencing requests targeting domains that are behind on processing transactions to protect users (Section 5.2.4).
2. **Configuration and fees.** The hub holds each domain's configuration, including its public key, its attestation policy, and a gas account into which the fees for the domain's transactions are paid. The domain owner uses this account to compensate its operators.

3. **Settlement and message authentication.** The hub accepts each PSD's attested execution results and, with them, a commitment to the messages the domain emitted. Messages leaving a domain are later checked for authenticity against this commitment.
4. **Cross-domain messaging.** While users communicate with PSDs by sending domain transactions to the hub, contracts on the main domain utilise a different interface on the hub to send messages to contracts living on PSDs (corresponding to making asynchronous calls). Symmetrically, the hub also processes inbound messages from other domains, routing them to their recipient contracts.
5. **Replay protection.** The hub records which messages have been delivered on the main domain, so that none is delivered twice.

The spoke contract is the hub's counterpart inside each PSD. Contracts inside the PSD call it to send messages to contracts on other domains (i.e., asynchronous calls). Unlike the hub, the spoke encrypts all outgoing messages to other PSDs. The spoke contract also processes incoming messages, i.e., it delivers them to their recipient contracts, again with replay protection. If messages are encrypted, they are decrypted by the system before they hit the spoke contract. The spoke does not verify the sender authenticity of incoming messages as the hub has already done so.

5.2 Transaction Lifecycle and State Settlement

A user transaction for a PSD passes through four stages: it is submitted and paid for on the main domain, sequenced into a domain block, executed by the domain's operators, and its results are attested back to the hub.

5.2.1 Submission and gas payment

A user creates and signs an ordinary EVM transaction for the target domain, using the domain's chain id. The wallet then encrypts the transaction under the domain's public key and wraps the ciphertext in a *courier transaction*: an ordinary Monad network transaction that calls the hub with the ciphertext as calldata. That courier transaction could also be sent by another party, e.g., a party that sponsors transactions for a domain's users. Only the target chain id and the gas limit of the domain transaction remain in the clear, with the recipient, calldata, nonce, sender, and signature being encrypted.

The courier transaction pays for both parts of the work. Its own gas covers the hub call on the main domain, and its value covers execution on the domain: the hub charges a fixed price per unit of gas times the declared gas limit, transfers that fee to the domain's gas account, and refunds any excess. Because the payer of the courier transaction need not be the signer of the inner transaction, gas sponsorship comes for free: a bank, an application, or the domain operator can pay for its users' transactions. Domain transactions carry no native value; MON can be bridged in as WMON (Section 5.5).

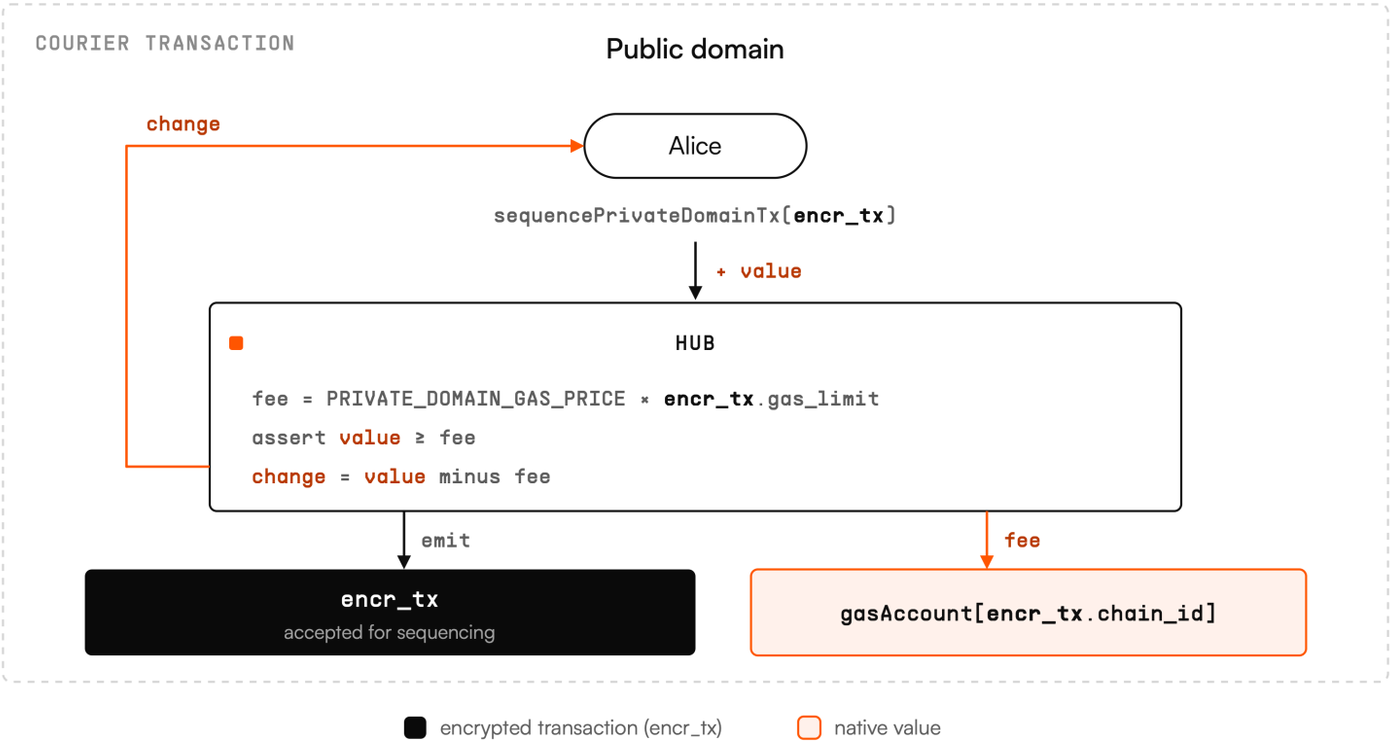


FIG. 3 Submitting an encrypted transaction inside a courier transaction: the hub prices the declared gas limit, forwards the fee to the domain's gas account, refunds the change, and records the encrypted transaction for sequencing. Decryption of the encrypted payload happens at domain block derivation.

5.2.2 Sequencing and block derivation

The operators of a domain do not run consensus. Instead, they derive the domain's blocks deterministically from the Monad network, using two rules:

1. **FIFO.** Transactions enter the domain in exactly the order in which the Monad network sequenced their courier transactions.
2. **Lock-step.** Each Monad block that contains at least one transaction for the domain induces exactly one domain block with precisely those transactions. Future versions may let a domain define its own cadence, for example one domain block per five Monad blocks.

To derive a block, each operator decrypts the accepted transactions with the domain's secret key. A transaction that fails to decrypt or is invalid is skipped rather than invalidating the block. Because domain block derivation depends only on the Monad network's blocks and decryption is deterministic, every operator of a domain will obtain byte-identical blocks for their domain. This also enables operators that were offline to catch up at any time.

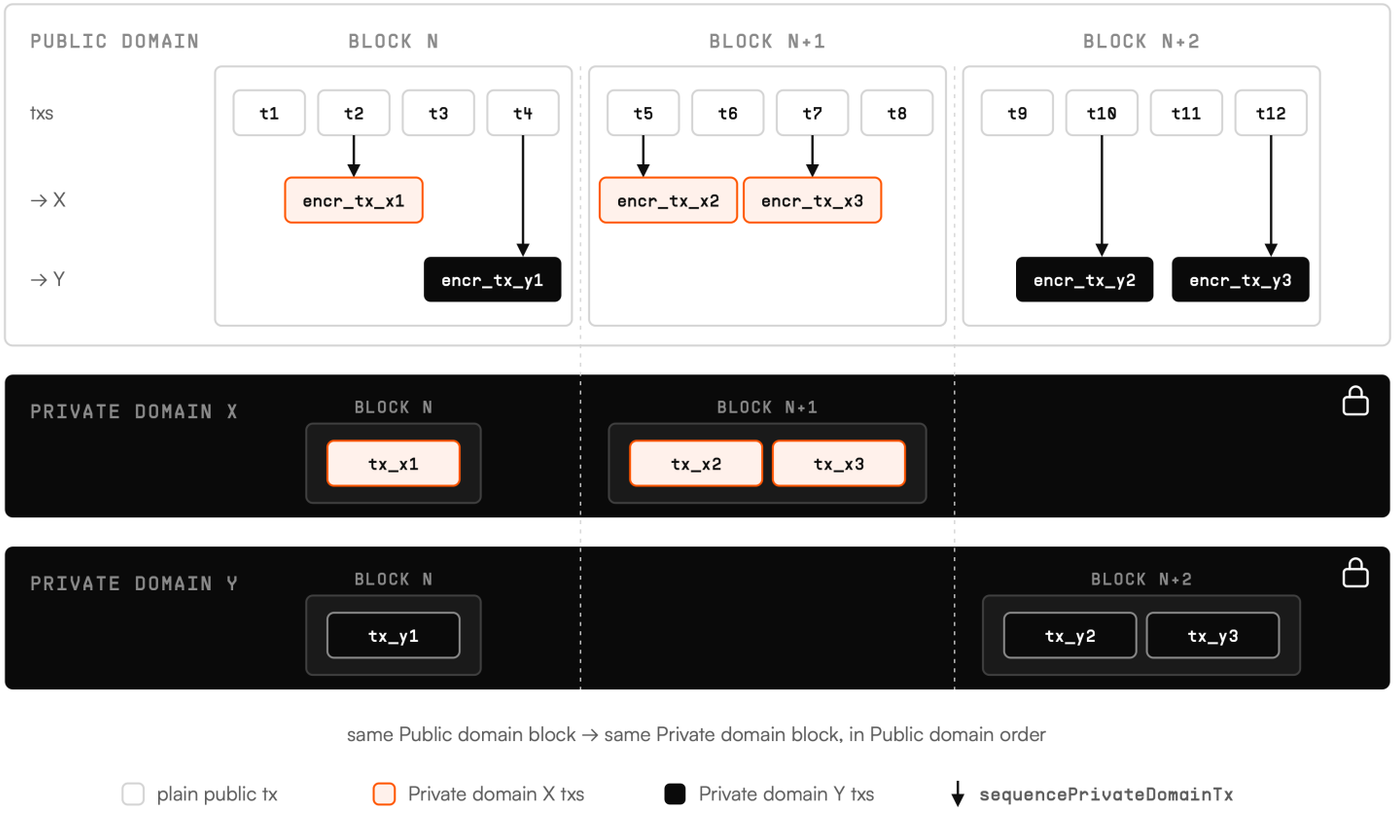


FIG. 4 Lock-step derivation: each Monad block carrying transactions for a domain induces exactly one domain block containing them in Monad order.

5.2.3 Execution and results attestation

The operators execute each block against the domain's state. The result is a new state and the list of messages the executed contracts want to send. The operators salt and Merklize both, and report to the hub a results record containing the block number, the new state root, and the root of the messages tree, called the *message anchor*. The salting keeps these public commitments from revealing anything about the state or the messages.

Each result must carry an attestation that matches the policy the domain registered at creation. This can be either signatures from a threshold of the domain's operators, a TEE attestation that the client executed inside the TEE, or a zero-knowledge proof that applying the derivation and execution rules to the previous state yields the reported roots. The hub accepts results strictly in block order, updates the domain's state root, and records the message anchor. In section 6, we elaborate on the different attestation mechanisms and their trust assumptions.

5.2.4 Block capacity and backpressure

Two limits protect domains and their users. First, the hub caps the total gas it accepts for each domain per Monad block (`PSD_BLOCK_GAS_LIMIT`), so that no domain block can be larger than its operators can execute. Any request that would exceed the cap fails and debits the caller nothing. Second, the hub stops accepting transactions for a domain whose execution has fallen too far behind, i.e., if the number of sequenced but not yet attested domain blocks reaches `MAX_EXECUTION_GAP`. Users are thus never charged for transactions sequenced into a long backlog with no prospect of execution, and a domain that stops reporting results stops accepting new work. Users can query the hub to detect if any domain is backlogged and refrain from sending transactions in such situations.

5.3 Cross-Domain Messaging

Cross-domain messaging lets a contract on one domain call a contract on another domain. Because the two contracts run in different execution environments, such a call cannot use the EVM's normal call mechanism: the target must be addressed by both its domain and its address and the call is necessarily asynchronous. A contract sends a message by calling the `sendMsg` interface on the hub (if the sending contract is on the main domain) or on the spoke (if it is on a Private Settlement Domain), naming the destination domain, the receiving contract, and the payload. The call returns immediately, with any reply having to come back as a separate message. To receive messages, a contract needs to implement the `processMsg` interface and must check that its caller is the hub or the spoke. The caller's address and domain are attached to the message by the protocol and therefore cannot be forged.

5.3.1 Message flow directions

Every message starts with a sending contract invoking `sendMsg` and ends with a call to `processMsg` on the receiving contract. The interface `sendMsg` on the hub and spoke contracts takes as input the receiver contract's address, its domain, and the payload. They will then form the full message by adding the sender's contract address and domain (this ensures that a sender cannot lie about its address or domain). The interface `processMsg` takes as arguments the sender contract's address, its domain, and the payload.

What happens in between calling `sendMsg` and `processMsg` depends on the caller and receiver's domains. After carrying out required authentication and replay-protection checks, messages are delivered to the recipient contract by the hub (on the main domain) or by the spoke (on PSDs). The recipient contract remains responsible to check that the caller of `processMsg` is the hub or spoke. To understand the details, as well as how cross-domain message transport occurs over the network, let's go through the three cases separately:

1. **main → private.** A contract on the main domain calls `sendMsg` on the hub. The caller pays the message's delivery with gas as for a transaction, and the destination domain's operators automatically wrap the message into a sequenced system transaction (with sender `DOMAIN_SYSTEM_RELAYER`) delivering the message to the spoke, which then forwards it to the recipient contract. The authenticity of the message sender is guaranteed by the EVM supplying the hub contract with the caller identity in `sendMsg`.
2. **private → main.** A contract on a PSD calls its spoke's `sendMsg` interface. The message is included in the message anchor of the next attested block. The operators carry their domain's outbound messages to the hub by wrapping them in transactions that they pay and send to the Monad network targeted to the hub contract. Each message is accompanied by an inclusion proof that the hub checks when receiving the message and before calling the recipient contract on the main domain to deliver the message. The authenticity of the message sender here is guaranteed by the inclusion proof that the hub verifies with respect to the message root that the operators submit together with the state root (Section 5.3.2).
3. **private → private.** This combines the two: the message is authenticated on the main domain as in the second case, and then delivered to the destination domain as in the first. The main domain relays the message without being able to read it as the spoke will encrypt it under the recipient's domain public key.

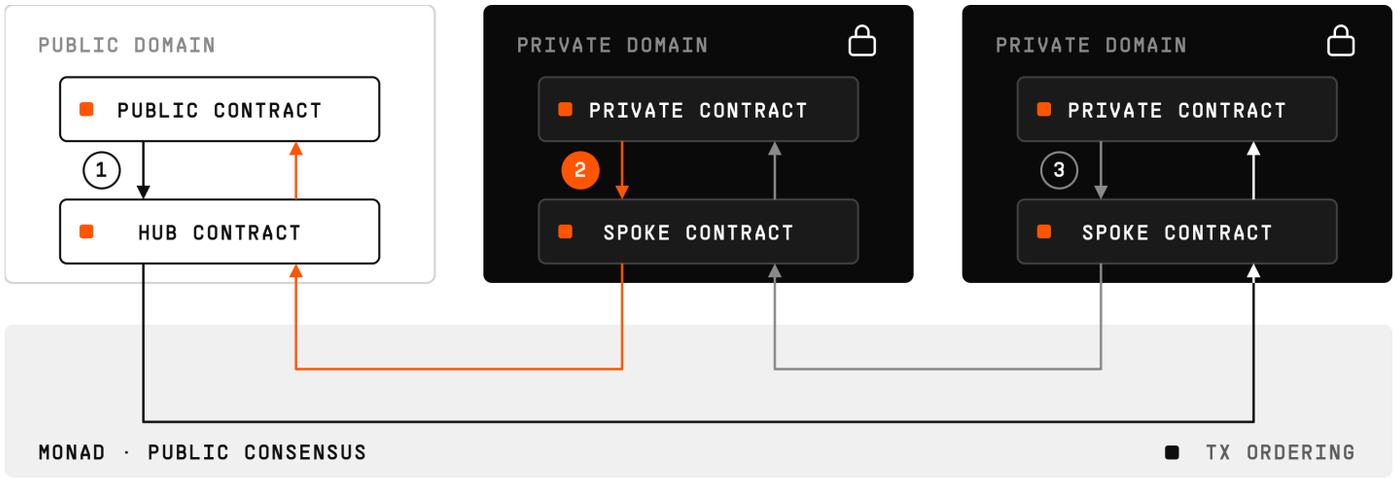


FIG. 5 Message flows to and from a Monad Private Settlement Domain.

5.3.2 Authentication with message anchors on the main domain

The main domain only accepts messages that a PSD actually emitted. When the hub accepts a domain block's results, it stores the block's message anchor, the root of a Merkle tree over the hashes of all outbound messages the block emitted. To deliver a message to the receiving contract on the main domain or forward it to the receiving domain, the domain operators of the sending domain present the message together with a Merkle path from the message's hash to an anchor stored in the hub; the hub then checks the path to authenticate the message. For private → main messages, it then delivers it to the receiving contract, while for private → private messages it re-emits it towards the destination domain, akin to a main → private send operation. Since anchors enter the hub only as part of attested execution results, a valid proof exists for exactly those messages that resulted from a domain's execution, and for every one of them.

5.3.3 Replay protection and retries

On top of ensuring message authenticity, the hub and spoke enforce replay protection (i.e., a message can only be successfully delivered once). Message delivery can also fail, typically due to an improperly set gas limit on the delivery transaction. The hub and spokes record the hashes of successfully delivered messages to avoid replays, and allow retrying unsuccessful deliveries with a new transaction with fresh gas (for messages with a PSD as destination, the retry transaction is no longer a system transaction). The hub additionally records hashes of the private → private messages it successfully forwarded to prevent double-forwards (that would erroneously create two associated system transactions).

5.3.4 Message encryption and key registry

During creation of every Private Settlement Domain, an encryption public key is specified under which messages (and transactions) for it are encrypted. That of course requires that key to be known authentically by the spoke contract on any domain sending messages to it. This is achieved by an interface on the hub contract to request that the public key of PSD A be sent to PSD B, causing the hub contract to send a message to PSD B's spoke contract containing the public key of PSD A. This ensures that only the public key registered for Domain A will be used by the spoke contract. The operators of Domain A will have received the corresponding secret key during the setup of the domain to be able to decrypt messages (and transactions) for their domain. In the first version of Private Settlement Domains, a domain's public key cannot be rotated; this will be supported in the future, however.

Messages between Private Settlement Domains are encrypted entirely, including caller and recipient contract addresses, under the destination domain's public key; the spoke does this automatically, provided that the encryption public key has been sent to it. If not, the spoke will reject the message and log the failure so that the operators can send a transaction to the hub to provide the missing public key to their domain's spoke. Messages to the main domain are in plaintext, because their recipient runs on a public chain. Each message to the main domain also carries a salt, so that its commitment in the message root will be randomized and not reveal anything about its content, as the message root will become public before the message (for encrypted messages this is not necessary of course).

Encryption of messages originating from a private domain is carried out by the spoke contract. Consequently, encryption has to be deterministic, meaning all randomness going into the encryption algorithm should be generated using a pseudo random function (PRF) using a secret seed shared by the domain's operators. We note that, unlike on a public chain, onchain encryption here makes sense as the state and execution are private.

5.4 Access Control to State and Contract APIs

To achieve privacy, Monad PSDs limit access to their private state only to authorized users. One approach to implement such access control would be to filter state access requests at the RPC boundary. This is needed, but not the whole story: it is insufficient as an adversary may try to indirectly access the sensitive state by deploying a smart contract on the Monad Private Settlement Domain and then reading the sensitive state through its deployed smart contract.

Sensitive data in the EVM is scoped at the account level (recall that contracts are also accounts): each account has a native balance, a nonce, and, if the account is a smart contract, associated storage and code. Smart contracts also emit logs, which would expose contract internals if left unprotected. Our design enforces access control at two entry points: RPC calls and execution calls. We also distinguish global reader access, given by the domain owner to select individuals such as technical staff of the owner's organisation or external auditors required for regulatory and jurisdictional compliance. The identities of these readers are configured by the domain owner on the hub, and is hence publicly accessible to everyone.

At the RPC layer, when an offchain querier asks to read data, we authenticate the reader, and enforce that it does not read data belonging to an account they do not own. RPCs provide several endpoints returning potentially sensitive data, ranging from direct balance, nonce and storage reads all the way to advanced simulation endpoints. In the former cases, we enforce that the reader can prove ownership of the account (e.g., by a signature), while in the latter we selectively disable functionality that can leak data and enforce additional checks. In particular, for `eth_call`, we do not allow simulating a call on behalf of an account the reader does not own (a normal EVM allows this). Likewise, transactions and their receipts (`eth_getTransactionByHash`, `eth_getTransactionReceipt`) are returned only to the transaction's sender. Even for the sender, we prune from the receipt the logs emitted by contracts that executed as part of the transaction, as these could reveal confidential information about the contracts' internals. Logs remain retrievable through the log-reading endpoints (`eth_getLogs` and the log filter endpoints), which return only the logs the querier is authorized to read. The rules are explained below, but, at a high level, a log can be read only by (i) readers authorized by the contract emitting it, and (ii) readers whose address matches one of the log's indexing keys.

At the Execution layer, we guard against contracts reading data they are not authorized to read (as we have seen above, an attacker would otherwise deploy a contract with the sole purpose of performing unauthorized reads). We do this by allowing contract owners to set access control policies on their smart contracts. Data belonging to smart contracts that needs protection comprises contract storage and code. Native balances do not need protection, as

they are zeroed-out on Private Settlement Domains: all value lives in ERC-20 tokens, including WMON. Balances of ERC-20 tokens are represented in contract storage, so they stay covered.

The EVM already limits the ways one contract may read another contract's storage. In particular, the only way a contract A can get access to another contract B's storage is through A calling B's methods using one of the opcodes `CALL`, `STATICCALL`, `DELEGATECALL` (or the out-of-use `CALLCODE`). In our design, contract deployers specify a policy enforcer contract outlining the access policy on the contract's data. Whenever a call is made to a contract with a set policy enforcer, the execution client automatically injects a call to the policy enforcer contract (contracts with no enforcer are set to always allow). The policy enforcer receives information about the call being made, including the calldata, identity of the caller and other contextual information about the transaction like the transaction sender, and returns a boolean: allow or deny. A policy denial reverts the whole transaction. This follows a minimum-information-leakage principle, deterring attackers from using conditional control flow to learn confidential information about the policies themselves (e.g., calling a contract, seeing the policy deny, then trying a different call, all within one transaction). Reading another contract's code happens through the `EXTCODE*` family of opcodes (`EXTCODESIZE`, `EXTCODECOPY` and `EXTCODEHASH`). Those get similar treatment.

We note that a contract and its policy enforcer contract can coincide, as long as the contract implements the required policy-checking interface. This enables flexible patterns where the policy takes into account contract state. Legacy contracts that do not want to opt for this two-in-one design do not need to change, and can instead use a dedicated policy enforcer contract.

In our design, contract access control policies are written in Solidity, thus inheriting Solidity's full expressibility. However, we recommend keeping policy checks simple: complicated policies incur more computational overhead in the injected calls to the policy enforcer contract: these calls are metered as part of the usual gas mechanism of the EVM, and hence can raise transaction costs.

We now return to access control to a contract's logs (events in Solidity). As discussed above, those are unconditionally pruned from transaction receipts and can instead be retrieved through the RPCs log reading and filtering endpoints. First, a contract owner configures a set of allowed readers for the contract's logs. RPC nodes will then allow any of these readers to retrieve all the contract's logs. Recall that domain RPCs authenticate read requests, so other parties cannot impersonate authorized log readers without appropriate authorization. This functionality is intended for the contract deployer and any other parties they trust, e.g., DevOps at the deployer's organisation. Second, we allow any party whose address matches one of the log's index keys to retrieve the logs (for context, EVM logs consist of an emitter address, up to four index keys, and everything else as data). This is intended to allow users of a contract to retrieve logs pertaining to them. For instance, Alice should be able to read all ERC20 transfers to and from her address.

A more general design here would be to let the contract deployer specify an explicit access control policy for the contract's logs (in Solidity), just like for opcodes in the `CALL` family. This is, however, undesirable, as it would require running the EVM on each log entry, which could be prohibitive from a performance standpoint. Moreover, any gas metered in these runs would not be paid for as requests to RPCs are not replicated.

Example: an ERC-20 token. We now discuss one potential set of access control policies for the widely used ERC-20 token contracts. An ERC-20 contract maintains tables of balances and spending allowances. It exposes metadata functions such as `name` and `totalSupply`, the read functions `balanceOf` and `allowance`, and the mutating functions `transfer`, `approve` and `transferFrom`. Moreover, it emits events (i.e., logs) on successful transfers and approvals. Most of these do not require any additional access control policy: the metadata functions are public, and the mutating functions already have EVM-layer checks to filter unauthorized access. Therefore, we only require policies for the read-only functions and events.

We summarize our suggested policies for the read-only functions in the table below. Here, we use `caller` to refer to the latest entry on the EVM call stack, and `readers[a]` for the list of accounts that are allowed to read `a`'s balance and allowances.

Function (in the ERC-20 contract)	Policy (in the associated policy-enforcer contract)
`balanceOf[account]`	`caller == account or` `caller ∈ readers[account]`
`allowance[owner, spender]`	`caller ∈ {owner, spender} or` `caller ∈ readers[owner] ∪ readers[spender]`

The list `readers[a]` is maintained in the storage of the policy enforcer contract, which provides an additional method allowing `a` to issue or revoke reading permissions to its balance and allowances.

When it comes to events (Solidity's abstraction over logs), the ERC-20 token emits two types of events, on successful transfers and successfully issued allowances respectively:

```
event Transfer[address indexed from, address indexed to,   uint256 value]
event Approval[address indexed owner, address indexed spender, uint256 value]
```

Solidity compiles those down into logs with three index keys: the first is a hash of the event signature string (event name and type information, no variable names), and the other two are `from` and `to` (for Transfer), and `owner` and `spender` (for `Approval`). The access control policy for logs then allows a user that authenticates to the RPC as Alice to read all logs corresponding to events of the forms:

```
Transfer[Alice, -, -]
Transfer[-, Alice, -]
Approval[Alice, -, -]
Approval[-, Alice, -]
```

i.e., all logs where one of the second or third index keys matches Alice. The astute reader will wonder whether, in principle, a user Mallory could read a log by matching the first index key, e.g., for Transfer events this would be h = `keccak256["Transfer[address, address, uint256]"]`. This is not possible unless Mallory can authenticate as address `h`, which would require Mallory to know the secret signing key for address `h`.

Many ERC-20 tokens implement additional functionality, like permits (with the associated nonces), burning and minting functionality, or functions only accessible to the contract owner. For such tokens, the access control policies can be extended.

5.5 Token Bridging

Processing tokens may be the most important use case of blockchains, and enabling token transfers across domains is a core goal of our design. Which tokens can be bridged into a Monad Private Settlement Domain is specified in its configuration. Bridging uses a lock-and-mint design in which tokens remain native to the main domain and PSDs hold a synthetic representation of them. The token bridge consists of two contracts: a main-domain bridge that escrows collateral on the main domain, and a domain bridge that mints and burns the bridged representation on the PSD. Each domain is served by its own main-domain bridge instance, so collateral is isolated per domain, and additional

functionality such as OFAC lists checks and KYC flows can be realized in these contracts. The trust assumptions of the bridge are those of the domain's results attestation (Section 5.2.3). Both bridge contracts communicate through the messaging layer of Section 5.3. To make native MON available in PSDs, WMON (wrapped MON) is used. In the following, we describe how token flows across domains work.

5.5.1 Token Deposits (main domain to a Private Settlement Domain)

To send a token from the main domain to an account on a PSD:

1. The user approves the main-domain bridge contract and requests a transfer of a token to a recipient address on a private domain.
2. The main-domain bridge pulls the tokens into escrow and verifies that the bridge's balance increased by exactly the requested amount.
3. The main-domain bridge sends a main→private message containing the recipient, the token, and the amount to the token bridge contract on the private domain.
4. The message is delivered to the domain token bridge contract as a system transaction (cf. Section 5.3.1).
5. The domain bridge validates the message's origin, deploys a bridged token contract (at a deterministic address) if needed, and mints the tokens to the recipient address.
6. The deposit is considered completed after Step 3. Should the minting fail, the delivery is recorded as failed and can be retried permissionlessly without blocking subsequent messages (cf. Section 5.3.3).

5.5.2 Token Withdrawals (Private Settlement Domain to main domain)

To transfer a token from a private domain to the main domain:

1. The user makes a request to the private domain's token bridge contract. The domain bridge burns the tokens first, then sends a private→main withdrawal message to the counterpart token bridge on the main domain.
2. The message is included in the message anchor of the next domain block attestation (Section 5.3.2).
3. Once the anchor is recorded in the hub contract on the main domain, domain operators carry the message together with its inclusion proof to the main domain. The user does not need to send a second transaction.
4. The hub receives the message and verifies that the anchor was recorded by the hub as part of the domain's attested results, the inclusion proof, and checks for replays. The hub then passes the message to the main-domain bridge.
5. The main-domain bridge receives the message, verifies the sender is the domain bridge and then releases the escrowed tokens to the address requested. Successfully processing the message by the main-domain bridge finalizes the withdrawal.

For the first version of Monad Private Settlement Domains, a token transfer from one Monad Private Settlement Domain to another is a withdrawal from the source domain to the main domain followed by a deposit into the destination domain.

5.5.3 White and Blacklists

A PSD can whitelist or blacklist tokens for deposits in its configuration. When a token is blacklisted, users cannot deposit it into the corresponding domain. The token bridge is also the place to implement OFAC address checking and to implement KYC'ed gating on addresses. Withdrawals are not affected, so users are always able to exit from a PSD. The design is flexible: different PSDs could have different rules.

06 Deployment and different realizations

So far, we have described how the main domain and Monad Private Settlement Domains (PSDs) communicate and how PSDs commit their state roots to the main domain. In this section, we discuss possible ways to instantiate PSDs and compare them. Note that we do not require PSDs to be EVM engines. They should be viewed as generic asynchronous communication protocols that enable main-domain contracts to communicate with the external world. As different realizations have different requirements on how transactions and messages to PSDs are encrypted, we allow PSDs to specify the encryption scheme they support.

6.1 Threshold Committee

In this approach, a committee of n operators runs a PSD, where each committee member executes the PSD transactions, jointly signs the result, and reports it back to the hub contract. For the joint signature, the committee can use a cryptographic threshold or multisignature scheme. Note that since sequencing happens on the main domain and execution is deterministic, the PSD committee is not required to run a consensus protocol. A key point to consider when choosing the operators and the signing threshold is the trade-off between PSD security and liveness. For instance, if a PSD requires, say, $n - k$ operators to sign, execution correctness can tolerate up to $n - k$ colluding operators. In contrast, liveness can tolerate up to k offline or malicious operators. Furthermore, since all operators see the full state, a single rogue operator is sufficient to break the privacy of the PSDs.

6.2 Operators with zkVMs

In this approach, the operators of a PSD will provide a cryptographic proof of the correctness of the PSD state transition using (Zero-Knowledge) succinct non-interactive arguments of Knowledge (zkSNARKs), informally known as zkVMs. At a high level, in this approach, each PSD operator would run a zkVM to prove the following:

1. Deriving the block from what is provided by the hub, correctly decrypting when necessary, yields a set of transactions;
2. Executing these transactions against the old state yields the new state; and
3. The message hashes in the message tree are correct as per the outgoing messages emitted by the spoke contract.

The advantages of this approach compared to the committee-based approach are that the PSD can be run with far fewer operators. This is because the zkVM approach relies on cryptography for correctness rather than on the collective opinion of operators. Fewer operators also reduce privacy leakage due to rogue operators. However,

existing zkVMs introduce significant overheads, which would require operators to have higher compute resources. Moreover, the throughput of a PSD will be limited by the throughput of the underlying zkVM.

6.3 Operators with TEEs

In this approach, PSD operators will run their code within trusted execution environments (TEEs), such as Intel TDX, AMD SEV-SNP, or AWS Nitro. Here, the decryption key of the PSD will exist only inside the operators' TEEs and the correctness of state roots is proved by using TEE remote attestation. Compared to the approaches above, the PSD state in this approach will be inside the TEE and will not be accessible to the operators. We note, however, that the correctness guarantee requires trusting the manufacturer and the assumption that an attacker cannot extract the signing key from a TEE. Similarly, for privacy, on the assumption that an attacker is unable to extract the PSD decryption key. It is also possible to combine the TEE and zkVM approaches, using both TEE attestation and zkVM to ensure correctness, and relying on the security of the TEEs for privacy only. This means that an attacker would need to break both the TEE and the zkVM cryptography to violate correctness.

6.4 Non-EVM Private Settlement Domains

The first version of PSD is an extension of the Monad client and therefore an EVM environment. However, the design does not require a PSD to be an EVM environment; it only prescribes how the main domain and PSD interact. In the following, we briefly discuss alternatives and their benefits.

6.4.1 Cryptographic compute services

Monad PSDs can also be instantiated as interfaces to a cryptographic compute service, i.e., a network of machines computing on encrypted data without revealing the plaintexts to any single machine. Contracts on the main domain or on other Monad PSDs can submit requests to this service as messages. These requests may also refer to data stored by a data availability (DA) layer. Such a service can be realized with multiparty computation (MPC), fully homomorphic encryption (FHE), or a network of TEEs. We will briefly describe each of these and discuss their tradeoffs.

In the MPC realization, the operators will compute functions over secret-shared inputs without disclosing the plaintext inputs to any single operator. The privacy and security guarantees of such a PSD would rely on those of the underlying MPC protocol. These MPC-based realizations are suitable for application-specific PSDs, including private tokens (cf. Merces by Taceo), private auctions, dark-pool matching, credit scoring, or joint analytics over data contributed by mutually distrusting institutions.

In an FHE realization, operators will evaluate contracts directly on ciphertexts (e.g., Zama's fhEVM). The FHE decryption key will be held by a threshold committee that decrypts only designated outputs. Thus, privacy assumes that this committee does not reconstruct the key. Moreover, for correctness of execution, we can again either use a committee-based or zkVM-based approach. Given the substantial computational overhead of FHE, this realization is well-suited to high-value, low-throughput applications.

Finally, in a TEE realization, a network of TEE workers will schedule each request onto one of the enclaves, which returns the attested result. Unlike the TEE committee approach we discussed earlier, in this realization the worker set can be open and dynamically assigned. Input and output state can be maintained in encrypted form under a key managed by the TEE network. This approach works best if requests operate on disjoint state.

6.4.2 PSDs as Oracles

PSDs can also serve as external oracles for the main domain and other PSDs. A smart contract on a main domain or other PSDs can subscribe to a PSD that monitors real-world events and be notified via a message when an event occurs. As events and their requests are encrypted under the PSD's key, they will be private to non-involved domains. Additionally, compared to bespoke oracle integrations, using PSDs as an oracle provides uniform message routing and fees. We note that to ensure the accuracy of information about external events, the oracle PSDs should adopt the fault-tolerant approaches we discussed earlier.

6.4.3 Bridges to other blockchains and manage their assets

PSDs can also serve as cross-chain bridges, allowing contracts on the main domain or other PSDs to interact with external blockchains. For example, a bridge PSD can operate a light client of a foreign chain and submit transactions to it. Contracts can send requests to the bridge PSD and receive messages reporting events observed on the foreign chain. Alternatively, a bridge PSD can communicate with RPC nodes of a foreign chain and emit their responses as events. Unlike oracles, we can implement foreign-chain-specific verification rules on the main domain to verify these responses. For example, a smart contract on the main domain can maintain and validate Bitcoin's header chain using headers relayed by a PSD, enabling independent verification of reported events. Another useful property of PSD-based bridges is that requests to the bridge are encrypted, so the intent to transact (destination, asset, and amount) remains hidden until the transaction appears on the public foreign chain. Finally, PSDs can provide threshold signing capabilities, which, in combination with the above, allows smart contracts on the main domain or other PSDs to own assets on foreign chains.

07 Example Applications Realizations

At the core, Monad Private Settlement Domains are secure, private, and programmable execution environments that can natively interact with the Monad main domain, that is a secure public programmable execution environment. The two kinds of domains can address requirements that the other cannot by itself. In combination, they can address a very wide range of use cases. In the following section, we briefly discuss two example scenarios to show how the use cases can be addressed and demonstrate the interplay between different domains.

7.1 Cross-Border Wholesale Central Bank Digital Currency

Section 3 outlines the core objectives of wCBDCs. Here, we examine how the cross-border, multi-currency transfer model explored in Project Agorá can be realized using Monad Private Settlement Domains (PSDs). The primary goal is to replace traditional institutional payment flows, which currently rely on complex networks of pair-wise correspondent banking relationships to route value across borders.

One possible solution using PSDs is as follows. Each central bank, together with the commercial banks in its jurisdiction, operates a PSD. Each central bank issues its wCBDC on Monad and makes it available in its PSD. The commercial banks can transact within the domain, where they maintain their own ledgers to manage their customers' funds and can implement and enforce any necessary KYC workflow. The latter is now possible onchain because of the privacy offered by PSDs. Since all wCBDCs are also available on the main Monad domain in this solution, global liquidity pools and FX can be implemented there. This prevents liquidity segregation and allows liquidity from other pairs to be used for less liquid pairs.

Now, consider an example flow in this scenario. An enterprise that is a customer of Bank A in a Euro domain wants to settle a bill of, say, $9,500 from a supplier that is a customer of Bank B in a Dollar domain. To do so, the enterprise sends Bank A a transaction asking it to send that amount in dollars to the supplier. Upon receiving this transaction, Bank A's smart contract first sends a cross-domain message to Bank B to inquire whether the supplier's account is valid and the payment can be executed. Upon receiving the reply from Bank B's smart contract, which resides on the Dollar Private Settlement Domain, Bank A's contract sends a message to the shared liquidity pool for the Dollar-Euro pair on the main domain. This message asks the pool to lock $9,500 for Bank B and to provide a quote for the amount of euros, say €8,500, needed to settle the bill. Upon receiving the confirmation, Bank A debits the enterprise's account by €8,500 and initiates a token transfer of €8,500 to the liquidity pool address. This causes the pool to release $9,500 to Bank B, which credits the supplier's account, settling the bill.

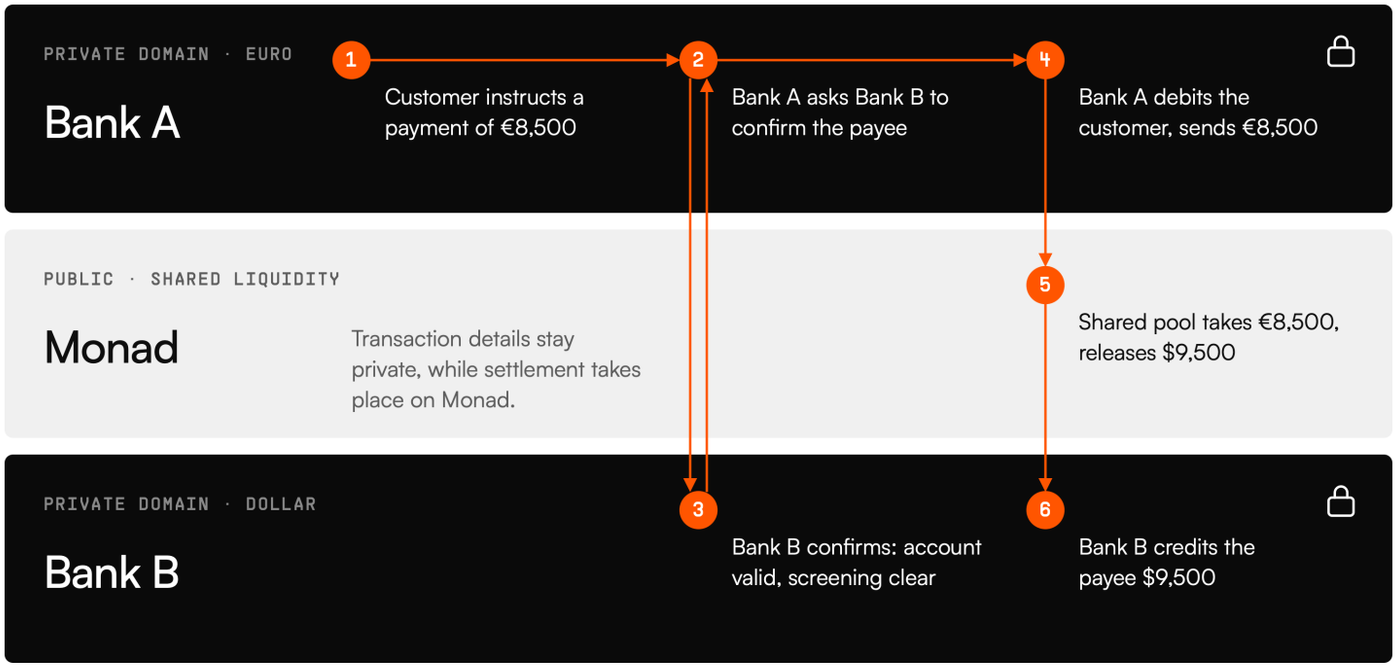


FIG. 6 Cross-domain settlement flow.

7.2 Multi-Currency Derivative Swaps and Tokenized Offerings

Our second example is a dealer bank, Bank D, serving hedge funds that want exposure to a listed equity basket with different payoff profiles and in different currencies. An issuer sells a tokenized equity-linked note (ELN), whose redemption amount is calculated from the prices of the basket's listed shares according to a formula fixed at issuance (for example, 80% of the basket's return, capped at a 30% gain and floored at a 10% loss). The ELN's register is held in a PSD operated by the issuer's transfer agent, which allowlists eligible investors after KYC, and Bank D acts as arranger. Subscription settles delivery-versus-payment in tokenized cash. Funds can then use swaps with Bank D to reshape that exposure, for instance into capped, downside-protected, or floating-rate payoffs, settled in the currency of their choice. Bank D hedges in the underlying shares through its custodian, in conventional or, where available, tokenized form.

Payoffs depend on official closing prices of the basket's shares on the dates set in the terms, adjusted for dividends and corporate actions, together with an FX fixing and, for floating legs, interest rate benchmarks. These come from a market-data service, a PSD operated by a data vendor, which returns each figure from a named source with agreed

fallbacks. Contracts query it by cross-domain message and pay per query, with queries and replies encrypted under the receiving domain's key (Section 5.3.4). The querying contract records the source, timestamp, and value, so the figure behind every payoff can be audited. Because the payoff derives from listed prices, the ELN is easy to value.

Each swap is governed by an ISDA Master Agreement and Credit Support Annex (CSA) signed offchain. A swap contract per fund in the dealer domain automates it, storing the terms and the hash of the signed confirmation. Access control (Section 5.4) is what ultimately makes this use case work: it ensures that no fund sees another's trade or the dealer's book. A reporting service submits each trade to a trade repository as the applicable regime requires and records the acknowledgement. A calculation agent computes valuations and margin amounts offchain from the same market data and submits them to the contract. Variation margin moves as eligible tokenized collateral within the dealer domain. Initial margin sits in per-fund segregated accounts in a custodian's domain, from which Bank D cannot withdraw it.

On each payment date, the contract obtains the closing prices, dividends, adjustments, and fixings from the market-data service, computes the payoff, and converts it at the FX fixing set in the confirmation. Bank D settles the resulting conversion atomically against the shared liquidity pools on the main domain, as in Section 7.1, and bears any difference to the fixing. It batches conversions across funds per currency, which limits what the main domain reveals about individual payoffs. A margin default follows the CSA's notice and grace periods as contract timers, and close-out follows the agreed procedure.

The example depends on regulatory acceptance of the tokenized collateral under the regime that applies to the swaps (which differs for single-name and index underlyings), a compliant offering, legal opinions on netting and collateral enforceability, a licence from the exchanges or vendors for the data the service redistributes, and enough tokenized cash and FX liquidity in each currency. We note that the Issuer, Dealer, and Custody domains could be a domain that the entities jointly operate, with each controlling their respective contracts.

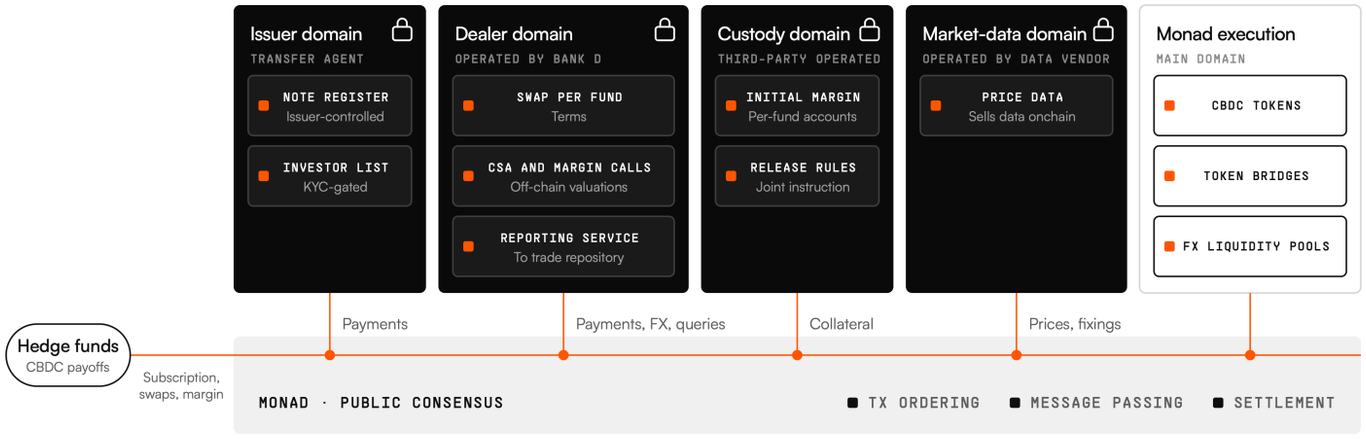


FIG. 7 Swaps on a tokenized equity-linked note, with the issuer, dealer, custodian, and market-data service each running their own domain and settling on the main domain.

7.3 Service Composition

A private domain integrated with a blockchain offers unique capabilities and enables new kinds of applications because data in them is confidential and they can bridge to Web2 services. Taking the previous example further, organizations can expose specialized functionality through smart contracts deployed within their sovereign domain.

Contracts on other domains can consume and pay asynchronously for products and services via cross-domain messaging. Autonomous services can thus be composed into complex multi-domain workflows while participating entities maintain absolute sovereignty over their proprietary state, execution logic, and operational infrastructure, as illustrated in Figure 7. Example services could provide information such as stock prices or perform computation with data that needs to be private; for instance, data may include transaction screening, eligibility proofs, checking credit information, or travel rule data exchange. Today, all of this is done offchain because of the sensitivity and/or value of the data.

As an example, consider a lender, Bank C, that wants a credit assessment of a loan applicant. The assessment is produced by a scoring service that runs in its own PSD, operated by a specialized analytics provider, and combines inputs from two data providers. The first, a credit bureau, operates a PSD that holds repayment histories. The second is a bank that sells, subject to the customer's consent, aggregated cash-flow indicators from a domain of its own. Each of the four parties runs its own domain with its own operators and access control policies. The flow is as follows. Bank C's contract sends a request message to the scoring service contract, together with a fee in a stablecoin that is escrowed in a locking contract on the main domain. The message includes sufficient information on the loan application (which is possible as all communication is encrypted). The service contract authenticates the request, checks the applicant's information and consent attached to it. It then sends one message to each data provider, each carrying an escrowed share of the fee (managed by the locking contract on the main domain). The data provider contracts verify via their policy enforcers that the service contract is an authorized reader, compute their answers, and reply with a message encrypted under the scoring domain's public key. The scoring contract combines the inputs and sends the result to Bank C. Upon receiving it, the locking contract on the main domain releases the escrowed fee to the service provider and the two data providers, in the agreed proportions. If any party fails to respond, the escrow is refunded, so no one pays for a service it did not receive and no one delivers data without payment. Beyond credit scoring, the same pattern applies to any service whose value lies in an answer rather than in the underlying data, such as reusable KYC attestations, market quotes released only to approved counterparties, fraud flags drawn from a consortium's records, and oracle feeds for real-world events.

This kind of composition has been impractical until now, for four reasons. On permissioned chains such as Canton or Hyperledger Fabric, parties that are not on the same chain and therefore cannot interact, and payment settles in a separate system. On public chains, every request, every answer, and every price would be visible to competitors. Private rollups keep data confidential but, because each runs its own sequencer, cannot call into each other. And outside blockchains, the same flow needs bilateral contracts, API keys, invoicing, and reconciliation with each provider. PSDs remove these obstacles.

This example clearly shows the benefits of Monad Private Settlement Domains and their integration into the Monad network. Data owners can monetize data that is sensitive and could not otherwise be shared, because they release only answers, only to authorized callers, and only against payment. Consumers integrate once, with a message interface and a settlement mechanism, instead of once per provider. New providers or services can be added by adding an address to the application, and existing ones can be replaced without redesigning the flow. Each participant remains sovereign over its own domain, including where its operators are located, which supports compliance with data-residency rules. Finally, because all domains settle against the same main domain, revenue splits and refunds are enforced by code and every party can audit its own payments.

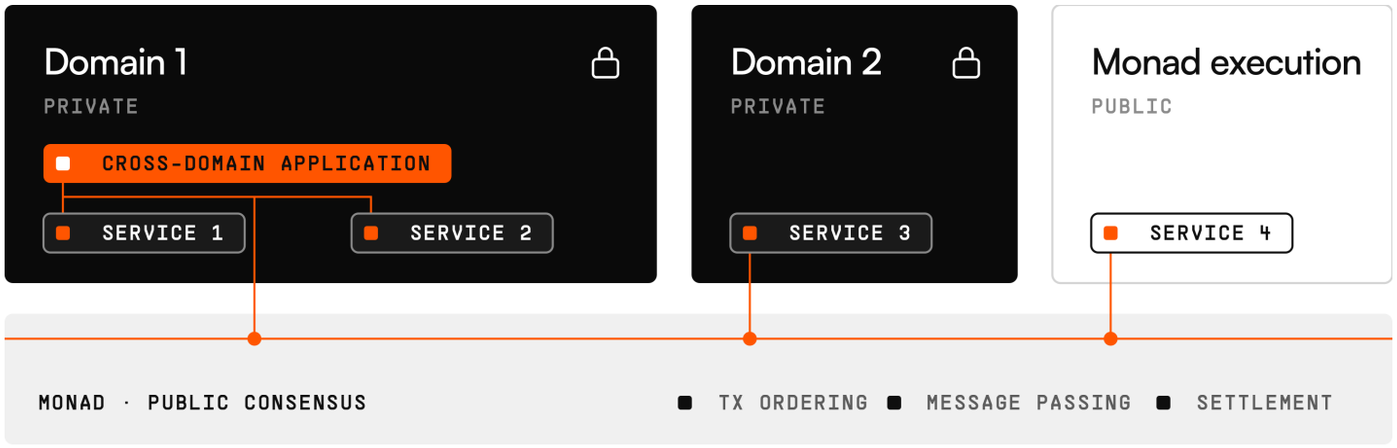


FIG. 8 A cross-domain application on Domain 1 is composed of four service contracts from private and public domains.

08 Planned Features of Private Settlement Domains

We have described the first version of Monad Private Settlement domains in this document. There are many features we wish to add in the future to extend PSDs to offer even better privacy, and address a wider range of use cases. This section discusses some of these features to give the reader a preview of what is coming and to solicit feedback on which of them should be prioritized for the next version.

Gas payment. Because all Monad PSD transactions are submitted to the Monad network, users need a main domain account to pay for them. This can be a substantial hurdle: institutions often cannot hold volatile cryptocurrencies, and end users find it cumbersome to obtain and hold cryptocurrency for gas payments. We will incorporate gas sponsorship mechanisms that allow the receiving Monad PSD to pay for the gas. Another solution is to allow users to pay fees in a stablecoin.

Confidential cross-domain token transfers. Token transfers in the current design involve the ERC-20 token ledgers on the main domain and the token bridges for the respective PSDs. While this keeps the sender's and recipient's account addresses hidden, the token amount is necessarily public, as each domain has a separate token bridge and tokens are moved between the addresses of these two bridges. To address this, we will provide confidential token transfers between the token bridges of different PSDs using cryptographic techniques that encrypt balances and transfers.

Cross-domain addresses. In the current design, messaging between smart contracts on different chains involves new calling interfaces. This is necessary because the normal call context assumes all contracts are on the same chain, so contract addresses are local. In our setting, however, the caller and receiver can be on different domains, and their addresses might not be unique: the identity of the chain each one runs on is important information as well. Transmitting both the caller contract's address and its chain identity therefore requires the receiving contract to implement a new calling interface. In the future, we want to support a better scheme, such as encoding both components in the address itself, following the approach that Arbitrum and Optimism have taken.

Data availability layer with jurisdictional awareness. An important missing component is a privacy- and location-aware data availability (DA) layer. Such a layer will allow a PSD to ensure that data remains available, while also enabling domain operators to protect and handle data in a regulatory-compliant manner. This DA layer needs to

support encryption (or, more precisely, decryption of encrypted data by authorized parties) and ensure that data is only ever stored in locations that comply with applicable privacy and other legislation.

Backup and recovery mechanisms. If all of the operators of a Monad PSD disappear, its state and all the funds it manages can no longer be accessed. One would therefore want a means to recover the state and resume operation, possibly with a different set of operators, and/or a means for users to recover their funds. Fortunately, there are a number of ways to achieve this using cryptography and governance protocols.

Web2 integration. The target use cases of PSDs include applications run by institutions and enterprises. In such cases, integration with legacy systems, including identity management, is important. A first step in this direction is our work on flexible authentication, which supports key management, two-factor authentication, and account recovery. Other integrations include communication with Web2 systems, which typically requires API keys for authentication. Because PSDs are private (and, for additional protection, can run inside TEEs), API keys can be stored onchain. Furthermore, PSDs provide asynchronous messaging, which enables message passing between external systems and contracts. Together, these features enable PSDs to integrate with legacy systems and access Web2 systems. To enable this, we will implement the necessary communication patterns in the PSD clients. This will also make KYC workflows smoother.

Improving cross-domain interactions. We have made a number of simplifications in the design of PSDs that we will address in the next version. The first is that the block rates of the main domain and all PSDs are synchronized, which means all PSDs must process blocks as fast as the main domain. While this is generally not a problem, as PSDs are expected to have less load, it prevents the use of zkVMs, which incur higher latency. We will decouple the block rate in the next version. This will improve the back-pressure mechanisms so that if a PSD is under load, the load can be managed, for example by increasing the gas price for transactions sequenced for that PSD, similar to the main domain's base fee mechanism (in place exactly for congestion control). Another possible improvement in this area is treating cross-domain messages separately from user transactions. Such designs will ensure that when a group of PSDs is under load, they can prioritize clearing that load whilst accepting new transactions perhaps only at a reduced rate.

Supporting different state attestation mechanisms. In Section 6, we discussed different ways to implement a PSD. In the first version, only multi-signature attestation is supported. In the next version, we will support attestation with zkVMs and TEEs. We stress that it is, of course, already possible to run the PSD client inside a TEE; what is missing is a check by the hub contract that the submitted state root was indeed computed inside a TEE. Furthermore, at the time of writing, Taceo has already integrated their multiparty protocol solution as a PSD, demonstrating that our design is sufficiently generic and flexible.

Operator and key management. The public key of a PSD is provided as part of its configuration when the PSD is created. Currently, this public encryption key and the identities of the domain operators cannot be modified. We will address this issue in a future version.

Less inter-domain trust. PSDs trust the main domain for the authenticity of message senders. Once a user has sent a token to a PSD, the domain has custody of that token. Using cryptography, these trust assumptions can essentially be removed if applications require it.

09 Conclusion

Public blockchains offer global, resilient, and deeply liquid settlement infrastructure, but their emphasis on total transparency rules out much of what institutions and enterprises do. Permissioned networks restore confidentiality, but at the cost of fragmented liquidity, bespoke integrations, and an operator whose governance and liability remain undefined. Monad Private Settlement Domains (PSDs) remove this trade-off. A PSD is a sovereign, confidential execution domain run by operators chosen by its creator, while sequencing, state settlement, and cross-domain messaging remain on the Monad blockchain. They are general-purpose technical infrastructure designed to support private, configurable confidential execution domains. The availability of privacy, access-control, and settlement functionality does not independently establish compliance with applicable legal or regulatory requirements. Rather, they are the basis on which such requirements can be addressed. Domain creators, operators, and application providers are responsible for determining and implementing the controls appropriate to their specific use case, activities, and jurisdictions.

The design of Monad Private Settlement Domains achieves the integration of private domains with the Monad public network by a hub contract on the Monad blockchain as the main domain and a spoke contract in each connected private domain. It requires no change to the Monad protocol. Transactions are encrypted, sequenced publicly, and executed privately. Operators attest to their domain's confidential state root to the hub, and the hub authenticates every outbound message of the domain against them. The resulting asynchronous messaging lets contracts compose across domains and with public liquidity. Private Settlement Domains come with access control at both the RPC and execution layers, and contract owners decide who can read their state and interact with their contracts. Because the design only prescribes how domains interact with each other and with the main domain, PSDs can be implemented in different ways addressing different requirements, making use of operator committees, zkVMs, and TEEs.

The discussed use cases show what our design makes possible. Central banks and commercial banks can run wholesale CBDC and cross-border payments under their own jurisdictions yet settle atomically against shared liquidity. Enterprises can run treasuries, asset managers can run funds, and market makers can run OTC desks, each without exposing their positions. Services in different domains can be composed into applications with the services getting paid, lowering the bar to monetize services.

What is presented in this white paper is the first version of PSDs and as such has some limitations that will be lifted in future versions. The forthcoming features described in Section 8 address these limitations, among them confidential cross-domain transfers, decoupled block rates, zkVM and TEE attestation, a jurisdiction-aware data availability layer, operator and key rotation, backup and recovery, and gas payment in stablecoins. We invite feedback from institutions, regulators, and developers on which of these to prioritize.

Monad PSDs let institutions and enterprises build on the foundation of a public high-performant network without giving up the confidentiality and control that their obligations require. Instantiating a PSD in the simplest case requires only issuing a creation transaction to specify the domain's configuration, designating its operators, and establishing access control parameters. Once initialized, contracts can be deployed in the domain to create the desired application that then executes confidentially, interacting natively with the public domain and other PSDs across the network.

References

[1] Amit Agarwal, Sourav Das, Babak Poorebrahim Gilkalaye, Peter Rindal, and Victor Shoup. BTX: Simple and efficient batch threshold encryption, 2026. https://eprint.iacr.org/2026/754.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, 2018.

[3] Avalanche Foundation. Acp-77: Reinventing subnets, 2024. https://github.com/avalanche-foundation/ACPs/tree/main/ACPs/77-reinventing-subnets.

[4] Aztec Labs. Aztec protocol documentation. https://docs.aztec.network/.

[5] Kushal Babel, Fatima Elsheimy, Lioba Heimbach, Mohammad Mussadiq Jalalzai, Tobias Klenze, Jovan Komatovic, Jason Milionis, Mike Setrin, and Victor Shoup. Cadence: Extreme pipelining with multiple concurrent proposers, 2026. https://arxiv.org/abs/2607.02275.

[6] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC), pages 81—91, 2022.

[7] Andrei Constantinescu and Lioba Heimbach. Scheduled transactions: Letting contracts act on their own, 2026. https://www.category.xyz/blogs/scheduled-transactions.

[8] Digital Asset. Canton network: A network of networks for smart contract applications, 2024. https://www.digitalasset.com/hubfs/Canton/Canton%20Network%20-%20White%20Paper.pdf.

[9] Mohammad Mussadiq Jalalzai, Tobias Klenze, and Jovan Komatovic. Quantum Cadence, 2026. https://arxiv.org/abs/2610.32803.

[10] Jae Kwon and Ethan Buchman. Cosmos: A network of distributed ledgers. https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md.

[11] Matter Labs. Zksync Prividium: Overview and architecture. https://docs.zksync.io/zk-stack/prividium/overview.

[12] Offchain Labs. Arbitrum nitro: A second-generation optimistic rollup, 2022. https://docs.arbitrum.io/nitro-whitepaper.pdf.

[13] Optimism Collective. Op stack specification. https://specs.optimism.io/.

[14] Tempo Labs. Tempo zones protocol specification, 2026. https://github.com/tempoxyz/zones/blob/main/specs/spec.md.

DISCLAIMER

This document is provided for informational purposes only. It does not constitute legal, financial, investment, tax, or regulatory advice, and it is not an offer or solicitation to buy or sell any asset or to use any product or service. Nothing in this document is a representation or warranty, and no one should rely on it in making any decision. Certain statements describe features that are under development or subject to change and are forward-looking; there is no assurance they will be delivered as described. This document is not intended for distribution in any jurisdiction where doing so would be unlawful.