# Shared State Trees - SST's

Lasse Herskind

November 17, 2023

**Abstract**

This document provides a description of Shared State Trees (*SST*'s). *SST*'s are data structures that allow state information to be accessed in both private and public execution contexts. This document will cover the basic structure of *SST*'s, explaining how they are built and used. It also addresses common challenges and offers solutions to improve their performance, trying to be a useful resource for those interested in implementing or understanding *SST*'s.

## Contents

# 1 Introduction

In the Aztec system, while both *public* and *private* execution environments operate with distinct state management (*public* and *private* states), there are scenarios where accessing data across both domains is advantageous. Such shared access is particularly useful in cases like:

`Address Registry`: Enabling contracts to interact with other contracts more easily requires address storage accessible in both *public* and *private* executions.

`Immutable Values`: Certain constants, such as specific contract addresses, benefit from being stored in a single, unchanging location for consistency across both domains.

`Access Control`: Managing privileges in contracts, such as a token contract owner's ability to mint new tokens, is streamlined when control information is shared between *public* and *private* executions.

This concept is encapsulated in what we term a *Shared State Tree* (SST), designed for data that is non-sensitive yet requires persistence and cross-domain accessibility.

**This document will:**

- Present a basic *SST* model facilitating "shared" storage between *public* and *private* domains for individual contracts, but leaking contract addresses in private executions.
- Discuss enhancements to this model to obscure contract addresses in private executions, along with associated challenges.
- Describe the development of a 'slow update' *SST* variant to address these challenges.
- Explore performance optimizations for the slow update *SST*.
- Provide an overview of adapting the slow update *SST* for handling "immutable" values.
- Discussing whether *SST*'s should be enshrined or not.

# 2 Single layer SST

Generally, the design relies on vector commitments where the values and commitment are publicly available in such a manner that they can be easily read in *public* and used for inclusion checks in *private*.

Logically, you can think of a contract specific *SST* as what is shown in `Figure 1`. Where we have a commitment $\mathcal{C}_m$ to a set of values $V_{m,j} \forall j$ for a specific contract $m$.

> **Comment 2.1: Current design**
>
> In the current design, we are using Merkle trees to commit to the values.

## 2.1 Reading a value from the SST

**In public:** To read in public, we can simply read the value $V_{m,j}$ directly from state (be it public storage or an enshrined tree).

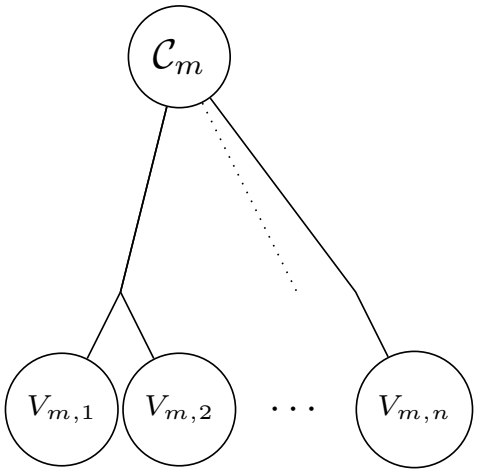



Figure 1: A "tree" committing to the "shared state variables" for a specific contract. We refer to the value $\mathcal{C}_{m,j}$ as $V_{m,j}$ to more easily differ between the commitments and values, and have $\mathcal{C}_m$ be the commitment to the values $V_{m,j} \forall j$.

**In private:** We can provide a membership proof for $V_{m,j} \in \mathcal{C}_m$ to show that the value is indeed part of the commitment, and then check that the commitment matches the one stored in the state. One issue of this, is the fact that changes to $\mathcal{C}$ will invalidate our membership proof, and make our read fail (we will refer to this as `Comment 2.2`).

**Comment 2.2: Invalidating membership proofs**

Relying on a provided membership proof makes the proof susceptible to invalidation whenever the tree is updated.

If the values are changed often this can be a serious problem for the contract, as it will invalidate every pending transactions that is trying to read from the tree. However, for most uses this is not an issue since it should not be updating the values in the tree often.

## 2.2 Updating a value in the SST

To update the value $V_{m,j}$ to $V'_{m,j}$, we can simply update the value in the state, and then update the commitment $\mathcal{C}_m$ to reflect the new value. This is similar to how we would normally update a commitment such as a Merkle root.

Generally, the update relies on a two simple steps:

1. Prove that $V_{m,j} \in \mathcal{C}_m$, e.g., the current value in the state.
2. Compute the new commitment $\mathcal{C}'_m$ where $V_{m,j}$ is replaced by $V'_{m,j}$.

While the logic of updating the *SST* is relatively simple this way, it have some important considerations. For one, if multiple actors are able to write to the *SST* they can have race-conditions depending on how the data is provided for the membership proof (recall `Comment 2.2`).

Nevertheless, for the case of a *SST* for a specific contract, it might be acceptable to not mitigate the issue completely, hence it is expected to apply some access control for privileged writes. The use case should allow us to accept this flaw as only one entity would likely be updating the values and it is not expected to happen often.

This way an update need just update $2$ values, the value ($V_{m,j}$) itself and the commitment ($\mathcal{C}_m$).

### 2.3 Issues

Let summarize the issues we encounter from this design, whether it is acceptable flaws can be decided separately.

**Contract leakage:** Since we have a contract specific *SST* to lookup, we will be leaking what contract is doing a lookup. In many cases, this is undesirable since it is breaking some of the guarantees of the system. You might not leak that it is your transaction, but heuristics becomes more powerful, especially if linking to non-blockchain data.

**Membership invalidation:** If the membership proof used in validation is user-provided, any changes to $\mathcal{C}_m$ will invalidate that proof.
This have two important implications:

- It invalidates private reads
- It invalidates updates

Meaning that if two actors are trying to update the values of the same contract at the same time, the first one to finish will invalidate the proof of the second one!

The issue of invalidating *updates* can somewhat mitigated through two different methods of coordination:

`Store more data`: If we can ensure that all the information required to perform an update is available in public state, such that only the new value is required as input, we can coordinate the updates entirely trough the public VM. The DA[1] requirements for this approach depend on the commitment scheme used.

`Enshrine the tree`: We can use the sequencer to coordinate thee updates similarly to how it coordinates the nullifier tree. Essentially, let the sequencer provide the membership proofs, while still only storing the leaves and the commitment.

Note that while these two approaches address the issue of invalidating updates, they do *not* address the issue of invalidating reads!

> **Comment 2.3: Tree implementation**
>
> Currently the tree used is a sparse depth $254$ tree, but for efficiency we should look at making a variation of the successor merkle tree.

## 3 Multilayer SST

To mitigate the issue of leaking specific contract that whose *SST* we are reading, we can adopt a multilayered approach. Essentially, create a separate *SST* which have the roots from the individual contracts as its leaves. This way, we can have a single commitment $\mathcal{C}$ for the entire tree, and have the individual commitments $\mathcal{C}_i$ as leaves.

[1] Data availability. Data that can retrieved by actors of the network. Each public state update require $64$ bytes of data.

By adding this extra layer, we can leak only the top commitment $\mathcal{C}$ (that everyone already knows) while still being able to read the values $V_{i,j}$ for a specific contract $i$. A visualization of this can be seen in Figure 2.

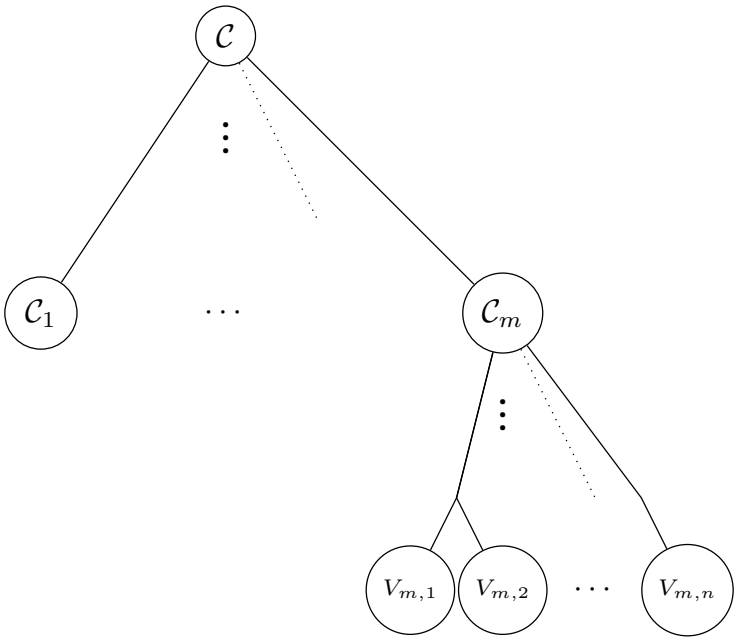



Figure 2: A multilayered "tree" of commitments. The leaves of the commitment $\mathcal{C}$; $\mathcal{C}_i$, which in turn are the collective commitment of the values $V_{i,j}$ for a specific contract $i$.

## 3.1 Reading a value from the SST

**In public:** To read in public, we can simply read the value $V_{i,j}$ directly from state (be it public storage or an enshrined tree).

**In private:** We can provide a membership proof for $V_{i,j} \in \mathcal{C}_i$ to show that the value is indeed part of the commitment, and then a second membership proof for $\mathcal{C}_i \in \mathcal{C}$ to show that the contract *SST* is in the state. As earlier, we are susceptible to the issue of Comment 2.2 from any $\mathcal{C}_i$ changing under our feet while our transaction is pending.

## 3.2 Updating a value in the SST

While much of the updates stays the same, we need to update the top-level commitment $\mathcal{C}$ as well.

1. Prove that $V_{m,j} \in \mathcal{C}_m \wedge \mathcal{C}_m \in \mathcal{C}$, e.g., the current value is in the state.
2. Compute the new commitment $\mathcal{C}'_m$ based on the new value $V'_{m,j}$.
3. Compute the new commitment $\mathcal{C}'$ based on the new $\mathcal{C}'_m$.

## 3.3 Issues

Since the membership proofs for $\mathcal{C}_m \in \mathcal{C}$ are now dependent on changes in *other* contracts, it is much more likely to run into the invalidation issue introduced in subsection 2.3[Issues].

This means that it is practically impossible to use the *SST* for reads in private execution since the membership proofs will be invalidated by changes in other contracts.

**Enshrining don't save us:** Note that reads will be invalidated *even* if the updates are addressed through enshrining the tree into the protocol.

# 4 Slow update SST

To mitigate the race-condition issues from the multilayered *SST* we introduce the idea of having a "delayed" or "slow" update. Logically this means keeping track of two trees, one for the current values and one for the pending values. Whenever an *epoch* has passed, we replace the current tree with the pending tree, and start a new pending tree. A visualization of the structure can be seen in Figure 3.

This way, we can ensure that the values are stable throughout the epoch, and that the membership proofs are not invalidated by changes in other contracts more than once every epoch.

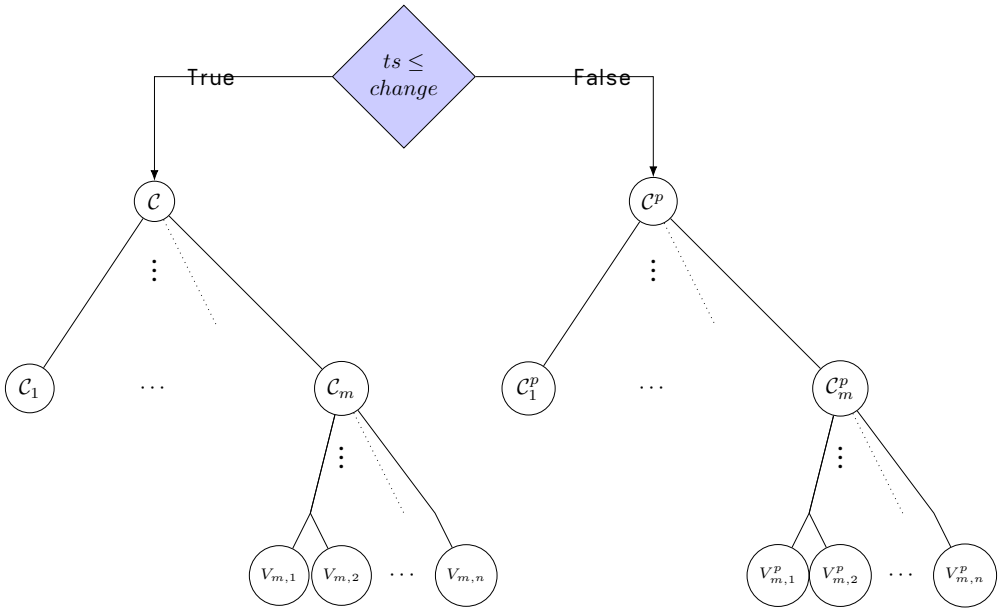



Figure 3: Logical representation of the "Slow" *SST*. Keeping track of a "current" and a "pending" tree. Pending tree marked with a superscript $p$, e.g., $\mathcal{C}^p$.

From the developer side of things, this means that updates will *not* take effect immediately, which might be a little odd to work with initially, but should be fine when you get used to it.

## 4.1 Reading a value from the SST

**In public:** To read in public, we can simply read the value $V_{i,j}$ directly from state (be it public storage or an enshrined tree).

**In private:** We can provide a membership proof for $V_{i,j} \in \mathcal{C}_i$ to show that the value is indeed part of the commitment, and then a second membership proof for $\mathcal{C}_i \in \mathcal{C}$ to show that the contract *SST* is in the state.

Contrary to earlier, we are not as susceptible to Comment 2.2 from a $\mathcal{C}_i$ changing under our feet while our transaction is pending since we are checking against the current commitment $\mathcal{C}$. However, if our proof used $\mathcal{C}$ and time have progressed enough for the the epoch to have have ended moving the tree to $\mathcal{C}^p$ our proof will be invalidated anyway. Invalidation once every epoch is nevertheless better than every time any contract is updated.

### 4.2 Updating a value in the SST

Updating the values in the *SST* change a little because of the epoch update, but otherwise stays mainly the same.

**Developer** From the developer point of view, the update is very close to the earlier steps, with the main difference being that they are updating on the pending tree instead of the current tree. This means that the update will not take effect until the next epoch.

1. Prove that $V^p_{m,j} \in \mathcal{C}^p_m \wedge \mathcal{C}^p_m \in \mathcal{C}^p$, e.g., the current pending value is in the pending state.
2. Compute the new commitment $\mathcal{C}^{p'}_m$ based on the new value $V^{p'}_{m,j}$.
3. Compute the new commitment $\mathcal{C}^{p'}$ based on the new $\mathcal{C}^{p'}_m$.

As earlier, the issue of how to provide the membership proofs persist, but can be addressed with more store or enshrining.

**Sequencer** At the beginning of a new epoch, the sequencer will update the commitment $\mathcal{C}$ to the new value $\mathcal{C}^p$ and start a new pending tree $\mathcal{C}^p$. The values retrieved from the tree shall reflect the ”current” values, e.g., the values from the current epoch $\mathcal{C}$. The new tree $\mathcal{C}^p$ should be initialized as equal to $\mathcal{C}$.

### 4.3 Issues

While we have limited the issues of invalidating reads greatly, actually making it possible to use the *SST* for reads in private execution, we still have the issue of invalidating updates due to changes in other contracts as discussed in subsection 2.3.

Separately, the amount of storage changes require to keep track of two separate trees that are updated at the same time is very high. If done with public state could make it prohibitively expensive, and for enshrined it could greatly increase the complexity of the circuits to address the epoch update.

## 5 Practical Slow Updates

To mitigate the number of storage updates and/or complexity of the two tree approach of section 4 we can make a special variant of a tree that is optimized for slow updates. In this variation, we amortize the cost of the epoch swap across the updates.

Namely, instead of storing only the value $V_{m,j}$ at the leaves we will store a tuple $(b, a, c)$. Where $b$ is the value *before* or at time $c$ and $a$ the

value *after* $c$. By applying conditions at the time of read, we can implement this without requiring any storage updates at epoch changes, but doing a bit extra work at each update. This makes it more practical for us since the individual proofs are much smaller.

Essentially, we define a leaf as $X = (X^b, X^a, X^c)$, where $X^b$ is the value before and $X^a$ the value after and $X^c$ the timestamp of the change. We can then read the "current" value using $\rho$ as outlined in `Equation 1`. We denote the global variables as $\mathcal{G}$ and the global timestamp as $\mathcal{G}_{ts}$):

$$\rho(X) = \begin{cases} X^b & \text{if } \mathcal{G}_{ts} \leq X^c \\ X^a & \text{otherwise} \end{cases} \tag{1}$$

We use $(0,0,0)$ as the empty leaf and initialize $\mathcal{C}_i \forall_i$ as $(\chi, \chi, \text{max})$, where $\chi$ is the commitment of the empty tree (tree filled with empty leafs) and max the maximal timestamp. $\mathcal{C}$ is then the commitment to all of these commitments.

The commitments $\mathcal{C}_m$ and $\mathcal{C}$ follow a similar structure to the leafs and can be read with $\rho$ in a similar fashion. The main difference being on how the values are updated. For the commitments, we will update the before and after such that the $b$ will be the commitment to the values if building a tree passing $\mathcal{C}_m^c$ as the global timestamp and $a$ the commitment to the values if building a tree passing $\mathcal{C}_m^c + epoch$ as the global timestamp.

This way, we can update as if we had two trees without needing to copy values around at epoch changes.

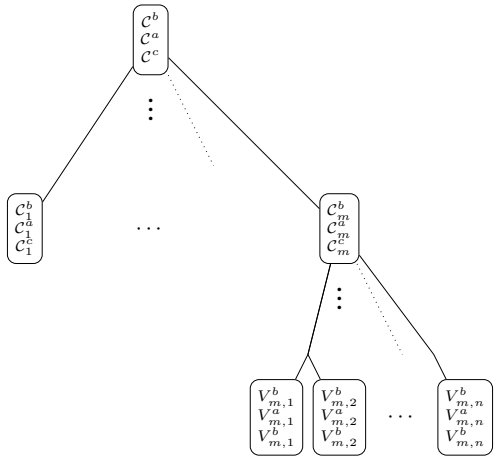



Figure 4: A modified slow updates *SST* that better lends itself to addressing the issue of epoch updates. Each node in being a tuple specifying a before, after and change value.

## 5.1 Reading a value from the SST

**In public:** To read in public we read the leaf tuple, and return the value based on `Equation 1`.

**In private:** We can provide a membership proof for $V_{i,j} \in \rho(\mathcal{C}_i)$ to show that the value is indeed part of the "current" commitment, and then a second membership proof for $\rho(\mathcal{C}_m) \in \rho(\mathcal{C})$ to show that the contract *SST* is in the state. As in the naive slow updates *SST*, proofs will be invalidated whenever we enter a new epoch.

## 5.2 Updating a value in the SST

When updating the value of a leaf $V_{m,j}$ we need to:

1. Updating the values to ensure they are ”current”:
   (a) Update the value $V^b_{m,j}$ to the value returned by $\rho(V_{m,j})$;
   (b) Update the value $C^b_m$ to the value returned by $\rho(\mathcal{C}_m)$;
   (c) Update the value $C^b$ to the value returned by $\rho(\mathcal{C})$;
2. Prove that $V^b_{m,j} \in \mathcal{C}^b_m \wedge \mathcal{C}^b_m \in \mathcal{C}^b$ to ensure that the before value was updated correctly.
3. Prove that $V^a_{m,j} \in \mathcal{C}^a_m \wedge \mathcal{C}^a_m \in \mathcal{C}^a$, ensuring that ”current after” is part of the ”after” tree,
4. Update ”after” and prove correctness of update
   (a) Update the value $V^a_{m,j}$ to the new value $V^{a'}_{m,j}$;
   (b) Use the membership proof for $V^a_{m,j} \in \mathcal{C}^a_m$ to update the commitment $\mathcal{C}^a_m$ to become $\mathcal{C}^{a'}_m$ which replaces $V^a_{m,j}$ with $V^{a'}_{m,j}$.
   (c) Use the membership proof for $\mathcal{C}^a_m \in \mathcal{C}^a$ to update the commitment $\mathcal{C}^a$ to become $\mathcal{C}^{a'}$ which replaces $\mathcal{C}^a_m$ with $\mathcal{C}^{a'}_m$.
5. Compute the next epoch timestamp as $ts' = \left(\frac{ts}{epoch} + 1\right) \cdot epoch$.
6. Update timestamps:
   (a) Set $V^c_{m,j} = ts'$;
   (b) Set $\mathcal{C}^c_m = ts'$;
   (c) Set $\mathcal{C}^c = ts'$;

As earlier, these updates are susceptible to the issue of how to provide the membership proofs persist, but can be addressed with more storage or enshrining[2].

### 5.2.1 Example of updates

Sine the logic for each layer is practically the same. We show with one layer to keep the figures a bit simpler and easier to follow (also easier to fit on the page). In Figure 5 we go from the initial tree ($\mathcal{C}_\mathcal{I}$), and inserts $1$ at index $1$. We are using $1$ as the value of choice since it can emulate a flag such as being a ”minter” or similar.

When performing a lookup or membership for $V_1$ in this tree fetch the value $\rho(V_1)$ and prove membership in $\rho(\mathcal{C})$.

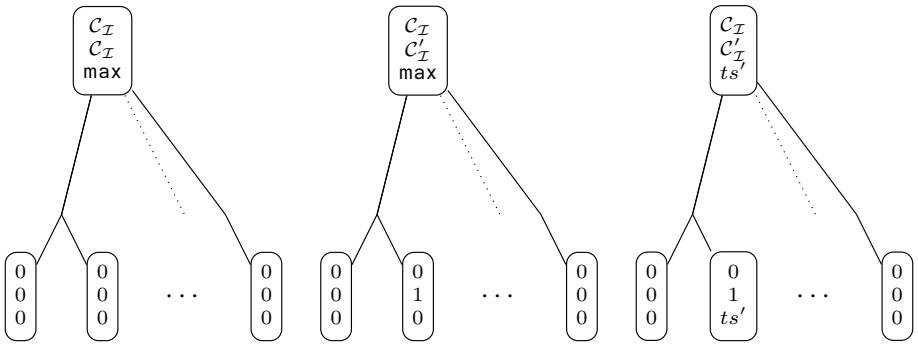
**Comment 5.1: Code-reuse**

Implementation wise, we can keep track of the pending updates using our existing sparse tree and then wait until the epoch has passed to `commit` the data. That way using the pending information will give you ‘after‘ while using the committed data will give you ‘before‘.

In Figure 6 we let the duration of the epoch pass, and when in the new epoch, we will update index $0$ to contain the value $1$.

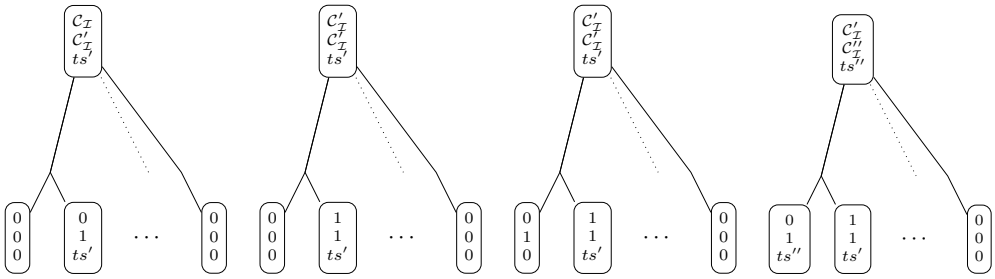
As seen from the figure, only the leaf at index $0$ and the commitment $\mathcal{C}$ have changed to reflect the new value.

[2]Might be solvable with a different commitment schemes as well.




(a) Initial state (b) Step 4. (c) Step 6.

Figure 5: Initial insertion of the value $1$ into the state at index $1$.




(a) Initial state (b) Step 1. (c) Step 4. (d) Step 6.

Figure 6: Inserting the value $1$ into the state at index $0$ after time $ts'$.

**Comment 5.2: A note on Data Requirements**

If we need to keep track of the intermediates in state, I think we need a full leaf. That would be $3 * 64$ bytes per node if in public state, so for a depth $32$ tree that would be $6144$ bytes.

**Comment 5.3: Implementation**

Beware that the current noir implementation is not complete, but rather a single layer tree that is leaking the contract address.

### 5.3 Immutable extension

The *SST* can be updated to reject updates of values that are already set, thereby allowing us to have immutable values that are available in both private and public. The insertion is still delayed, but for things like constructor specified values it seems like a fair tradeoff.

## 6 Enshrine or not?

The decision to enshrine the *SST* in the protocol - that is, integrating it at the kernel and rollup circuit levels - warrants careful consideration of both its potential advantages and limitations.

**Rationale for Enshrining the *SST*:**

- **Complex Update Coordination:** As previously noted, managing updates for the top-level of the *SST* poses significant challenges due to membership invalidation issues - enshrining could streamline this process.

#### Benefits of Enshrining

`Reduced Execution and Data Costs:` Direct protocol integration could lead to more efficient handling of updates, reducing both execution and data costs. This efficiency comes from utilizing optimized circuits instead of the public VM and the sequencer's role in providing membership proofs.

`Contract Upgrades:` Enshrining the *SST* allows for direct use at the kernel level for purposes like storing contract class references, potentially simplifying contract upgrades[3].

#### Drawbacks of Enshrining

`Reduced Flexibility:` Integrating the *SST* into the protocol could significantly limit our ability to modify its design in the future.

`Increased Complexity:` The addition of the *SST* would complicate the kernel and rollup circuits, introducing another layer of complexity to manage.

`Limited Scope of Benefits:` The main advantage of enshrining the *SST* lies in update facilitation, with little to no improvement in read operations, hence these must occur in the application for flexibility.

`Uncertain Requirements:` Given the ongoing exploration of various *SST* variations (e.g., immutables, slow updates), the exact requirements are not fully understood, which complicates the decision to enshrine a specific implementation.

**Recommendation:** Given these considerations, especially the uncertainty around specific *SST* requirements and the potential loss of flexibility, I recommend **against** enshrining the *SST* at this stage. Instead, we should focus on research and optimizations (`section 7`), particularly aimed at making updates more efficient and less data-intensive. This approach maintains the protocol's flexibility, which seems paramount given our current understanding of *SST* use cases.

Regarding contract upgrades, while not enshrining the *SST* necessitates alternative solutions like `delegatecall` should be possible.

## 7 Future work

- The current noir implementation of a Slow updates tree should be extended to fully implement the practical *SST* (see `Comment 5.3`).
- Explore the use of other commitment schemes than Merkle trees to reduce the storage requirements. Example could be *Semacaulk*: https://github.com/geometryresearch/semacaulk/.
- Explore the use of queue for updates to reduce the storage requirements[4].

[3] Discussed in https://forum.aztec.network/t/contract-upgrades-and-shared-private-state/2393

[4] We can use a queue to make "mass" updates, but unless we can ensure that queue cannot be used to dos the updates the solution is unacceptable