dWallet System 1—Asynchronous

3MI Labs, Leuven, Belgium

22nd October 2024

Abstract

This document describes the first generation of the dWallet mainnet protocol for implementing 2PC-MPC over the Sui blockchain.

1 Overview

The dWallet mainnet protocol executes instances of the 2PC-MPC ECDSA signing protocol using the infrastructure of the Sui blockchain. The Validators executing this protocol make use of Narwhal and Bullshark [DKKSS22, SGSK22] to broadcast and obtain consensus on transactions and MPC messages.[1]

Note 1. In accordance with the code's use of the consensus protocol also as a broadcast protocol, this document uses the terms *broadcast rounds* and *consensus rounds* interchangeably.

Purpose and components. This system's primary purpose is to coordinate MPC Instances spawned by User transactions and to identify malicious Validators that should be ignored by future MPC Instances. This selection is done with the following processes:

1. A data structure is updated to monitor which Validators are alive or down and honest or malicious in the present moment.
2. A set of rules which determines whether down Validators should be added to the list of malicious Validators.

Malicious Validators. The list of malicious Validators is maintained according to the following rule(s):

1. If a Validator explicitly misbehaves during an MPC Instance, it is considered malicious until the end of the epoch.

This rule ensures that any Validator that was actively malicious is never allowed to interact in any further Instances.

New MPC Instances. Newly spawned MPC Instances should receive and, at the end of a consensus round, process MPC Messages as follows.

1. If they received at least q valid MPC messages from honest parties, they proceed to the second MPC round without waiting for more MPC messages;

[1] Bullshark can be replaced by Mysticeti [BCD+23] without changing this architecture.

These rules for MPC Instances imply that active Validators can choose to "register" for participation by sending valid MPC Messages at any round during the execution of an MPC Instance. If an instance terminates successfully, any Validator whose messages were included in the processing of the instance will receive rewards proportionally to the number of rounds it participated in. For more information about these rewards, and conversely penalization, please refer to Sec. 8.

1.1 Notation

This document uses the following notation:

$N \in \mathbb{N}^+$	The number of Validators in an epoch
$t = \lfloor \frac{N-1}{3} \rfloor$	A strict threshold on the number of tolerated malicious Validators
$q = \lfloor \frac{2N+2}{3} \rfloor$	The number of Validators required for quorum
$f \in [N]$	The number of malicious Validators in an epoch

Claim 1. For N, t and q as above, it holds that $q + t \leq N$.

Proof. By definition, $t \leq \frac{N-1}{3}$ and $q \leq \frac{2N+2}{3}$; therefore $q + t \leq N + \frac{1}{3}$. Since $t, q \in \mathbb{N}$, this implies $q + t \leq N$. □

2 Preliminaries and Assumptions

2.1 Malicious Validators

Assumption 1. During one epoch, at most t Validators may turn out to be malicious.

2.2 Broadcast Protocol

This specification assumes that the broadcast protocol is Narwhal [DKKSS22].

The terms used in our explanation are introduced as follows:

- **Validator**: a Validator is a node in the Narwhal network that plays the role of a Primary and/or Worker.
- **Primary**: a Primary in Narwhal is responsible for constructing a Directed Acyclic Graph (DAG) that consists of all transactions that have been disseminated in the Narwhal mempool. This DAG is the system's main data structure and is used for consensus, data retrieval, etc.
- **Worker**: a Worker in Narwhal subscribes to aid a unique Primary and is responsible for performing a low-level broadcast of transactions in the network in the form of *Batches*. The digests of these batches are then included by Primaries into blocks which constitute the DAG.
- **Batch**: the Workers construct a batch, which is a collection of transactions. Multiple batches are included in a block made by a Primary.
- **Block**: the Primary constructs a block, which is a collection of batch digests that it previously received from its workers. Additionally, it also contains links to previous blocks in the DAG.

A primary can have multiple Workers subscribed to it; however, a Worker must subscribe to only one Primary.

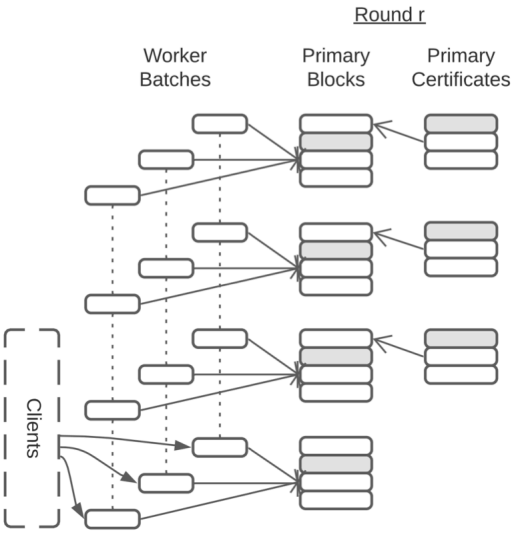

Figure 1: Communication flow in Narwhal

Following the invocation of the consensus engine in Sui, the entities in Narwhal communicate as depicted in Fig. 1. The communication in order to achieve broadcast of blocks is described as follows:

- Each protocol message sent by the clients to Narwhal is split into transaction data and metadata, and a load balancer ensures that transactions data are received by all Workers at a similar rate.
- Upon receiving this transactions data, each Worker creates a *batch of transactions*; given that each Worker receives transactions data at a similar rate, these batches can theoretically be constructed by all Workers in parallel.
- Upon batch creation, each Worker sends their batch to the Worker nodes of other Validators in the delegate committee of Sui.
- Workers then verify each batch they receive, and once verification is successful, they send an acknowledgment to the sender Worker node. The goal of these acknowledgments is to collect a quorum agreement that a super-majority (q) of these Workers have seen the same batches of transactions.
- Once a Worker receives a quorum of these acknowledgments on a batch, they send a hash of this batch to the Primary of their Validator.
- The Primary includes the hashes of these batches in their *block*, and broadcasts this block to the Primaries of all other Validators in the delegate committee of Sui.
- Each Primary then verifies the validity of this block; more importantly, a Primary signs a block only if it includes batches that the Primary's own Worker nodes have stored locally. In case a batch is missing in their local storage, the Primary instructs its Workers to pull the batch directly from the Worker of the block creator.
- Upon receiving a super-majority (q) of signatures from other Primaries, the block

creator generates a *certificate of block availability*, and re-broadcast this certificate to other Primaries.

Narwhal utilizes threshold clocks for round progression, i.e., each Validator only progresses from round r to round $r+1$, if they have received certificates of block availability from q distinct Validators.

Most batches are available to other Validators before the Primary block arrives, because Workers constantly create and share batches of transactions in the background. Additionally, since Primary blocks are much smaller when hashes of batches instead of Transactions are included, this ensures that transactions never suffer more than some maximum latency (which is usually the network latency itself).

2.3 Consensus Protocol

This specification assumes that the consensus protocol is Bullshark [SGSK22] which provides partially synchronous secure consensus. It further assumes that the output of Bullshark is produced in successive "commitments" where each output commits to a sub-graph of the Narwhal DAG, called a `CommittedSubDag` in the `narwhal/types` crate.[2] This sub-graph is composed of a sequence of Narwhal Certificates (of Availability) that has been ordered according to the Bullshark logic.

Together with the `CommittedSubDag`, the `ConsensusOutput` type (in the same crate) also contains the Narwhal Batches that are referred to by the committed Narwhal Certificates. These batches contain the transactions. We will at times refer to an object of the `ConsensusOutput` type as a "(consensus) delivery"

In the Sui codebase, the `ConsensusOutputAPI` trait is implemented for the Narwhal `ConsensusOutput` type[3] and gives access to the ordered sequence of transactions committed by this output (with the `transactions()` function) as well as some additional metadata:

1. The leader round: the number of the Narwhal round in which the leader block of this output was proposed. It is accessed by the `leader_round()` function. It is always increasing between one consensus output and the next, although not always by the same amount.

2. The index of the committed sub-DAG: a unique global index for each committed sub-DAG, computed by Narwhal and Bullshark and accessed with the `commit_sub_dag_index()` function of the `ConsensusOutputAPI` trait.

For this document, we will use the index of the committed sub-dag as an indication of "time" passing on the broadcast channel. Each index value will correspond to one "moment" that all Validators can agree on, even if it is received at different clock times by each Validator.

Replacing Bullshark by Mysticeti

Optimisation 1. *The Bullshark consensus protocol could be replaced by the Mysticeti protocol [BCD+23] which commits sub-graph more frequently and more efficiently.*

[2] All references to the code are made to commit `f05d65a0f6f5c03047193080c6422808aa82b61a` of the `dwallet-network` repository.

[3] `sui-core/src/consensus_types/consensus_output_api.rs:34`

3 Protocol Specification

The dWallet System protocol is built on top of Sui and contains the following major components that each Validator must run locally:

Transaction Handler Receives MPC transactions or aggregated MPC transaction certificates from users, checks them and passes them to other components.

This handling system already exists in Sui for processing User transactions and executing them (with or without sequencing). In Section 3.1 we describe how MPC transactions (or aggregated MPC transaction certificates) should be handled differently from existing Sui transactions.

Any other transaction type that is allowed by the dWallet Network (including, but not limited to, transferring tokens between addresses or creating new dWallet objects) are currently out of scope of this document. That being said, the transaction handler should only authorise transactions that are allowed by the dWallet Network, and ignore other types (e.g., arbitrary smart contracts).

Consensus MPC Delivery Handler Receives committed sub-DAGs from the consensus protocol and checks and filters the MPC-related transactions and messages within them, passing them to other components.

MPC Reputation Handler Based on information received from other components, keeps track of the MPC activity of Validators and maintains the list of Validators that have been identified as malicious. It also records whatever participation information is required for the purpose of reward computation.

MPC Instance Handler Based on MPC transactions and MPC messages received from the Consensus MPC Delivery Handler, moves active MPC instances forward and communicates their outputs to the MPC Consensus Sender and the MPC Reputation Handler.

MPC Consensus Sender Receives MPC messages, transactions or certificates from other components and submits them to the consensus protocol for broadcasting and ordering.

3.1 MPC Transaction Handler

In Sui, a transaction submitted by a User goes through two steps: Process (when it doesn't have an aggregated transaction certificate) and Execute (when it does). This sections describes how MPC transactions should flow through these steps.

3.1.1 Processing MPC Transactions

Before the validator issues a (partial) transaction certificate on a Sui transaction, they must check that it is "valid". MPC transactions are processed in the same way, for an appropriate definition of "valid".

3.1.2 Executing MPC Transactions

MPC transactions follow the "shared object path" of Sui: they must be sequenced by the consensus protocol before they can be executed. Even though MPC transactions could be considered for the "owned object path" because dWallet objects are not necessarily shared, the present system requires ordering of the transactions so that the Validators can synchronize and participate in the same MPC transactions at the same

time. Furthermore, in the Sui blockchain, execution of a transaction in the owned object path does not require input from multiple Validators, whereas this is required for MPC transactions, so other Validators need to be informed of the transaction's existence before being able to participate in its execution.

When the Transaction Handler receives an aggregated MPC transaction certificate from a User, it checks that the certificate is valid and then submits the aggregated MPC transaction certificate to the Consensus Sender according to the rules of Sui (i.e., by first checking which Validator should be responsible for submitting this aggregated certificate to consensus and giving that Validator the chance to submit it first).

If Sui specifies that some response should be sent to the User, then the Transaction Handler sends this response for aggregated MPC Transaction Certificates too.

3.2 MPC Consensus Delivery Handler

Recall that the `transactions()` function of the `ConsensusOutputAPI` returns a list[4] of ordered transactions contained in the committed sub-DAG. Every MPC Transaction contained in this list should be passed to the MPC Consensus Delivery Handler by the system that first receives the output from the consensus protocol. These must be passed in the same order as they are sequenced in by the consensus protocol and, furthermore, the processing of the MPC Transactions from one committed sub-DAG must be finished before the MPC Transactions from the next one can be passed to the MPC Consensus Delivery Handler.

The MPC transactions contained in such a delivery can be of the following type:

1. User Initiate MPC Transaction: this is an Aggregated Initiate MPC Transaction Certificate that (a) contains the User's first round message in one of the MPC protocols, (b) contains q Validator signatures that were aggregated by the User during the Process phase, and (c) was submitted to consensus by possibly multiple Validators as part of the Execute phase.
2. User Finalise MPC Transaction: this is an Aggregated Finalise MPC Transaction Certificate that (a) contains the User's last round message in one of the MPC protocols, (b) contains q Validator signatures that were aggregated by the User during the Process phase, and (c) was submitted to consensus by possibly multiple Validators as part of the Execute phase.
3. Validator MPC Message: this is an MPC Message that was submitted to consensus by a Validator as part of the execution of an MPC Instance.
4. Validator MPC Output: this is the output of the successful execution of an MPC Instance that was submitted to consensus by a Validator.

The next sections describe how each of these types of MPC Transactions should be processed by the MPC Consensus Delivery Handler.

3.2.1 User Initiate MPC Transaction

First, the 2PC-sid corresponding to this Initiate MPC Transaction is derived from identifying information about the User and this transaction (see Section 4).

If this 2PC-sid has already been seen during this delivery of MPC Transactions, then this Initiate MPC Transaction is a duplicate of one that was previously processed in

[4] The `ConsensusOutputTransactions` type is defined at `consensus_output_api.rs:14`

this delivery and must be skipped. If this 2PC-sid has not already been seen, then it is added to the list of 2PC-sids seen during this delivery.

Note 2. This method of checking the 2PC-sid means that every Validator will take the first 2PC-sid that it sees as the valid one. By Assumption 1, forks cannot happen in the consensus protocol and therefore the ordering of the transactions in the consensus deliveries will be identical in each honest Validator's local view, implying that they will agree on the order of appearance of the 2PC-sids.

Next, the MPC Consensus Delivery Handler queries the MPC Instance Handler to check whether this 2PC-sid is already known. If it is, then this MPC Initiate MPC Transaction is a duplicate of one that was processed during a *previous* delivery and must be skipped. Otherwise, the MPC Consensus Delivery Handler gives the Initiate MPC Transaction to the MPC Instance Handler so that the requested MPC protocol can be started and executed.

3.2.2 User Finalise MPC Transaction

A correctly formed User Finalise MPC Transaction links to the MPC-sid of the MPC instance that produced the output to which this Transaction is responding. The MPC Consensus Delivery Handler therefore checks whether the MPC Output for this MPC-sid has been recorded in the Object store.

The User Finalise MPC Transaction is given to the MPC Instance Handler for it to process the User's 2PC last round message and record the final update to the dWallet object in the Object Store.

3.2.3 Validator MPC Message

A Validator MPC Message should be 1. correctly formed (it contains at least a payload, a destination sid, a sender, and a signature) and 2. correctly authenticated (the signature verifies against the sender's public key).

If it is correctly formed and authenticated, the payload, destination sid and sender must be passed to the MPC Instance Handler so that it can deliver it to the appropriate MPC Instance.

Sending an invalid MPC Message is explicit malicious behaviour. If an honest Validator observes an invalid MPC Message, the MPC Consensus Delivery Handler informs the MPC Reputation Handler of the identity of the sender (being the identity of the Validator that proposed the Narwhal block in which the invalid MPC Message is contained) so that it can be marked as malicious for the rest of the epoch. The sender of the invalid MPC Message should also be considered malicious for the computation of the rewards.

3.2.4 Validator MPC Output

If the transaction is an MPC Output, it should be sent to the MPC Instance Handler so that the Output inside it can be checked for equality against the Output stored in the dWallet object in the Object Store.

Broadcasting an invalid MPC Output is explicit malicious behaviour. If an honest Validator observes an invalid MPC Output, the MPC Consensus Delivery Handler informs the MPC Reputation Handler of the identity of the sender (being the identity of the Validator who sent a correctly authenticated but invalid MPC Output transaction,

or if the transaction is not correctly authenticated, the identity of the Validator that proposed the Narwhal block in which the invalid MPC Message is contained) so that it can be marked as malicious for the rest of the epoch. The sender of the invalid MPC Output should also be considered malicious for the computation of the rewards.

3.3 MPC Reputation Handler

During each consensus output, the MPC Reputation Handler keeps track of the list of malicious Validators between consensus deliveries. It performs this role by maintaining one data structures:

1. A list of identities of Validators that have been identified as malicious.

After each consensus delivery has been executed (i.e., every MPC Transaction or Message has been processed), the MPC Reputation Handler inspects this structure and announces who the malicious validators are to MPC Instances.

3.3.1 List of Malicious Validators

The MPC Reputation Handler maintains a list of at most t Validators that have been identified as malicious. This list is updated by the MPC Instance Handler directly in the case of malicious messages.

In summary, a Validator is labelled as malicious if:

1. it sends an incorrect MPC Message during an MPC Instance.

If the list of malicious Validators grows to contain strictly more than t Validators, the Validators should trigger the end of the epoch preemptively.

3.3.2 Committee Selection

At the end of every delivery, the MPC Reputation Handler observes the liveness counters and the list of malicious validators and forms a committee that it communicates to the MPC Instance Handler. Any MPC Message sent by a Validator who is not on this committee should be ignored by MPC Instances.

Rules for committee selection:

1. malicious Validators must not be selected;

Optimisation 2. *If it is useful to know which Validator are likely to be alive in the* next *consensus delivery (e.g., for an optimisation of the Sign protocol which requires the identities of the parties in advance) then the MPC Reputation Handler can also make a list of the honest Validators that were active (i.e., who sent at least one message) in the* current *delivery.*

3.4 MPC Instance Handler

The MPC Instance Handler maintains lists of active MPC Instances. It receives ordered messages from the Consensus Delivery Handler which are either MPC transactions for which new MPC instances should be spawned, or MPC messages that should be delivered to running MPC instances.

When it receives an MPC transaction that should spawn a new MPC instance, the MPC Instance Handler initialises a new attempt counter to 1 for this transaction,

derives a 2PC-sid from the transaction and its counter (see Section 4), spawns a new MPC instance and adds it to the list of active instances (see Section 6).

When it receives an MPC message, it checks if it was sent by a malicious Validator. If it was, it drops the message. If it was sent by an honest Validator, it checks whether it is addressed to an active MPC instance. If it isn't, it drops the message. If it is properly addressed, it delivers the MPC message, together with the sender's identity, to the recipient MPC instance.

3.4.1 MPC Instance Processing

An MPC instance goes through the following states:

1. Initialise: MPC transaction, attempt counter and delivery counter are received as inputs.
2. First round message: the first message is computed and sent to the other Validators.
3. Waiting for a quorum of messages: the instance starts receiving, checking and saving messages. At the end of every consensus delivery, it performs end-of-delivery checks. If these fail, it stays in this state; if these pass, the instance moves to the next state.
4. Intermediary rounds: first compute this round's message based on the previous round's messages and then send it to the other Validators. If the computation of the message reveals a malicious message from the previous round, identify the sender as malicious to the MPC Reputation Handler. Otherwise, receive, check, and store this round's new messages until the end-of-delivery signal is received and then perform the end-of-delivery checks.

Optimisation 3. *Perform the quorum satisfaction check each time a new message is saved, instead of waiting until the end of the delivery.*

This means the processing of the next round's message can begin (slightly) sooner, but requires checking that new information coming from the rest of the delivery isn't missed.

Checking MPC messages. When an MPC Instance receives an MPC message from the MPC Instance Handler with the identity of its sender, it checks whether this is the first message from this sender for this round of the instance. If it is, it records the message. If this is not the first message from this Validator for this round of the instance, it checks whether this is a duplicate of the already-recorded message from this sender. If it is a duplicate, it ignores the repeated message. If it is not a duplicate, it identifies the sender as malicious to the MPC Reputation Handler.

If a message is successfully recorded, the MPC Instance informs the MPC Reputation Handler of the sender's contribution to this MPC Instance during this delivery.

End-of-delivery checks. When an MPC instance is notified that a delivery has ended, it performs the following checks:

- If a quorum of messages is received, proceed to the next round in the MPC protocol.
- If a quorum of messages is not already received, remain in the current round of the MPC protocol.

3.5 Consensus Sender

The Consensus Sender submits messages as inputs to the broadcast channel.

MPC Messages When an MPC Instance, identified by its 2PC-sid, submits a message, the Consensus Sender wraps it as a valid MPC Message and submits it as input to the broadcast channel as either a Validator MPC Message or a Validator MPC output.

4 MPC Session Identifiers

The initial 2PC-sid is derived when a User's MPC transaction is first executed and broadcast through the consensus protocol. This is used by Validators to index the MPC instances that are spawned to execute the corresponding MPC protocols.

4.1 Initial 2PC-sid Derivation

Derive2PCsid
1 : $\mathsf{sid}_{\mathsf{2PC}} \leftarrow \mathsf{Hash}(\mathsf{TxDigest}, \mathsf{attempt\#})$

In the asynchronous version, this 2PC-sid is sufficient because the transaction digest is guaranteed to be unique from the non-repeating object identifiers; combined with the collision-resistance of the hash function, this implies that the 2PC-sid will never repeat.

5 Latency Analysis (Narwhal + Bullshark)

This section brings together elements from Sections 2.2, 2.3 and 3 to analyse the latency of the proposed system. It assumes ideal network conditions and Validator participation which implies that every Validator proposes a block in every Narwhal round and that every leader block of even Narwhal rounds include references (a.k.a. strong edges) to *all* blocks from the previous (odd) round.

Assume that a User submits an Initiate MPC Transaction tx, after the Process phase has completed, and it is proposed in Narwhal round r by several primaries. If r is even, then tx is committed by Narwhal round $r+2$ *if and only if* it was included in the leader's block at round r. If it was not included in Narwhal round r's leader's block, then tx can be committed at the earliest in round $r+4$. If r is odd, then tx can be committed at the earliest when Narwhal round $r+1$'s leader's block is committed, which is in Narwhal round $r+3$. So, on average, the latency for consensus to deliver a User's transaction is 3.5 Narwhal rounds.

We assume now that every Validator is able to process inputs and produce the next Narwhal message (an MPC Message or an MPC Output) quickly enough to be proposed in the next Narwhal round.[5]

Let r be the round in which the User's tx was *proposed*.

[5] This is a strong assumption since proposing a block in Narwhal requires gathering q acknowledgements from other Narwhal primaries on each batch that is proposed in the block.

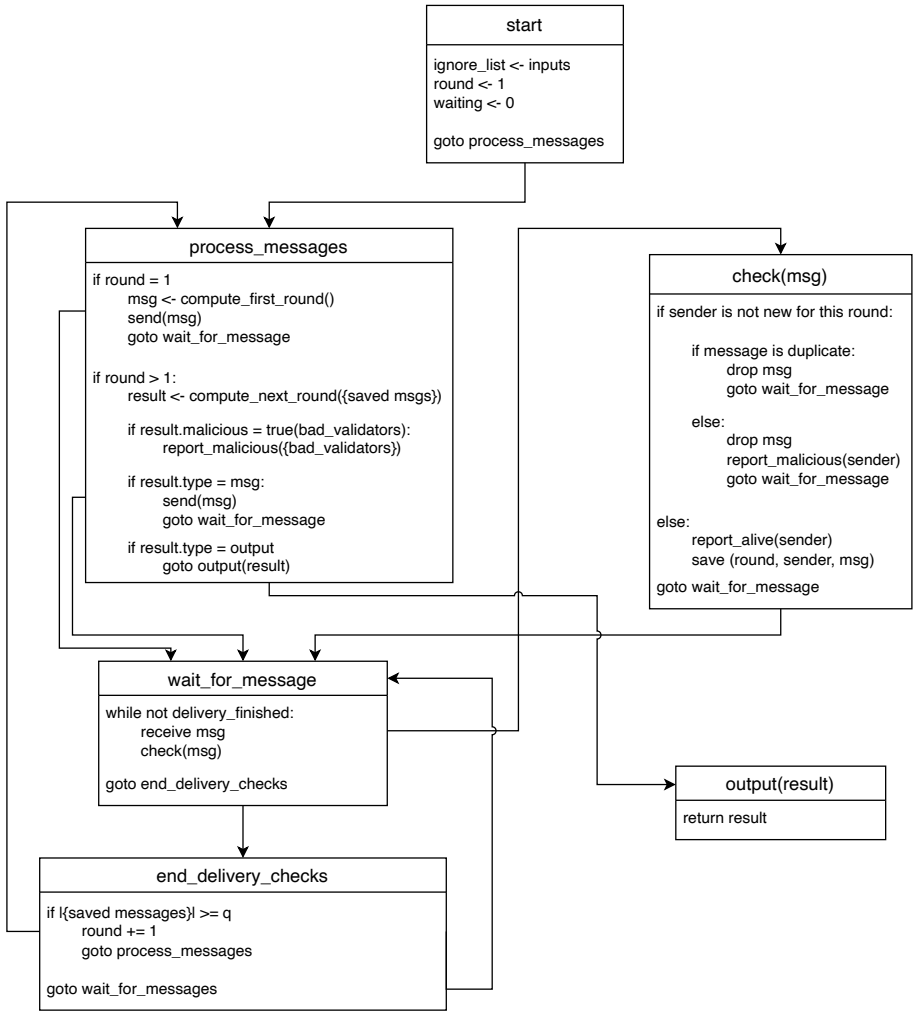

Figure 2: MPC Instance state machine

MPC Protocol	User Txs	MPC Rounds	Narwhal Rounds	Latency
KeyGen	2	2	15	7.5 s
PreSign (ECDSA)	1	3	15.5	7.35 s
PreSign (Schnorr)	1	2	11.5	5.75 s
Sign	1	2	11.5	5.75 s

Table 1: Estimated average latency of the asynchronous MPC Protocols in terms of Narwhal Rounds and seconds, assuming 0.5 s per Narwhal round. The number of MPC rounds includes one broadcast round for the output of the protocol after it has completed.

1. If r is even and tx was committed by Narwhal round $r+2$, then every Validator sends their first round MPC message in their own blocks proposed at Narwhal round $r+3$.
   In the best case scenario (which is unlikely), the leader block of Narwhal round $r+4$ has a strong edge to every block from Narwhal round $r+3$ and therefore commits all of these messages when it is committed by Narwhal round $r+6$. (So the messages are proposed within Narwhal round $r+3$ and delivered 3 rounds later during Narwhal round $r+6$.)
   If tx was only committed by Narwhal round $r+4$, then the same analysis applies with a 2-round delay and the MPC messages are committed by Narwhal round $r+8$.
2. If r is odd, then every Validator sees tx during Narwhal round $r+3$ (even) and sends their first round MPC message in their own blocks proposed at Narwhal round $r+4$ (odd).
   In the best case scenario (which is unlikely), the leader block of Narwhal round $r+5$ (even) has a strong edge to every block from Narwhal round $r+4$ and commits all of the first round MPC messages when it is committed by Narwhal round $r+7$. (So the MPC messages are proposed in Narwhal round $r+4$ and delivered 3 rounds later during Narwhal round $r+7$.)

The same analysis applies for subsequent rounds of MPC Messages: every broadcast round is proposed during an odd Narwhal round and delivered by the consensus protocol 3 Narwhal rounds later. Since one MPC round's messages cannot be proposed in the same Narwhal round that delivered the previous MPC round's messages, each MPC round of broadcast requires 4 Narwhal round to complete.

In conclusion, the average best-case latency of a k-round MPC protocol that requires l User Transactions using this system is

$$3.5 \cdot l + 4 \cdot k \text{ Narwhal rounds.} \tag{1}$$

6 MPC Rate Limiter

To concentrate the work of honest Validators and ensure that all MPC Instances are executed in a reasonable time, the dWallet system contains a Rate Limiter which functions according to the rules below. This Rate Limiter for the system is parameterized by the value $M_{\texttt{running}} \in \mathbb{N}^+$ which determines the maximum number of concurrent MPC instances executed by the system.

1. Each MPC Instance that has not aborted or timed out is in one of two states: `running` or `pending`.

2. There can be at most $M_{\texttt{running}}$ MPC Instances in the `running` state at any one time.

3. A new MPC Instance spawned from a User Transaction should be spawned in the `running` state *only if* there are fewer than $M_{\texttt{running}}$ MPC instances at the moment; otherwise, it must be spawned in the `pending` state and added to the queue of pending instances.

   To maintain ordering and fairness, new MPC Instances must be spawned in the order given to them by the consensus protocol.

   Optimisation 4. *To improve the latency of Sign Transactions in case of network congestion, it could be possible to make a rule that new Sign Transactions that need to be placed in the pending queue should be placed at the front of the queue, so that they can be processed first.*

4. An MPC Instance in the `running` state must be run to its conclusion before its state can be changed, which will either be an abort, a time out or an output.

   (a) If the conclusion is an output, then the MPC Instance is over and removed from the list of running instances; the MPC Instance at the front of the pending queue must then be set to `running`.

   (b) If the conclusion is an abort or a time out, then the MPC Instance spawned for the next attempt at the MPC Protocol must be spawned at the front of the pending queue so that it is the next one to be set to `running` when a spot opens up.

By giving MPC Instances their `running` or `pending` state in the order given by the consensus protocol, and by keeping count of the current number of running MPC Instances, every Validator should locally have the same set of running instances and queue of pending instances.

7 Synchronisation

This section describes the process that a Validator must follow in order to synchronise their state. This is required for them to be able to re-join the system after a crash for example.

1. Use the mechanism already implemented in Sui (inherited from the consensus protocol) to recover the DAG blocks and consensus deliveries that were missed.

2. Starting from the oldest delivery that was not processed, process the recovered deliveries to update the Object Store, MPC Instance Handler and MPC Reputation Handler with the following information:

   (a) New Transactions (MPC or not) that created, mutated or deleted Objects in the System or initiated MPC Protocols which should be processed to the MPC Instance Handler according to the processing rules and the MPC Rate Limiter.

   (b) MPC Messages that were sent for MPC Instances should be verified for the identification of malicious Validators. If such a Validator is identified, the

MPC Reputation Handler should be updated before processing the next recovered consensus delivery.

(c) MPC Output Transactions should be checked as during normal functioning before their effects are applied to the Object Store.

3. After the recovered consensus deliveries have been processed in sequence until they are back in sync with the current Narwhal round, the Validator can begin proposing Narwhal blocks and MPC Messages again.

Optimisation 5. *The rule which specifies that MPC Messages from malicious Validators should be ignored in MPC Instances places a computational burden on honest Validators that are synchronising. Namely, it requires that they verify the validity of MPC Messages for all MPC Instances that they missed in order to identify any malicious behaviour and appropriately ignore malicious Validators like the other honest Validators.*

To remove this computational burden, the rule could be changed to no longer ignore valid MPC Messages sent by Validators that have previously behaved maliciously. However, any honest Validator that observed the malicious behaviour would consider the offending Validator as malicious from that point onwards for the purpose of performance measurement and reward distribution.

Enabling this optimisation essentially tolerates malicious Validators as long as they resume honest behaviour. This is not standard practice in security analysis and would need to be studied further, and so would the impact on the reward distribution of only certain honest Validators knowing about the malicious behaviour of another. This optimisation is therefore presented here with no guarantee of security analysis.

8 Tokenomics

This section discusses the Sui Economic Model and analyses which desired properies for the dWallet network can be realized from it.

8.1 Sui Economic Model

Sui works with a Delegated Proof-of-Stake protocol, where a committee of validators is elected to maintain the system for a fixed period of time, called an *epoch*.

During epoch e, the total stake in the system is:

$$TS_e = \mathcal{F}_e + \mathcal{S}_e = \mathcal{F}_e + \sum_{v \in \mathcal{V}} S_e(v) \tag{2}$$

where $\mathcal{S}_e$ is the total stake delegated during epoch e, and $\mathcal{F}_e$ is the balance of Sui's *storage fund* at the start of the epoch. Every epoch, part of the storage fund is made available to incentivize validators to store the data necessary for maintaining the health of Sui and ensure that present and future validators get additional rewards for storing this data. Given TS_e, we can calculate the proportion of delegated stake in epoch e to be $\alpha_e = \mathcal{S}_e / TS_e$.

The validators and delegators in Sui are rewarded through *gas prices*. The gas prices combine two separate prices, i.e., (i) execution price and (ii) storage price.

These prices are calculated as follows:

- Execution price. The execution price is determined in a three-step process that takes place at the start of the epoch:
  - Gas Price Survey: First, all validators submit the gas price that they deem acceptable ($\bar{p}_e^C$). These gas prices are used to determine a *reference gas price* ($\bar{P}_e^C$), i.e., the price for which the validators are expected to respond promptly, thus enabling fast executions of transactions.
  - Second, the system sets a *price floor* p', i.e., the minimum gas price.
  - The execution gas price for a transaction τ can then be computed as follows:

$$P_e^c[\tau] = \bar{P}_e^C + \zeta[\tau] \tag{3}$$

    where $\zeta[\tau]$ is the transaction tip. Hence, execution prices for each transaction can differ.
- Storage price: At the start of an epoch, the system determines the storage price; validators cannot influence this. All transactions submitted during the epoch use the same base storage price. This price may change between epochs.

The rewards received for transactions executed during an epoch are initially pooled. At the end of the epoch, these rewards are split among the validators and their delegators based on their performance. Next, we look into the exact mechanism used to perform this split.

8.1.1 Sui Reward Distribution

At the end of an epoch, the system calculates the rewards to be distributed. The total rewards accumulated throughout the epoch are calculated to do this. Let the total stake rewards be $StakeRewards_e$. For each epoch, the delegators also agree on a commission for validators in the current committee; let this commission be $\delta \in [0, 1]$. Additionally, let the portion of storage fund rewards accumulated within the current epoch given to validators be $\gamma \in [0, 1]$.

The rewards for the set of delegators and validators, in addition to the reinvestment in the storage fund, are then distributed as follows:

$$DelegatorsReward = (1 - \delta) \cdot \alpha_e \cdot StakeRewards_e \tag{4}$$

$$ValidatorsReward = [\delta \cdot \alpha_e + \gamma \cdot (1 - \alpha_e)] \cdot StakeRewards_e \tag{5}$$

$$StorageFundReinvestment = (1 - \gamma) \cdot (1 - \alpha_e) \cdot StakeRewards_e \tag{6}$$

Validator Reward Distribution Next, Sui assigns individual rewards for each validator in the committee. Sui does this in the following steps:

1. Gas Price Survey: As previously discussed, the gas price survey takes as inputs "ideal gas prices" from each validator in the committee and fixes a *reference gas price* for the current epoch.
2. Tallying Rule: Each validator in the committee submits a subjective opinion about the performance of every other validator in the system. The median of these opinions is chosen as the performance of a validator. To calculate this opinion, the following steps take place:

- For the set of transactions T_e executed in epoch e, where each transaction $\tau \in T_e$ includes a computation gas price $P_e^c[\tau]$, the executed gas price distribution is:

$$T_e[p] = \tau \in T_e, \text{ s.t. } P_e^c[\tau] \geq p \tag{7}$$

- Each validator v also calculates the *Reasonable Execution Metric* ($\hat{T}_e^v(v')$) for every other validator $v' \neq v$. This metric measures how many transactions matching v''s bid were promptly executed by them.

$$\hat{T}_e^v(v^{'}) = \tau \in T_e[\bar{p}_e^c(v^{'})] \tag{8}$$

  such that τ was executed promptly.

- Based on the executed gas price distribution and the reasonable execution metric, each validator then calculates the subjective performance of every other validator that is represented by a multiplier $\hat{\mu}_e^v(v^{'})$.

$$\hat{\mu}_e^v(v^{'}) = \phi^v \cdot \frac{\sum_{\tau \in \hat{T}_e^v(v')} \text{CompUnit}(\tau) \cdot P_e^c[\tau]}{\sum_{\tau \in T_e[\bar{p}_e^c(v')]} \text{CompUnit}(\tau) \cdot P_e^c[\tau]'} \tag{9}$$

  where ϕ^v is a normalizing multiplier, i.e., all μ's submitted by v for all other validators average out to 1 but in which validators with good performance get a boost ($\mu > 1$) and validators with a relatively bad performance get a discount ($\mu < 1$).

Using this multiplier, Sui can then calculate individual validator rewards from the $ValidatorsReward$ calculated in equation 4 as follows:

$$\hat{\sigma}_e(v) = \begin{cases} \psi \cdot (1+\kappa) \cdot \hat{\mu}_e^v(v) \cdot \sigma_e(v), & \text{if } \bar{p}_e^c(v) \leq P_e^c \\ \psi \cdot (1-\kappa) \cdot \hat{\mu}_e^v(v) \cdot \sigma_e(v), & \text{if } \bar{p}_e^c(v) > P_e^c \end{cases} \tag{10}$$

where ψ is a normalizing multiplier such that $\sum_{v \in \mathcal{V}_e} \hat{\sigma_e}(v) = 1$, $\bar{p}_e^c(v)$ is the bid submitted by validator v during the gas price survey, and $\bar{P}_e^c$ is the reference gas price.

8.2 dWallet Tokenomics based on Sui

The Sui economic model is very elegant and enables a fair distribution of rewards: responsive validators get boosted rewards, and lazy validators get reduced rewards. This distribution has a two-fold effect. Firstly, validators cannot be rewarded if they do not perform at all; their effective multipliers would be close to 0. This is because their total transactions are lower than they should have executed (see equation 9); thus, the median multiplier for these validators would be close to 0. Secondly, delegators might remove their stake from underperforming validators since they are not rewarded.

The MPC validators in dWallet should be rewarded based on their responsiveness. In addition, there should be bonus rewards for validators participating in longer subprotocols.

We conjecture that equation 9 covers these requirements. More specifically, the CompUnit(τ) captures the total computation units a transaction incurred, which explains how complicated/long a transaction was.

Assuming that the base CompUnit in dWallet would be the computation required for *Sign* subprotocol, all MPC validators that participate in MPC sub-protocols such as KeyGen or PreSign would have their aggregated CompUnits much higher than a validator who only engages in Sign. Thus, equation 9 would give them a higher performance multiplier ($\hat{\mu}_e^v$) as compared to lazy validators who either are not responsive or do not participate in longer subprotocols.

As for the *Malicious* validators, each honest validator could just set their performance multiplier to 0. Assuming that at least Q validators are honest and set $\hat{\mu}_e^v(v^{'}) = 0$, the median for malicious validators would also be 0; thus, not entitling them to any rewards.

The rewards that a malicious validator miss out on are proportionally distributed to honest validators.

8.2.1 What is missing?

Sui does not have any built-in slashing for malicious validators. This is due to the delegated PoS nature, where the delegators remove all stakes from a malicious entity (as a form of punishment).

If the dWallet network requires slashing for malicious validators, this is not covered in the current Sui economic model.

This incentive mechanism also does not cover rewards for *wasted effort*. For instance, a PreSign protocol that executes for 5 rounds but fails to finish due to a dropped validator does not reward alive and honest validators that carried out this protocol; we conjecture that this might have a negative implication for validators, i.e., they might not want to join longer subprotocols due to their associated risks.

References

[BCD+23] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path, 2023.

[DKKSS22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[SGSK22] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022.