Sui Lutris: A Blockchain Combining Broadcast and Consensus

Sam Blackshear*, Andrey Chursin*, George Danezis*†, Anastasios Kichidis*, Lefteris Kokoris-Kogias*‡, Xun Li*, Mark Logan*, Ashok Menon*, Todd Nowacki*, Alberto Sonnino*†, Brandon William*, Lu Zhang*
*Mysten Labs, †University College London, ‡IST Austria

***Abstract*—Sui Lutris is the first smart-contract platform to sustainably achieve sub-second finality. It achieves this significant decrease in latency by employing consensusless agreement not only for simple payments but for a large variety of transactions. Unlike prior work, Sui Lutris neither compromises expressiveness nor throughput and can run perpetually without restarts. Sui Lutris achieves this by safely integrating consensusless agreement with a high-throughput consensus protocol that is invoked out of the critical finality path but makes sure that when a transaction is at risk of inconsistent concurrent accesses its settlement is delayed until the total ordering is resolved. Building such a hybrid architecture is especially delicate during reconfiguration events, where the system needs to preserve the safety of the consensusless path without compromising the long-term liveness of potentially misconfigured clients. We thus develop a novel reconfiguration protocol, the first to show the safe and efficient reconfiguration of a consensusless blockchain. Sui Lutris is currently running in production as part of a major smart-contract platform. Combined with the Move Programming language it enables the safe execution of smart contracts that expose objects as a first-class resource. In our experiments Sui Lutris achieves latency lower than 0.5 seconds for throughput up to 5,000 certificates per second, compared to the state-of-the-art real-world consensus latencies of 3 seconds. Furthermore, it can gracefully handle crashes and recovery of validators and does not suffer observable performance degradation during reconfiguration.**

1. Introduction

Traditional blockchains totally order transactions across replicated miners or validators to mitigate "double-spending" attacks, i.e., a user trying to use the same coin in two different transactions. It is well known that total ordering requires consensus. In recent years, however, systems based on consistent [1] and reliable [2] broadcasts have been proposed instead. These rely on objects (e.g., a coin) being controlled by a single authorization path (e.g., a single signer or a multi-sig mechanism), responsible for the liveness of transactions. This concept has been used to design asynchronous, and lightweight alternatives to traditional blockchains for decentralized payments [1], [3], [4]. We call these systems *consensusless* as they do not require full consensus. Yet, so far they have not been used in a production blockchain.

There are multiple reasons for this. First, consensusless protocols typically support a restricted set of operations limited to asset transfers. Second, deploying consesusless protocols in a dynamic environment is challenging as they do not readily support state checkpoints and validator reconfiguration. Supporting these functions is vital for the health of a long-lived production system. Finally, consensusless protocols are not tolerant of client bugs as any equivocation locks the assets forever.

As a result all existing blockchains implement consensus-based protocols that allow for general-purpose smart contracts. Sadly, this flexibility comes at the cost of higher complexity and significantly higher latency even for transactions that operate on unrelated parts of the state.

In this paper, we present Sui Lutris, the first system that combines the consensusless and consensus-based approaches to provide the best of both worlds when processing transactions in a replicated Byzantine setting. Sui Lutris uses a consistent broadcast protocol between validators to ensure the safety of all operations, ensuring lower latency as compared to consensus. It only relies on consensus for the correct execution of complex smart contracts operating on shared-ownership objects, as well as to support network maintenance operations such as defining checkpoints and reconfiguration. It is maintained by a permissionless set of validators that play a similar role to miners in Bitcoin.

Challenges. Sui Lutris requires tackling 3 key issues: Firstly, a high-throughput system such as Sui Lutris requires a checkpoint protocol in order to archive parts of its history and reduce the memory footprint and bootstrap cost of new participants. Checkpointing however is not as simple as in classic blockchains since we do not have total ordering guarantees for all transactions. Instead, Sui Lutris proposes an after-the-fact checkpointing protocol that eventually generates a canonical sequence of transactions and certificates, without delaying execution and transaction finality.

Secondly, consensusless protocols typically provide low latency at the cost of usability. A misconfigured client (e.g., underestimating the gas fee or crash-recovering) risks deadlocking its account. We consider this an unacceptable compromise for production systems. We develop Sui Lutris such that client bugs only affect the liveness of a single epoch, and provide rigorous proofs to support it. The current epoch length for our production system is 24h, but we ran experiments with an epoch length of 10 minutes without seeing any effect on the performance.

Finally, the last challenge to solve is the dynamic participation of validators in a permissionless system. The lack of total ordering makes the solution non-trivial as different validators may stop processing transactions at different points compromising the liveness of the system. Additional challenges stem from the non-starvation needs of misconfigured clients coupled with ensuring that final transactions are never reverted across reconfiguration events. To this end, we design a custom reconfiguration protocol that preserves safety with minimal disruption of the processing pipeline.

Real-world system. SUI LUTRIS has been designed for and adopted as the core system behind a major new blockchain, that uses the Move programming language [5] for smart contracts. As of July 30, 2023, its mainnet is operated by 105 geo-distributed heterogeneous validators and processes over 2.5 million certificates a day over 100 epoch changes using the SUI LUTRIS protocols. It stores over 80 million objects, owned by over 2.6 million addresses, defined by over 6,000 Move packages. On the peak throughput day (Jul. 27, 2023) it processed over 65 million certificates, the highest volume of any blockchain, and higher than the total number of transactions on all chains combined, on that day. For this reason, we present in the paper details that go beyond merely illustrating core components. SUI LUTRIS achieves finality within 0.5 seconds with a committee of 10 validators processing up to 5,000 cert/s or with a committee of 100 validators processing about 4,000 cert/s. SUI LUTRIS can provably withstand up to $1/3$ of crash validators without meaningful performance degradation and can seamlessly reconfigure despite the complexities of supporting a consesusless transaction processing path. We evaluated SUI LUTRIS against Bullshark [6], a state-of-the-art consensus protocol, operating within the same blockchain and show that SUI LUTRIS achieves finality up to 15x earlier than Bullshark.

Contributions. We make the following contributions:

- We present SUI LUTRIS, the first smart-contract system that forgoes consensus for single-writer operations and only relies on consensus for settling multi-writer operations, combining the two modes securely and efficiently.
- As part of SUI LUTRIS we show how to use a consensus engine to efficiently checkpoint a consensusless blockchain without forfeiting the latency benefits of running consensusless transactions. Our checkpointing mechanism puts transactions into a sequence after execution, reducing the need for agreement and therefore latency.
- Finally, we show how to perform reconfiguration safely and with minimal downtime. Unlike prior consensusless blockchains our reconfiguration mechanism allows for forgiving equivocation so that careless users can regain access to their assets.
- We provide full implementation and evaluate the performance of SUI LUTRIS, on a real geo-distributed set of validators and under varying transaction loads.

2. Overview

SUI LUTRIS uses a novel approach to processing blockchain transactions which ensures low latency by forgoing the need for consensus from the critical latency path. Yet, skipping consensus provides finality (knowing that a transaction will execute) and settlement (knowing the exact execution result) only for single-owner assets (assets that are immutable or owned directly or indirectly by a single address, see below).

Despite single-owner transactions constituting most of the load on our mainnet, these types of assets are not expressive enough to implement all types of smart contracts – since some transactions must process assets belonging to different parties. For the rest of the transactions, we can only get finality but need to postpone settlement until potential conflicts are resolved. For this reason, we couple SUI LUTRIS with a consensus protocol. This hybrid architecture is both a curse and a blessing: two different execution paths create the threat of inconsistencies and safety concerns; but, the consensus component enables us to implement checkpointing and reconfiguration, which consensus-less systems lack.

We define the problem SUI LUTRIS addresses, the threat model , the security properties maintained, and provide a high-level overview of the algorithm.

2.1. Threat Model

We assume a message-passing system with a set of n validators per epoch and a computationally bound adversary that controls the network and can statically corrupt up to $f < n/3$ validators within any epoch. We say that validators corrupted by the adversary are *Byzantine* or *faulty* and the rest are *honest* or *correct*. To capture real-world networks we assume asynchronous *eventually reliable* communication links among honest validators. That is, there is no bound on message delays and there is a finite but unknown number of messages that can be lost. Informally, SUI LUTRIS exposes to all participants a *key-value* object store abstraction that can be used to read and write objects.

Consensus Protocol. SUI LUTRIS uses a consensus protocol as a black box that takes some valid inputs and outputs a total ordering. It makes no additional synchrony assumptions and thus inherits the synchrony assumptions of the underlying consensus protocol. In our implementation, we specifically use the Bullshark protocol [6]. It is secure in the partially synchronous network model [7], which stipulates that after some unknown global stabilization time all messages are delivered within a bounded delay. It could be configured to run the Tusk protocol [8], making SUI LUTRIS asynchronous.

2.2. System Model

Objects. SUI LUTRIS validators replicate the state represented as a set of objects. Each object has a type associated that defines operations that are valid state transitions for the type. Each object may be read-only, owned, or shared:

- *Read-only objects* cannot be mutated or deleted within an epoch and can be used in transactions concurrently and by all users.
- *Owned objects* have an owner field. The owner can be set to an address representing a public key. In that case, a transaction is authorized to use the object, and mutate it, if it is signed by that address. A transaction is signed by a single address and therefore can use objects owned by that address. However, a single transaction cannot use objects owned by multiple addresses. The owner of an object (called a child object) can also be another object (called the parent object). In that case, the child object may only be used if the root object (the first one in a tree of possibly many parents) is part of the transaction, and the transaction is authorized to use the parent. This facility is used by contracts to construct dynamic collections and other complex data structures.
- *Shared objects* are mutable and do not have a specific owner. They can instead be included in transactions by anyone and they perform their own authorization logic as part of the smart contract. Such objects, by virtue of having to support multiple writers while ensuring safety and liveness, require a full agreement protocol to be executed safely.

Both owned and shared objects are associated with a version number. The tuple $(\mathsf{ObjID}, \mathsf{Version})$ is called an ObjKey and takes a single value, thus it can be seen as the equivalent of a Bitcoin UTXO [9] that should not be equivocated.

Transactions. A transaction is a signed command specifying the number of input objects (read-only, owned or shared), a version number per object, an entry function into a smart contract, and a set of parameters. If valid, it consumes the mutable input objects and constructs a set of output objects at a fresh version – which can be the same objects at a later version or new objects.

2.3. Core Properties

SUI LUTRIS achieves the standard security properties of blockchain systems relating to validity, safety, and liveness:

- **Validity**: State transitions at correct validators are in accordance with the authorization rules relating to objects, as well as the VM logic constraining valid state transitions on objects of defined types. This property is unconditional with respect to the number of correct validators in the network.
- **Safety**: If two transactions t and t' are executed on correct validators, in the same or different epochs, and take as input the same ObjKey, then $t = t'$. This property holds in full asynchrony subject to a maximum threshold of Byzantine nodes in the system.
- **Liveness**: All valid transactions sent by correct clients are eventually processed up to final status, and their effects persist across epoch boundaries. All ObjKey that have not been used as inputs to a committed transaction are eventually available to be used by a correct client as part of a valid transaction. This property holds under partial synchrony, due to our use of Bullshark [6] consensus (but would hold in asynchrony when using Tusk [8]).

Liveness encompasses censorship resistance. It also only holds for a correct client that may not equivocate by sending conflicting transactions for the same owned object version.

2.4. Core Protocol Overview

Figure 1 illustrates the high-level interactions between a client and SUI LUTRIS validators to commit a transaction. A user with a private key creates and signs a *user transaction* to either mutate objects they own (case 1) or a mix of objects they own (at least one is gas) and shared objects (case 2). This is the only time user signature keys are needed; the remaining process may be performed by the user or a gateway on behalf of the user (❶). The transaction is sent to each SUI LUTRIS validator, which checks it for validity, locks all input owned objects using their ObjKey, signs it, and returns the *signed transaction* to the client (❷). Algorithm 1 of Section 3.4 describes in details this step. It is critical to see that since an ObjKey is locked for a specific transaction, no double-spend can happen. As we discuss later, however, there is a chance of liveness loss if the client equivocates. This check is also used to prevent attacking consensus with spam transactions which is easy to do in Bullshark [6] as it orders bytes which could be duplicate transactions.

The client collects the responses from a quorum $(2f+1)$ of validators to form a *transaction certificate* (❸). As a result, unlike consensus-based blockchains, in SUI LUTRIS, the validators need not gossip signatures or aggregate certificates, which is now the responsibility of the client/gateway. Once the certificate is assembled, it is sent back to all validators, who respond once they check its validity (a quorum of responses at this point ensures *transaction finality*, see below). If the transaction involves exclusively read-only and owned objects the transaction certificate can be immediately processed and settled (shown as the *direct fast path* in the diagram).

All certificates are forwarded to a *Byzantine agreement protocol* operated by the SUI LUTRIS validators (❹). When step (❹) terminates consensus outputs a total order of certificates, and validators check the certificate and settle those containing shared objects. The execution result is a summary of how the transaction affects the state and is used to construct a *signed effects* response (❺). Algorithm 4 of Section 3.4 describes in details this step. Once a quorum of validators has executed the certificate its effects are final (in the sense of *settlement*, see below). Clients can collect a quorum of validator responses and create an *effects certificate* and use it as proof of the finality of the transactions effects (❻). Subsequently, checkpoints are formed for every consensus commit (❼), which are also used to drive the reconfiguration protocols (not shown, see Section 4.1 and Section 4.3).

The goal of the above protocol is to ensure that execution for transactions involving read-only and owned objects requires only reliable broadcast and a single certificate to proceed, costing a minimal $O(n)$ communication and com-

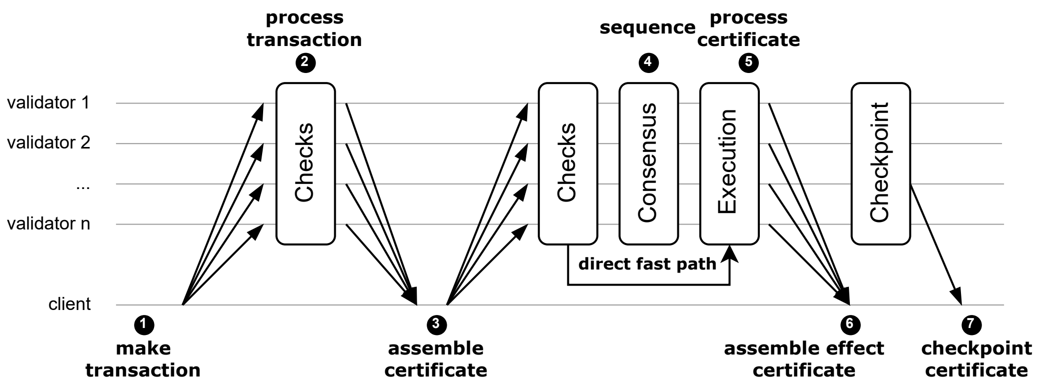

Figure 1. Overview and transaction life cycle. The Byzantine agreement protocol is only executed for transactions containing shared objects and is not necessary for transactions involving only single-owner objects.

putation complexity and *no validator to validator communication*. Smart contract developers can therefore design their types and their operations to optimize transfers and other operations on objects of a single user to reduce the cost of their transactions. SUI LUTRIS also provides the flexibility of using shared objects, through the classic Byzantine agreement path, and enables developers to implement logic that needs to be accessed by multiple users.

Transaction finality. The BIS [10] defines finality as the property of being "irrevocable and unconditional". We make a distinction between transaction finality, after which a transaction processing is final, and settlement, after which the effects of a transaction are final and may be used in subsequent transactions in the system. In SUI LUTRIS, unlike other blockchains, both finality and settlement occur before checkpoints are formed. In all cases, *a transaction becomes final when a quorum of validators accepts the transaction's certificate for processing, even before such a certificate is sequenced by consensus or executed.*. After this, no conflicting transaction can occur, the transaction may not be revoked, and is eventually executed and persist across epochs. However, the result of execution is only known a-priori for owned object transactions, and for shared object transactions it is known only after consensus. Transaction finality is achieved within two network round trips.

Settlement occurs upon execution on $2f + 1$ validators, when an effects certificate could be formed (even though it is not known by any single party). For owned object transactions this occurs upon $2f + 1$ correct validators executing the certificate without the delay of consensus; for shared object transactions this happens upon $2f + 1$ correct validators executing them just after the certificate has been sequenced. In both cases, settlement is not delayed by the process of commiting the transaction within a checkpoint, and thus, has lower latency than checkpoint creation.

3. The SUI LUTRIS System

We present the SUI LUTRIS core protocol by providing algorithms and specifying the checks performed by validators at each step of the protocol.

3.1. Objects Operations

SUI LUTRIS validators rely on a set of objects to represent the current and historical state of the replicated system. An *Object* (Obj) stores user smart contracts and data within SUI LUTRIS. Transactions can affect objects which can be $\mathsf{Created}$, $\mathsf{Mutated}$, $\mathsf{Wrapped}$, $\mathsf{Unwrapped}$ and $\mathsf{Deleted}$.

Calling *key*(Obj) returns the *key* (ObjKey) of the object, namely a tuple (ObjID, $\mathsf{Version}$). $\mathsf{ObjID}$is cryptographically derived so that finding collisions is infeasible. Versions monotonically increase with each transaction processing the object and are determined via Lamport timestamps [11]. Calls to *version*(Obj) and *initial*(Obj) return the current and initial version of the object respectively.

The *owner*(Obj) is either the owner's public key that may use this object, or the ObjID of another *parent* object, in which case this is a *child* object. *owned*(Obj) returns whether the object has an owner, or whether it is read-only or a shared object (see below). The last transaction digest ($\mathsf{TxDigest}$) that last mutated or created the object *creator*(Obj).

3.2. Protocol Messages

Validators and users run the core protocol described in Section 2.4 by exchanging the following messages.

Transactions. A *transaction* (Tx) is a structure representing a state transition for one or more objects. They support a few self-explanatory access operations, such as to get its digest *digest*(Tx), and different types of input objects *inputs*(Tx) (object reference), *read_only_inputs*(Tx), *shared_inputs*(Tx) (object ID and initial version), and *payment*(Tx) (the reference to the gas object to pay fees).

A Transaction may be checked for validity given a set of input objects, or can be executed to compute output objects:

- *valid*($\mathsf{Tx}, [\mathsf{Obj}]$) returns true if the transaction is valid, given the requested input objects provided. This check verifies that the transactions are authorized to act on the input objects, as well as sufficient gas is available to cover the costs of its execution. Transaction validity, as returned by *valid*($\mathsf{Tx}, [\mathsf{Obj}]$) can be determined statically without executing the Move contract.

- *exec*($\mathsf{Tx}, [\mathsf{Obj}_o], [\mathsf{Obj}_s]$) executes using the MoveVM [5] and returns a structure Effects representing its effects along with the output objects $[\mathsf{Obj}_{out}]$. The output objects are the new objects Created, Mutated, Wrapped, Unwrapped and Deleted by the transaction. When objects are created, updated, or unwrapped their version number is the Lamport timestamp of the transaction. $[\mathsf{Obj}_o]$ and $[\mathsf{Obj}_s]$ respectively represent the owned and shared input objects. A valid transaction execution is infallible and has deterministic output.

A transaction is indexed by the TxDigest over the raw data of the transaction , which may also be used to authenticate its full contents. All valid transactionshave at least one owned input, namely the objects used to pay for gas.

Certificates. A *transaction certificate* (TxCert) on a transaction contains the transaction itself, the signature of the user authorizing the use of the input objects, and the identifiers and signatures from a quorum of $2f + 1$ validators or more. For simplicity, we assume that every operation defined over a transaction is also defined over a certificate. For instance, "*digest*(Tx)" is equivalent to "*digest*(TxCert)". A certificate may not be unique, and the same logical certificate may be signed by a different quorum of validators or even have different authorization paths (e.g., a 2-out-of-3 multisig). However, two different valid certificates on the same transaction should be treated as representing semantically the same certificate. The identifiers of signers are included in the certificate to identify validators ready to process the certificate, or that may be used to download past information required to process the certificate. Additionally, the signatures are aggregated (eg. using BLS [12]), compressing the quorum of signers to a single signature.

Transaction effects. A *transaction effects* (Effects) structure summarizes the outcome of a transaction execution. Its digest and authenticator is computed by *digest*(Effects). It supports operations to access its data such as *transaction*(Effects) (returns the transaction digest) and *dependencies*(Effects) (the digest of all transactions to create input objects). The *contents*(Effects) returns a summary of the execution: Status reports the outcome of the smart contract execution. The lists Created, Mutated, Wrapped, Unwrapped, and Deleted, list the object references that underwent the respective operations. Finally, Events lists the events emitted by the execution.

Partial certificate. A *partial certificate* (TxSign) contains the same information, but signatures from a set of validators representing stake lower than the required quorum, usually a single one. We call *signed transaction* a partial certificate signed by one validator.

Effect certificates. Similarly, an *effects certificate* (EffCert) on an effects structure contains the effects structure itself, and signatures from validators[1] that represent a quorum for the epoch in which the transaction is valid. The same caveats, about non-uniqueness and identity apply as for transaction certificates. A partial effects certificate, usually containing a single validator signature and the effects structure is denoted as EffSign. For transactions that only include owned objects, this certificate provides both finality and settlement. For transactions with shared objects validators can first reply with an empty effects structure (so when an empty effects certificate is formed the system reaches finality) and then add the effects structure and re-sign after consensus (to reach settlement). For the rest of the paper, we omit the transmission of the empty effects structure for simplicity.

3.3. Data Structures

Each validator maintains a set of persistent tables abstracted as key-value maps, with the usual contains, get, and set operations.

Reliable broadcast on owned objects uses the *owned lock map* ($\mathsf{OwnedLock}[\mathsf{ObjKey}] \to \mathsf{TxSignOption}$) which records the first valid transaction Tx seen and signed by the validator for an owned object's ObjKey, or None if the ObjKey exists but no valid transaction using it as an input has been seen. The *certificate map* ($\mathsf{Ct}[\mathsf{TxDigest}] \to (\mathsf{TxCert}, \mathsf{EffSign})$) records all full certificates TxCert, including Tx, processed by the validator, along with their signed effects EffSign.

To manage the execution of shared object transactions, the *shared lock map* ($\mathsf{SharedLock}[(\mathsf{TxDigest}, \mathsf{ObjID})] \to \mathsf{Version}$) records the version number of ObjID assigned to a transaction TxDigest. The *next shared lock map* ($\mathsf{NextSharedLock}[\mathsf{ObjID}] \to \mathsf{Version}$) records the next available version (we discuss their use in Section 3.4).

The *object map* ($\mathsf{ObjDB}[\mathsf{ObjKey}] \to \mathsf{Obj}$) records all objects Obj created by executed certificates within Ct by object key, and also allows lookups for the latest known object version. This store can be completely derived by re-executing all certificates in Ct. Only the latest version is necessary to process new transactions and older versions are only maintained to facilitate parallel execution, reconfiguration, reads, and audits.

Only ($\mathsf{OwnedLock}, \mathsf{SharedLock}, \mathsf{NextSharedLock}$) require strong key self-consistency, namely a read on a key should always return whether a value or None is present for a key that exists, and such a check should be atomic with an update that sets a lock to a non-None value. This is a weaker property than strong consistency across keys and allows for efficient sharding of the store for scaling. To ensure this property, they are only updated by a single task (see Section 3.4). The other stores may be eventually consistent without affecting safety. Section 4.2 shows how to initialize and securely reset stores upon epoch changes.

1. Note that if the signature algorithm permits it, validator signatures can be compressed, but always using signature aggregation because tracking who signed is important for gas profit distribution and other network health measurements.

3.4. Validator Core Operation

Validators process transactions and certificates as described in Section 2.4. Transactions are submitted to the validator core for processing by users, while certificates can either be submitted by users or by the consensus engine.

Algorithm 1 Process transaction

```
    // Executed upon receiving a transaction from a user.
    // Many tasks can call this function.
 1: procedure ProcessTx(Tx)
 2:     // Check 1.1: Ensure all objects exist.
 3:     $[\mathsf{Obj}_o] = \textsc{LoadObjects}(\mathit{inputs}(\mathsf{Tx}))$
 4:     $[\mathsf{Obj}_r] = \textsc{LoadLatestVersionObjects}(\mathit{read\_only\_inputs}(\mathsf{Tx}))$

 5:     for $(\mathsf{ObjID}, \mathsf{InitialVersion}) \in \mathit{shared\_inputs}(\mathsf{Tx})$ do
 6:         $\mathsf{Obj}_s = \textsc{LoadLatestVersionObjects}(\mathsf{ObjID})$
 7:         if $\mathit{initial}(\mathsf{Obj}_s) \neq \mathsf{InitialVersion}$ then
 8:             return Error
 9:
10:     // Check 1.2: Check the transaction's validity (see Section 3.2).
11:     if $!\mathit{valid}(\mathsf{Tx}, [\mathsf{Obj}_o])$ then return Error
12:
13:     // Check 1.3: Try to acquire a mutex over $\mathit{inputs}(\mathsf{Tx})$
14:     guard = $\textsc{AcquireLocks}(\mathsf{Tx})$ ▷ Error if cannot acquire all locks
15:
16:     // Check 1.4: Lock all owned-objects.
17:     $\mathsf{TxSign} = \mathit{sign}(\mathsf{Tx})$
18:     for $\mathsf{ObjKey} \in \mathit{inputs}(\mathsf{Tx})$ do
19:         if $\mathsf{OwnedLock}[\mathsf{ObjKey}] == \mathsf{None}$ then
20:             $\mathsf{OwnedLock}[\mathsf{ObjKey}] = \mathsf{TxSign}$
21:         else if $\mathsf{OwnedLock}[\mathsf{ObjKey}] \neq \mathsf{TxSign}$ then
22:             return Error
23:
24:     return TxSign
```

Algorithm 2 Storage support (generic)

```
    // Check whether the certificate as already been executed.
 1: procedure AlreadyExecuted( TxCert)
 2:     $\mathsf{TxDigest} = \mathit{digest}(\mathsf{TxCert})$
 3:     $(\_, \mathsf{EffSign}) = \mathsf{Ct}[\mathsf{TxDigest}]$
 4:     if EffSign then return EffSign
        return None

    // All operations inside this function are atomic.
    // Note that all owned and shared objects have the same version $v$.
 5: procedure $\textsc{AtomicPersist}(\mathsf{TxCert}, \mathsf{EffSign}, [\mathsf{Obj}_{out}])$
 6:     $\mathsf{TxDigest} = \mathit{digest}(\mathsf{TxCert})$
 7:     $\mathsf{Ct}[\mathsf{TxDigest}] = (\mathsf{TxCert}, \mathsf{EffSign})$
 8:     for $\mathsf{Obj} \in \mathsf{Obj}_{out}$ do
 9:         $\mathsf{ObjKey} = \mathit{key}(\mathsf{Obj})$
10:         $\mathsf{ObjDB}[\mathsf{ObjKey}] = \mathsf{Obj}$
11:         if $\mathit{owned}(\mathsf{Obj})$ then
12:             $\mathsf{OwnedLock}[\mathsf{ObjKey}]$ = None
```

Process Transaction. Algorithm 1 shows how SUI LUTRIS processes transactions; that is, step ❷ of Figure 1 (see Section 2.4). The function LoadObjects (Line 3) simply loads the specified object(s) from the ObjDB store; LoadLatestVersionObjects (Line 4) loads the latest version of the specified object(s) from the ObjDB store; and AcquireLocks (Line 14) acquires a mutex for every owned-object transaction input. Upon receiving a transaction Tx a validator calls ProcessTx to perform a number of checks:

1) It ensures all object references $\mathit{inputs}(\mathsf{Tx})$ and the gas object reference in $\mathit{payment}(\mathsf{Tx})$ exist in the ObjDB store and loads them into [Obj]. For owned objects both the id and version should be available; for read-only or shared objects the object ID should exist, and for shared objects, the initial version specified in the input must match the initial version of the shared object returned by $\mathit{initial}(\cdot)$. This check implicitly ensures that all owned objects have the version number specified in the transaction since the call to LoadObjects (Line 3) loads the pair $\mathsf{ObjKey} = (\mathsf{ObjID}, \mathsf{Version})$ from the ObjDB store; ObjDB store holds a single entry (the latest version) per object.
2) It checks $\mathit{valid}(\mathsf{Tx}, [\mathsf{Obj}])$ is true. This step ensures the authentication information in the transaction allows access to the owned objects. That is, (i) the signer of the transaction must be the owner of all the input objects owned by an address input; (ii) the parent object of any included child object owned by another object should be an input to the transaction; and (iii) sufficient gas can be made available in the gas object to cover the minimum cost of executing the transaction.
3) It ensures it can acquire a lock for every owned-object transaction input; otherwise, it returns an error. Acquiring a lock ensures that no other task can concurrently perform the next step of the algorithm on the same input objects.
4) It checks that OwnedLock[ObjKey] for all owned $\mathit{inputs}(\mathsf{Tx})$ objects exist, and it is either None or set to *the same* Tx, and atomically sets it to TxSign. In other words, for each owned input version in the transactions: (i) a key for this object *exists* in OwnedLock and (ii) no other transaction $\mathsf{Tx}' \neq \mathsf{Tx}$ has been assigned as a value for this object version in OwnedLock, i.e. Tx is the first valid transaction seen using this input object. This is a key validity check to implement *Byzantine consistent broadcast* [13] and ensure safety.

Transaction processing ends if any of the checks fail and an error is returned. If all checks are successful then the validator returns a signature on the transaction, ie. a partial certificate TxSign. Processing a transaction is idempotent upon success, and always returns a partial certificate (TxSign) within the same epoch. Any party may collate a transaction and signatures (TxSign) from a set of validators forming a quorum for epoch e, to form a transaction certificate TxCert. It is *critical that this step happens for all transactions as it acts as spam protection* for order-execute consensus engines. In the original Bullshark [6] work it is trivial to drop the throughput to zero by a single malicious client sending corrupted transactions or duplicates. SUI LUTRIS prevents such attacks thanks to the stronger coupling with the state that allows only transactions with gas to make it to consensus.

Many tasks can call ProcessTx concurrently (or in parallel). SUI LUTRIS only acquires mutexes on the minimum amount of data: the owned-objects transaction inputs (Algorithm 1 Line 14).

Process user certificates. Algorithm 4 shows how SUI LUTRIS processes certificates; that is, step ❺ of Figure 1

Algorithm 3 Storage support (shared objects)

```
    // Assign locks to the shared objects referenced by TxCert.
 1: procedure WriteSharedLocks(TxCert, $v$)
 2:     TxDigest = digest(TxCert)
 3:     for (ObjID, InitialVersion) $\in$ shared_inputs(TxCert) do
 4:         Version = NextSharedLock[ObjID] || InitialVersion
 5:         SharedLock[(TxDigest, ObjID)] = Version
 6:         NextSharedLock[ObjID] = $v + 1$                ▷ Lamport timestamp

    // Check whether all shared objects referenced by TxCertare locked.
 7: procedure SharedLocksExist(TxCert)
 8:     TxDigest = digest(TxCert)
 9:     for ObjID $\in$ shared_inputs(TxCert) do
10:         if !SharedLock[(TxDigest, ObjID)] then
11:             return false                              ▷ lock not found
12:     return true

    // Ensure that TxCert is the next certificate scheduled for execution.
13: procedure CheckSharedLocks(TxCert)
14:     TxDigest = digest(TxCert)
15:     for ObjID $\in$ shared_inputs(TxCert) do
16:         Obj = ObjDB[ObjID]
17:         Version = version(Obj)
18:         if SharedLock[(TxDigest, ObjID)] $\neq$ Version then
19:             return false
20:     return true
```

(see Section 2.4). Algorithms 2 and 3 provide non-trivial support functions. Every certificate input to a function of Algorithm 4 is first checked to ensure it is signed by $2f+1$ validators. Upon receiving a certificate TxCert a validator calls ProcessCert to perform a number of checks:

1) It ensures *epoch*(TxCert) is the current epoch. This is a property of the quorum of signatures forming the certificate.
2) It loads the owned objects (Line 6) and shared objects (Line 7). Success in loading these objects ensures the validator already processed all past certificates concerning the loaded objects. If any object is missing, the validator aborts and returns an error. When loading shared objects, the function LoadObjects returns the shared object with the highest version number.
3) If the certificate contains shared objects, it ensures the certificate has already been sequenced by the consensus engine; otherwise, it forwards the certificate to consensus (Line 12). Finally, it checks the shared locks to ensure the current certificate TxCert is the next certificate scheduled for execution.

If all check succeeds, the transaction can be executed. The validator then atomically persists the execution results to storage (function AtomicPersist of Algorithm 2). It inserts the new (or mutated) objects in the ObjDB store and sets the version number of all owned and shared objects to the highest version number amongst all objects in the transaction plus 1 (Lamport timestamp); that is, the new version is $v = 1 + \max_{o \in [\mathsf{Obj}_o] \cup [\mathsf{Obj}_s]} \mathit{version}(o)$. The validator also persists the certificate along with the effects resulting from its execution; and updates the owned-object lock store to unblock future transactions using these objects. If transaction execution fails, Sui Lutris unlocks any owned objects used as input of the transaction. That is, it sets OwnedLock[ObjKey] = None, $\forall$ObjKey $\in$ *inputs*(Tx). Unlocking owned objects is essential to allow future transactions to re-use them. Gas payment is deduced from the payment objects whether the execution succeeds or fails.

Algorithm 4 Process certificate

```
Require: Input certificate ( TxCert) is signed by a quorum

    // Executed upon receiving a certificate from a user.
    // Many tasks can call this function.
 1: procedure ProcessCert(TxCert)
 2:     // Check 4.1: Ensure the TxCert is for the current epoch Epoch.
 3:     if epoch(TxCert) $\neq$ Epoch then return Error
 4:
 5:     // Check 4.2: Load objects from store, return error if missing object.
 6:     [$\mathsf{Obj}_o$] = LoadObjects(inputs(TxCert))
 7:     [$\mathsf{Obj}_s$] = LoadObjects(shared_inputs(TxCert))
 8:
 9:     // Check 4.3: Check the objects locks.
10:     if !SharedLocksExist(TxCert) then
11:         // Sequence the certificates
12:         ForwardToConsensus(TxCert)
13:         return
14:     if !CheckSharedLocks(TxCert) then return Error
15:
16:     // Execute the certificate.
17:     (EffSign, [$\mathsf{Obj}_{out}$]) = exec(TxCert, [$\mathsf{Obj}_o$], [$\mathsf{Obj}_s$])
18:     AtomicPersist(TxCert, EffSign, [$\mathsf{Obj}_{out}$])
19:     return EffSign

    // Executed upon receiving a certificate from consensus.
    // This function must be called by a single task.
20: procedure AssignSharedLocks(TxCert)
21:     // Ensure shared locks are assigned only once.
22:     if SharedLocksExist(TxCert) then
23:         return
24:
25:     // Extract the highest objects version.
26:     $v_o = 1$
27:     for (ObjID, Version) $\in$ inputs(TxCert) do
28:         $v_o = \max(v_o, \text{Version})$
29:     $v_s = 1$
30:     for (ObjID, InitialVersion) $\in$ shared_inputs(TxCert) do
31:         Version = NextSharedLock[ObjID] || InitialVersion
32:         $v_s = \max(vs, \text{Version})$
33:     $v_{max} = \max(v_o, v_s)$                          ▷ Lamport timestamp
34:
35:     // Lock all shared objects to $v_{max}$.
36:     WriteSharedLocks(TxCert, $v_{max}$)
```

Process consensus certificates. Upon receiving a certificate output from consensus, the validator calls AssignSharedLocks (Algorithm 4) to lock the transaction's shared-objects to a version number. This can be done without executing the transactions, only by inspecting and updating the transaction as well as the SharedLock and NextSharedLock tables. When an entry for a shared object does not exist in the tables it is assigned the initial version number given in the transaction input[2]. Otherwise, it is given the value in the NextSharedLock table. The NextSharedLock is updated with the Lamport timestamp [11] of the transaction: the highest version of

2. At the beginning of a new epoch, the most recent version from the objects table is used to support bootstrapping validators

all input objects used (owned, read-only, and shared) plus one. ASSIGNSHAREDLOCKS must be only called by a single task. After successfully calling ASSIGNSHAREDLOCKS, the validator can call (again) ProcessCert to execute the certificate.

Shared objects may be included in a transaction explicitly only for reads (we omit this special case from the algorithms for clarity). In that case, the transaction is assigned in SharedLock the version in the NextSharedLock table for the shared object. However, the version of the object in NextSharedLock is not increased, and upon transaction execution, the shared object is not mutated. In order to preserve the safety of dynamic accesses we make sure that within a Bullshark commit (level of concurrency) all read-only transactions on shared objects are executed on the initial version V and only after writes are allowed to execute and mutate the object. This facility allows multiple transactions executing in parallel to use the same shared object for reads. For example, it is used to update a clock object with the current system time upon each commit that transactions may read concurrently.

Additional checks. Algorithms 1 and 4 only describe the core validator operations. In practice, validators perform a number of extra checks to early reject duplicate messages. For instance, validators can easily check whether a certificate has already been executed by calling AlreadyExecuted (Algorithm 2). Such a check is useful to improve performance and prevent obvious DoS attacks but is not strictly needed for security (it does not guarantee idempotent validator replies on its own since both ProcessTx and ProcessCert can be called concurrently by multiple tasks).

Protecting Users from Bugs. Equivocation of an ObjKey is considered malicious in prior work and permanently locks the Obj forever. However, equivocation is often the result of a misconfigured wallet due to poor synchronization between gateways or poor gas predictions leading to re-submissions. As we discuss in Section 4.2 in SUI LUTRIS we only lose liveness until the end of the epoch. In the new epoch, the user can try again to access the same ObjKey with a fresh, hopefully correct, transaction. This is in contrast with every other consensusless blockchain that punishes all kinds of equivocation with a deadlock of the asset forever.

4. Long-Term Stability

Unlike prior consensusless systems, SUI LUTRIS is designed to work in production for long periods of time. For this to be practical we need protocols that allow for taking down validators and spinning up new ones, which is not possible in prior work which assumes infinite memory and static membership. This section outlines these protocols required to ensure its long-term stability, namely to produce checkpoints and enable reconfiguration.

To minimize latency, SUI LUTRIS executes before creating blocks. However, this makes it more complex for external parties to obtain proofs of transaction execution, perform complete audits, or even systematically replicate the state of the chain. To facilitate these functions, we introduce checkpoints. Further, to tolerate mistakes we need to safely unlock objects that were mistakenly locked through client double spending bugs. The validator set, and validator voting power, also need to evolve over time, to support permissionless delegated proof of stake. These functions are supported in the epoch-change and reconfiguration process. It persists all final transactions and their effects across epochs. Enables client to unlock objects involved in partial locked transactions; and is a secure means for validators to enter or exit the system without affecting its liveness and performance. The period between reconfiguration is a rare time of perfect synchrony and consistency across all validators, when software upgrades can be performed and global incentives and rewards can be distributed.

4.1. Checkpoints

SUI LUTRIS validators emit a sequence of certified *checkpoints* each containing an agreed-upon sequence of transactions, the authorization path of the transactions, and a commitment to their effects. These form a hash-chain, which is the closest SUI LUTRIS has to the blocks of a traditional blockchain. Checkpoints are used for multiple purposes: they are gossiped from validators to full nodes to update them about the state of the chain; they are used by validators to perform synchronization in case they fall behind in execution; as well as to bring new validators up to speed with the state at the start of each epoch. Checkpoints, packaged along with the transactions and the effects structures they contain are also the canonical historical record of execution used for audit.

Checkpoints are created asynchronously and/or in parallel with execution and attaining finality. However, a key safety property holds due to our reconfiguration protocol design. Namely, if a transaction is final within an epoch then it will be present in a checkpoint within the epoch and will persist across epochs. And conversely, if a transaction is present within an epoch its effects are final. The checkpoint creation process guarantees these invariants.

Checkpoint creation. Upon receiving a valid certificate a correct validator records it and commits to including it in a checkpoint before the end of the epoch. A validator schedules all certificates for sequencing using the consensus engine. For owned transactions this does not block execution, which can continue before the transaction is sequenced; in the case of shared object transactions execution resumes once the transaction certificate has been sequenced and the shared object locks are determined (see Section 2.4). A correct validator will not proceed to end an epoch before all valid certificates it received are sequenced into checkpoints (either as a result of itself or others inserting them into the consensus engine).

Periodically, using a deterministic rule, validators pick a consensus commit to use as a checkpoint (our current implementation checkpoints each and every commit separately). The new checkpoint contains all transactions present in the commits since the last checkpoint and any additional transaction required for the execution to be causally complete. If a transaction's certificate exists more than once, the first occurrence is taken as the canonical one and the accompanied user signature is included in the checkpoint for audit purposes. Note that due to asynchrony and failures, the certificates sequenced may not be in causal order, or may contain 'gaps', i.e., missing transaction dependencies. Checkpoint creation waits for all transactions necessary for the checkpoint to be causally complete to be within a commit, sorts them in canonical topological order, and includes them in the checkpoint.

Each correct validator uses the same sequence of commits to forming the checkpoint and therefore will yield and sign the same checkpoint. The checkpoint header and validator signature of each validator can then be sequenced and the first $2f + 1$ forms the canonical certificate for the new checkpoint. Note that due to the need to fill causal gaps in execution using subsequent commits, a checkpoint for a commit may only be constructed after a future commit.

Properties and uses. Eventually, the complete set of transactions will be sent to be checkpointed, and the entire causal history will be in the checkpoint history. Informally, since a final transaction is executed by $2f + 1$ nodes, $f + 1$ correct nodes will try to sequence it until they succeed. As we discuss in the next section $2f + 1$ validators need to consent to close the epoch and by quorum intersection, at least one of them will delay until a transaction that is final is in the checkpoint. Causally preceding transactions must have been executed and be final, thus they will eventually be sequenced leading to a complete causal history being checkpointed. All these processes may happen eventually without blocking earlier transactions finalize or settle.

Checkpoints are used as part of reconfiguration but are also used as a simpler synchronization mechanism between validators, as well as full nodes. Since correct validators eventually include all certificates in consensus, this ensures that all correct validators will eventually see and execute all transactions without the need for complex state synchronization protocols. Full nodes, in turn, may download the sequence of checkpoints to ensure they have a complete replica of the state of the system. In peer-to-peer synchronization full nodes can use the sequence number of the checkpoint as a reference when requesting blocks of information from other peers in sequence or in parallel.

Algorithm 5 Reconfiguration Contract

```
// Smart contract state
T                                    ▷ Minimum stake to become a validator
S                          ▷ Last sequence number before starting epoch change
total_old_stake = GenesisStake ▷ Total stake of the old committee
total_new_stake = 0              ▷ Total stake of the new committee
old_keys = GenesisKeys          ▷ Identifiers of the old committee members
new_keys = {}            ▷ Identifiers of the new committee members
epoch_edge = 0 ▷ Sequence number of the last epoch's checkpoint
state = Register               ▷ Current state of the smart contract
stake = 0                   ▷ Variable counting the accumulated stake

// Step 1: Users register to become the next validators.
1: function REGISTER(sender)
2:     if state ≠ Register then return
3:     if sender.stake ≥ T then
4:         new_keys = new_keys ∪ sender
5:         total_new_stake += sender.stake

// Step 2: New validators signal they are ready to take over.
6: function READY(sender)
7:     seq = GETLATESTCHECKPOINTSEQ()
8:     if state = Register and seq ≥ S then
9:         state = Ready
10:    if state ≠ Ready then return
11:    if sender ∈ new_keys then
12:        stake += sender.stake
13:    if stake ≥ 2 * total_new_stake/3 + 1 then
14:        state = End-of-Epoch
15:        stake = 0
16:        PAUSETXLOCKING                   ▷ Stop signing messages

// Step 3: Old validators signal the epoch can safely finish.
17: function END-OF-EPOCH(sender)
18:    if state ≠ End-of-Epoch then return
19:    if sender ∈ old_keys then
20:        stake += sender.stake
21:    if stake ≥ 2 * total_old_stake/3 + 1 then
22:        state = Handover
23:        stake = 0
24:        epoch_edge = GETLATESTCHECKPOINTSEQ()

// Step 4: The new validators take over.
25: function HANDOVER(sender)
26:    if state ≠ Handover then return
27:    seq = GETLATESTCHECKPOINTSEQ()
28:    if seq ≥ epoch_edge + 1 then
29:        old_keys = new_keys
30:        total_old_stake = total_new_stake
31:        state = Register
32:        new_keys = {}
33:        total_old_stake = 0, epoch_edge = 0
34:        SHUTDOWN                  ▷ Old validators can shutdown
```

4.2. Committee Reconfiguration

Reconfiguration occurs between epochs when the current committee is replaced by a new committee. Other changes requiring global coordination, such as software updates, and parameter updates also take effect between epochs. Immutable objects, such as system parameters or software packages, may be mutated in that period.

The goal of the reconfiguration protocol is to preserve safety of transactions between epochs, while allowing for liveness recovery of equivocated transactions. To this end, we require that if a transaction Tx was committed during epoch e or before, no conflicting transaction can be committed after epoch e. This is trivial to ensure when running only a consensus protocol since a reconfiguration event logged on-chain clearly separates transactions committed in epoch e from transactions committed in epoch $e + 1$. However, in SUI LUTRIS solutions are not as straightforward. More specifically, SUI LUTRIS requires a final transaction at epoch e to have its effects reflected in all subsequent epochs.

The main challenge for SUI LUTRIS reconfiguration at each validator is the race between committing transactions and constructing checkpoints, that are running potentially asynchronously to each other. If a checkpoint snapshots the end-state of epoch e at time T and is only committed at time $T+1$, we cannot set that checkpoint as the initial state of epoch $e+1$. If we did, all transactions happening during the last timestamp are at risk of being unsafe when validators drop their OwnedLock to allow for liveness recovery of equivocated transactions. This is not an issue for the consensus path of SUI LUTRIS since we can define as end state the checkpoint at time T plus all transactions ordered before it is committed at $T+1$. That end state is well-defined thanks to the total ordering property of consensus. Unfortunately, it is not easy to establish a set of committed transactions on the consensus-less path.

Remember that the consensusless path works in two phases, first a transaction locks the single-owner object and produces a certificate. Then this certificate is sent as proof to the validators who reply with a signed effects certificate (execution). The safety risk during a reconfiguration is that a transaction is executed during the transition phase without the checkpoint recording it. However, SUI LUTRIS splits reconfiguration into multiple steps instead of doing it atomically. As part of the last step, we pause the consensusless path. This allows us to show that if a transaction had executed before the reconfiguration message then there is no safety risk because we can guarantee that at least one honest party will persist the execution in the state. We enforce this by introducing an *End-of-Epoch* message that the committee members of the current epoch send when they have seen all the certificates they processed in the consensus output sequence. The new committee takes over completely only after $2f+1$ such End-of-Epoch messages are ordered. As a result, SUI LUTRIS manages to preserve safety without needing to block for long periods.

4.3. The SUI LUTRIS Reconfiguration Protocol

The SUI LUTRIS reconfiguration logic is coded as the smart contract shown in Algorithm 5. Once a quorum of stake for epoch e votes to end the epoch, authorities exchange information to commit to a checkpoint, determine the next committee, and change the epoch. More specifically, reconfiguration happens in four steps, (i) *stake recalculation*, (ii) *ready new committee*, (iii) *End-of-Epoch*, and (iv) *Handover*. Each step is handled by the interaction of the new and old committee members with the smart contract functions. This is key for the correctness of the protocol.

The smart contract is parametrized with (i) the total number of checkpoints before initiating the epoch change protocol S and (ii) the minimum amount of stake T required to become a validator. The function GETLATESTCHECKPOINTSEQ (Line 24) simply loads the latest checkpoint sequence number known by the validator.

Step 1: Registration of new validators. Users wishing to become validators in the next epoch submit a transaction to the reconfiguration contract calling the REGISTER function. This function establishes the static stake distribution for the next epoch.[3] The smart contract accepts registrations until the S$^{\text{th}}$ is created.

Step 2: Ready new committee. Before taking over committee operations, future validators run a full node to download the required state to become a validator. At the bare minimum, they ensure their following stores are up-to-date: NextSharedLock, Ct, and ObjDB (see Section 3.3). Once a validator for the new epoch is ready to start validating they call the READY function to signal they have successfully synchronized the required state. This function can only be called towards the end of the epoch, after the creation of S checkpoints (Line 8). The cutoff period for the epoch is when a quorum of new validators is ready. At this point, the old committee stops locking objects (Line 16), i.e., executing Algorithm 1.

Step 3: End of Epoch. When the new committee is ready the old committee only runs consensus and their only job is to make sure all the transactions they executed are sequenced by the consensus engines (so that they are part of the next checkpoint). To this end, they stop accepting certificates from clients and instead make sure that all the certificates they have processed are sequenced by the consensus engine. They then call the END-OF-EPOCH function. This means that any transactions submitted by clients for this epoch are discarded with an end-of-epoch message and need to be resent with an updated epoch number.

Step 4: Handover. After $2f+1$ old validators call the END-OF-EPOCH function, the system enters the handover phase. After an extra checkpoint (Line 28) anyone can call the function HANDOVER that effectively terminates the epoch. At this stage, the state of the smart contract is reset and the old validators can shut down (Line 34). If a validator also participates in the new epoch, it must perform the following operations before entering the next epoch:

- It drops the OwnedLock and SharedLock stores (see Section 3.3).
- It rolls-back execution of any transaction that did not appear in any checkpoint so far. As we discussed this is safe as these transactions were not final.

The protocol design decouples all the essential steps needed for a secure reconfiguration. This decoupling allows for the necessary logic of defining the new committee, providing the new committee sufficient time to bootstrap from the actual handover, and preserving the safety of consensusless transactions across epochs. As a result service interruption is minimized to only a few seconds.

3. As this part is contained within one epoch the only risk is that some validators are censored. Fortunately, Bullshark [6] (the consensus protocol used by SUI LUTRIS) provides sufficient chain quality to allow honest validators to lock their stake.

5. Implementation

We implement a networked multi-core SUI LUTRIS validator in Rust forking the FastPay[4], Narwhal[5], and Bullshark[6] projects. It uses Tokio[7] for asynchronous networking, fastcrypto[8] for elliptic curve based signatures. Datastructures are persisted using RocksDB[9]. We use QUIC[10]—to achieve reliable authenticated point-to-point channels. The implementation of SUI LUTRIS is around 30 kLOC and over 10 kLOC of tests. Contrarily to most prototypes, our implementation is production-ready and fully-featured. It runs at the heart of a major new blockchain (integrated in 350 kLOC) mainnet. As of July 30, 2023, the mainnet is operated by 105 geo-distributed heterogeneous validators, has processed over 730 million certificates, at the rate of over 2.5 million certificates a day, in over 9 million checkpoints, and has undergone over 100 epoch changes. It stores over 80 million objects, owned by over 2.5 million addresses, defined by over 6,000 Move packages. We are open sourcing our implementation of SUI LUTRIS[11].

6. Evaluation

We evaluate the throughput and latency of our implementation of SUI LUTRIS through experiments on AWS. We particularly aim to demonstrate the following claims. **C1** SUI LUTRIS achieves high throughput despite its consensus-intensive checkpointing mechanism. **C2** SUI LUTRIS finalizes owned-objects transactions with sub-second latency (in the WAN) for small and medium committee sizes or when the system is under low load. **C3** SUI LUTRIS is robust when some parts of the system inevitably crash-fail. **C4** SUI LUTRIS validators are capable of quickly recovering after crashes without visible performance impact. **C5** SUI LUTRIS's epoch change mechanism only causes a small disruption. Experimental evaluation of BFT protocols in the presence of Byzantine faults is an open research question [14]. However, we provide extensive safety and liveness proofs in Appendix A.

6.1. Experimental Setup

To demonstrate those claims we compare the fast-path of SUI LUTRIS with an implementation of Bullshark that uses the same execution engine and consensus protocol as SUI LUTRIS. This means that although Bullshark is used as a baseline it also represents the actual performance of SUI LUTRIS when deployed only with a shared objects workload. We use our implementation of Bullshark, which is an extension of the original codebase, for three reasons. First, the original Bullshark protocol only measures throughput by counting the number of bytes committed; as a result the throughput reported did not correspond to the performance of a real (production-ready) system because there was no execution or state to be updated (i.e., Bullshark is a BAB [15] and not an SMR [16]). Second, as a result of counting bytes there was no duplicate suppression, resulting in a potential $O(n)$ duplicate transactions. In our implementation we report the goodput of the system, i.e., the number of distinct certificates finalized by the system. Finally, due to lack of state it was trivial to DoS the original Bullshark by sending invalid and malformed transactions such that the goodput would drop to zero. Instead, in our experiments we ensures that all transactions forwarded to the consensus engine use valid gas objects that are spent regardless of whether the transaction bytes are valid.

We deploy a fully-featured SUI LUTRIS testbed on AWS, using `m5d.8xlarge` instances across 10 different AWS regions: N. Virginia (us-east-1), Oregon (us-west-2), Canada (ca-central-1), Frankfurt (eu-central-1), Ireland (eu-west-1), London (eu-west-2), Mumbai (ap-south-1), Singapore (ap-southeast-1), Tokyo (ap-northeast-1), and Sydney (ap-southeast-2). Validators are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 22.04. SUI LUTRIS persists all data on the NVMe drives provided by the machine (rather than the root partition). We select these machines because they provide decent performance and are in the price range of 'commodity servers'.

In the following graphs, each data point in the latency graphs is the average of the latency of all transactions of the run, and the error bars represent one standard deviation (errors bars are sometimes too small to be visible on the graph). We instantiate several geo-distributed benchmark clients submitting transactions at a fixed rate for a duration of 10 minutes; unless specified otherwise each benchmark client submits at most 350 tx/s and the number of clients thus depends on the desired input load. When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when it assembles an effect certificate over the transaction (and the transaction has reached settlement finality, see Section 2). The latency of SUI LUTRIS shared-object transaction finality, are lower than the ones of Bullshark in our experiments: SUI LUTRIS reaches transaction finality before execution with a latency equal to the owned-object transaction. The Bullshark latency is similar to SUI LUTRIS's shared-object settlement latency instead (see Section 2). When referring to *throughput*, we mean the number of effect certificates over *distinct* transactions over the entire duration of the run. Since a SUI LUTRIS transaction may contain a block of multiple operations (which are also transactions) we report throughput as certificates per second (cert/s) to denote distinct unrelated transaction throughput.

Transactions processed by SUI LUTRIS are payment transfers and transactions processed by Bullshark are increments of a shared counter which simulates the sequence number of a shared account. We do not use the (real) mainnet load for our experiments because the system's launch is

4. https://github.com/novifinancial/fastpay
5. https://github.com/facebookresearch/narwhal
6. https://github.com/asonnino/narwhal/tree/bullshark
7. https://tokio.rs
8. https://github.com/MystenLabs/fastcrypto
9. https://rocksdb.org
10. https://github.com/quinn-rs/quinn
11. https://github.com/mystenlabs/sui

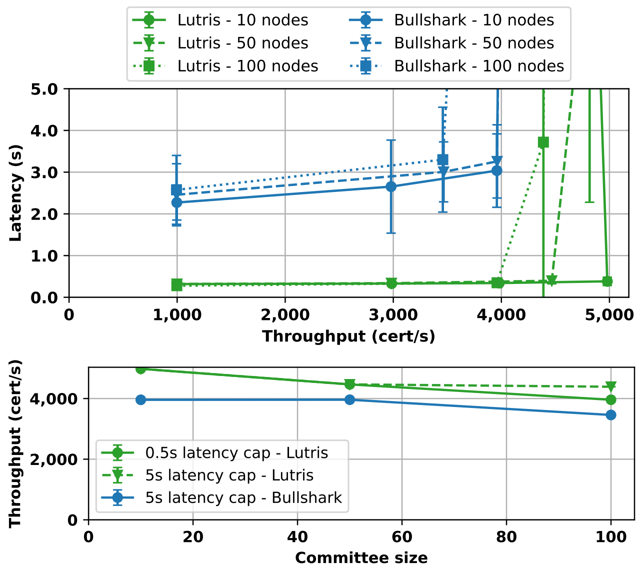

Figure 2. SUI LUTRIS and Bullshark WAN latency-throughput with 10, 50, and 100 validators (no faults).

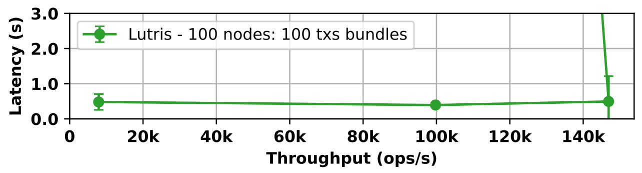

Figure 3. SUI LUTRIS latency-throughput with bundles of 100 transactions.

very recent and there is current not enough historic data to produce more than a few seconds of benchmarks.

We benchmark the version of the codebase running on `mainnet-v1.4.3`[12] and open-source all orchestration scripts, benchmarking scripts, and measurements data to enable reproducible evaluation results[13]. Appendix B provides a tutorial to reproduce our experiments.

6.2. Benchmark in the Common Case

Figure 2 compares the performance of SUI LUTRIS with the baseline Bullshark when running with 10, 50, and 100 non-faulty validators. The lower part of the figure provides another view of the same data by showing the maximum achievable throughput while keeping the latency below 0.5 seconds and 5 seconds (the system's SLA).

Bullshark finalizes 4,000 cert/s with a committee size of 10 or 50 and 3,500 cert/s with a committee size of 100. In comparison, SUI LUTRIS finalizes 5,000 cert/s with a committee size of 10 or 50, and 4,000 cert/s with a committee size of 100. In all cases, SUI LUTRIS's throughput outperforms Bullshark's despite its checkpointing mechanism (see Section 4.1) that sequences all certificates through the consensus engine. This observation validates our claim **C1**.

Regardless of the committee size, Bullshark's latency is about 3 seconds while SUI LUTRIS's latency is less than 0.5 seconds – corresponding to a 6x latency improvement. This observation validate our claim **C2**.

Transaction bundles. SUI LUTRIS allows a client to 'bundle' multiple transactions into a single transaction, and signing only the bundle rather than each individual transaction. This is useful for large exchanges and corporate entities submitting numerous transactions on behalf of their users. Bundling transactions reduces the number of messages exchanged between the client and the system and greatly reduces the cost of signature verification. Figure 3 shows the performance of a 100-validator deployment of SUI LUTRIS executing a payload composed of bundles of 100 transactions[14]. The graphs shows a peak at 150,000 opts/s indicating that SUI LUTRIS can process 1,500 bundles of 100 transactions per second while keeping the latency below 0.5 seconds.

6.3. Benchmark with Faults

Figure 4 compares the performance of SUI LUTRIS with the baseline Bullshark for a 10-validator deployment when the system experiences (crash-)faults; after running without faults for one minute, 1 and 3 validators crash. The figure shows that Bullshark can finalize 3,500 cert/s in about 5 seconds and 3,000 cert/s in about 7.5 seconds when respectively 1 and 3 validators crashes. In contrast, SUI LUTRIS is largely unaffected by validator's crashes: it can still finalize over 4,000 cert/s with a latency of less than 0.5 seconds. We thus observe that SUI LUTRIS provides up to 15x latency reduction when the system experiences (crash-)faults and validate our claim **C3**.

Figure 5 shows the throughput and latency of SUI LUTRIS and Bullshark when 3 validators are crashing and recovering. The plots are divided in 5 zones by vertical black lines; no validators are crashed in the first zone; then respectively 1, 2, and 3 validators are crashed in the 2nd, 3rd, and 4th zone; and all validators recover in the last zone. The systems are submitted to a constant load of 3,000 cert/s (regardless of the number of faults). Each point on the graphs is the average metric observed by the clients (averaged over all clients and with a 15-seconds window). As expected, the throughput of Bullshark slightly degrades (barely visible) and its latency increases when the number of crash-faults increases. The 5th zone shows that performance starts recovering when all validators recover. The slight delay in recovery is due to the overhead required by our orchestration to apply changes to the deployment environment. In contrast, SUI LUTRIS is largely unaffected by crash-recovering validators. These observations validate our claim **C4**.

12. https://github.com/asonnino/sui/tree/sui-lutris (commit `7f3d922`)
13. https://github.com/asonnino/sui-paper/tree/main/data

14. These measurements required 40 benchmark clients, each an an individual machine, to avoid client-side bottlenecks.

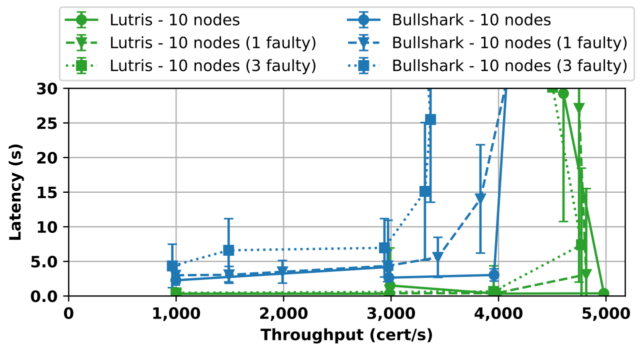

Figure 4. SUI LUTRIS and Bullshark WAN latency-throughput with 20 validators (1, 3, and 6 faults).

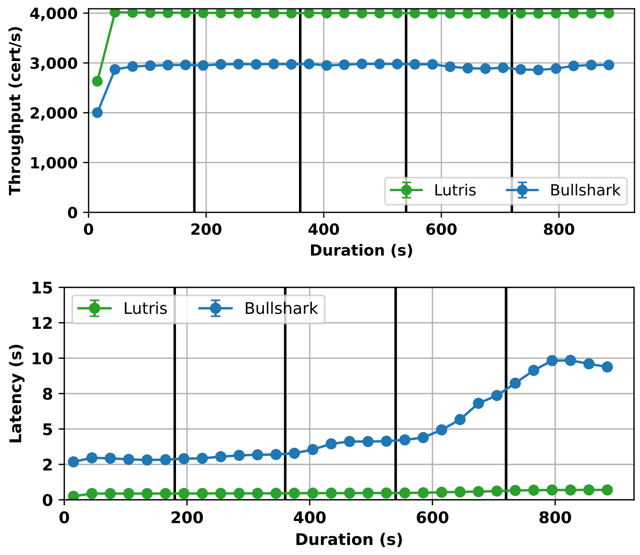

Figure 5. Performance of a 10-validators committee when up to 3 validators crash and recover.

6.4. Stability during Epoch Changes

Figure 6 shows the throughput and latency over time of 10-validator deployments of SUI LUTRIS and Bullshark. The systems are submitted to a constant load of 3,000 cert/s for about 35 minutes during which the systems undergo 3 epoch changes (one every 10 minutes, indicated by black vertical lines). We observe that the performance of both SUI LUTRIS (and Bullshark) are largely unaffected by the epoch changes, thus validating our claim **C5**.

7. Related and Future Work

In our deployment, we demonstrated the power of SUI LUTRIS which manages to provide a fully expressive smart contract platform for mutually distrustful parties to collaborate and exchange value. In one sentence, SUI LUTRIS is the first secure smart-contract platform that provides sub-second finality for distributed ledger transactions without compromising the expressiveness and the performance of state-of-the-art consensus protocols such as Bullshark. To achieve

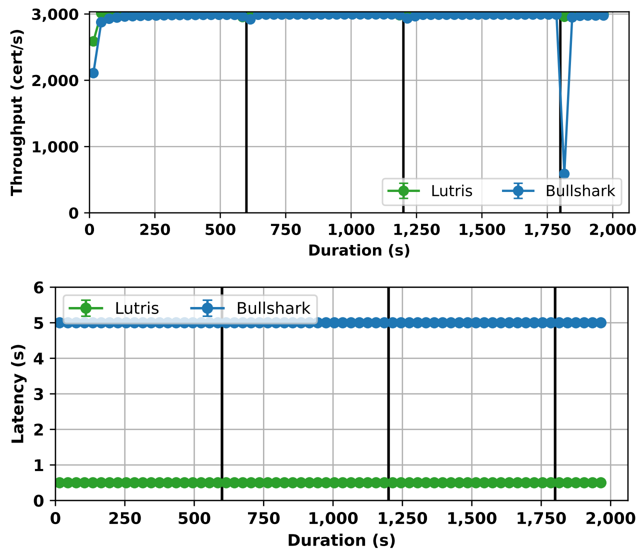

Figure 6. Performance of a 10-validators committee during epoch changes.

this it combines said state-of-the-art consensus protocols with the ideas of the FastPay [1] low-latency settlement systems in order to gain the ability to operate on arbitrary objects through user-defined smart contracts, and with a delegated proof-of-stake committee [17].

The SUI LUTRIS owned object path is based on Byzantine consistent broadcast [18]. Previous works suggested using this weaker primitive to build payment systems [2], [3], [19] but lack an integration with a consensus path making it both impractical to run for a long-time (no garbage-collection or reconfiguration) as well as limited functionality (only payments) and usability (client-side bugs result in permanent loss of funds).

Other systems similar to SUI LUTRIS are Astro [3] and ABC/CoD [20], [21]. Astro relies on an eager implementation of *Byzantine reliable broadcast* [13] which achieves *totality* [13] without relying on an external synchronizer at the cost of higher communication in the common case. Additionally, Astro is designed as a standalone payment system but does not handle checkpointing or reconfiguration. Similar to Astro and FastPay, ABC [20] proposes a relaxed notion of consensus where termination is only guaranteed for honest senders, this however is quite disruptive for the client experience as simple mistakes cause complete loss of access to user assets. Additionally ABC provides no implementation or evaluation and is also limited to payments.

As part of SUI LUTRIS, we require integration with a consensus protocol able to keep up with checkpointing the throughput the fast-path is able to execute. For this reason, we chose the use of Narwhal-Bullshark [6], [8] in a variant without asynchronous fallback [22], due to their reported and observed high performance. Similarly, SUI LUTRIS requires integration with a deterministic execution engine in order to provide end-to-end application level semantics instead of simply ordering bytes. In our evaluation, we interface with a modified version of Move that exposes objects as programmable entities because of its secure interfaces and code maturity. Nevertheless, our design is modular [23] and can interface both with any total-ordering protocol [24], [25], [26], [27], [28], [29] as well as with any deterministic execution engine [30], [31].

8. Conclusion

SUI LUTRIS is the first smart-contract platform that forgoes consensus for single-writer operations and only relies on consensus for settling multi-writer operations, combining the two modes securely. We describe system aspects to achieve long-term stability and production-readiness that are typically overlooked by research prototypes. It proposes checkpointing transactions into a sequence after execution (for large categories of transactions) to reduce the need for agreement and latency, and shows how to safely reconfigure the system with minimal downtime. SUI LUTRIS is currently running in production as part of a major smart-contract platform. Its mainnet is operated by 105 geo-distributed heterogeneous validators (Jul. 30, 2023) and processes an average of over 2.5 million certificates a day. At peak, it sustained a chain utilization of over 65 million certificates a day, which was higher than the total number of transactions on all blockchains combined, during that period.

Acknowledgments

This work is funded by MystenLabs. We thank the Mysten Labs Engineering teams for valuable feedback broadly, and specifically Dmitry Perelman and Todd Fiala for managing the implementation effort. A number of folks contributed to specific aspects of the implementation of SUI LUTRIS (amongst many other contributions to the overall blockchain): Francois Garillot, Laura Makdah, Mingwei Tian, Andrew Schran, Sadhan Sood and William Smith implemented and optimized aspects of both SUI LUTRIS and Narwhal / Bullshark consensus; Alonso de Gortari oversaw the crypto economics of the blockchain, and Emma Zhong, Ade Adepoju, Tim Zakia and Dario Russi designed and implemented staking and gas mechanisms. Adam Welc designed several Move tools and provided great feedback on the manuscript. We also extend our thanks to Patrick Kuo, Ge Gao, Chris Li, and Arun Koshy for their work on the SUI LUTRIS SDK, clients, and RPC layer; Kostas Chalkias, Jonas Lindstrøm, and Joy Wang designed and implemented cryptographic components.

References

[1] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance byzantine fault tolerant settlement," in *Advances in Financial Technologies (AFT)*, 2020.

[2] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "The consensus number of a cryptocurrency," in *Principles of Distributed Computing (PODC)*, 2019.

[3] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y.-A. Pignolet, D.-A. Seredinschi, A. Tonkikh, and A. Xygkis, "Online payments by merely broadcasting messages," in *Dependable Systems and Networks (DSN)*, 2020.

[4] M. Baudet, A. Sonnino, M. Kelkar, and G. Danezis, "Zef: Low-latency, scalable, private payments," *arXiv preprint*, 2022.

[5] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, R. in, D. Russi, S. Sezer, T. Zakian, and R. Zhou, "Move: A language with programmable resources," https://move-book.com, 2019.

[6] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag BFT protocols made practical," in *Computer and Communications Security (CCS)*, 2022.

[7] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, 1988.

[8] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a DAG-based mempool and efficient BFT consensus," in *European Conference on Computer Systems (EuroSys)*, 2022.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

[10] Committee on Payment and Settlement Systems, "A glossary of terms used in payments and settlement systems," Bank for International Settlement (BIS) Report, 2003.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019.

[12] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2001.

[13] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[14] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: BFT systems made robust," *Principles of Distributed Systems (OPODIS)*, 2020.

[15] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," *Information and Computation*, 1995.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *Computing Surveys (CSUR)*, 1990.

[17] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "SoK: Consensus in the age of blockchains," in *ACM Advances in Financial Technologies (AFT)*, 2019.

[18] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[19] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "AT2: asynchronous trustworthy transfers," *CoRR*, 2018.

[20] J. Sliwinski and R. Wattenhofer, "Abc: Asynchronous blockchain without consensus," ArXiv preprint, 2019.

[21] J. Sliwinski, Y. Vonlanthen, and R. Wattenhofer, "Consensus on demand," in *Stabilization, Safety, and Security of Distributed Systems*, 2022.

[22] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: The partially synchronous version," *ArXiv preprint*, 2022.

[23] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Proof of availability & retrieval in a modular blockchain architecture," *Financial Cryptography and Data Securty (FC)*, 2023.

[24] D. Malkhi and K. Nayak, "Hotstuff-2: Optimal two-phase responsive bft," *Cryptology ePrint*, 2023.

[25] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Financial Cryptography and Data Security (FC)*, 2022.

[26] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” in *Security and Privacy (SP)*, 2018.

[27] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, “Chainspace: A sharded smart contracts platform,” in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[28] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *Usenix OSDI*, 1999.

[29] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers,” in *European Symposium on Security and Privacy (EuroS&P)*, 2020.

[30] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, “Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing,” in *Principles and Practice of Parallel Programming*, 2023.

[31] W. Zhang and T. Anand, “Ethereum architecture and overview,” in *Blockchain and Ethereum Smart Contract Solution Development: Dapp Programming with Solidity*. Springer, 2022.

Appendix A. Security Proofs of SUI LUTRIS

SUI LUTRIS satisfies validity, safety, and liveness as informally described in Section 2.3. These properties hold against any polynomial-time constrained adversary as long as the following assumptions hold:

- **BFT:** Every quorum of $3f+1$ validators contains at most f Byzantine validators. Correct validators of previous epochs never leak their signing keys.
- **Network:** The network is partially synchronous [7]. SUI LUTRIS operates in the partial synchrony model only due to our use of Bullshark [6] as consensus protocol (it would operate in asynchrony if we used Tusk [8] instead).

When referring to a ‘valid’ transaction in the sections below, we mean a transaction that successfully pass check (1.2) defined of Algorithm 1 (Section 2.4).

A.1. Validity

We show that correct SUI LUTRIS validators only execute valid transactions. Validity holds unconditionally to the network assumption. The protocol described in Section 2.4 ensures validity as long as the BFT assumption holds. However, our implementation ensures validity unconditionally because correct validators re-run all checks of Algorithm 1 upon processing a certificate. Should the BFT assumption break, they would thus early-reject certificates over invalid transactions before even starting to process them.

Lemma 1 (Valid Certificates)**.** *All certified transactions are valid with respect to the authorization rules relating to owned object (defined in Section 2.2).*

Proof. Certificates are signed by at least $2f+1$ validators, out of which at least $f+1$ are correct. Correct validators only sign a transaction after performing check (2) of Algorithm 1, as a result, no invalid transaction will ever be signed by a correct authority, and will thus never be certified. □

Theorem 1 (SUI LUTRIS Validity)**.** *State transitions at correct validators are in accordance with (i) the authorization rules relating to owned object (defined in Section 2.2), and (ii) the Move smart contract logic constraining valid state transitions on objects of defined types.*

Proof. State transitions at correct validators only happen if a valid certificate TxCert in Algorithm 4 exists and is processed. Point (i) is thus directly proven by the application of Lemma 1.Point (ii) is proven by noting that correct validators only apply a state transition (Line 18 of Algorithm 4) after calling $exec(\mathsf{TxCert})$ at Line 17 of Algorithm 4. This function only produces $[\mathsf{Obj}_{out}]$ by calling the Move smart contract logic constraining valid state transitions on objects of defined types. □

A.2. Safety

We prove the safety of the SUI LUTRIS protocol. That is, we show that at the start of every epoch, all correct validators have the same state. We prove safety using three main ingredients: (i) correct validators execute the same set of transactions, (ii) correct validators execute those transactions in the same order whenever (partial) order matters, and (iii) the execution of those transactions causes the same state transition across all correct validators.

Execution set. We start by showing that all correct validators eventually execute the same set of transactions.

Lemma 2 (Owned-Object Execution Set)**.** *No correct validators have executed a different set of owned object transactions by the end of epoch e.*

Proof. Correct validators assemble checkpoints by observing the sequence of transaction certificates committed in consensus (Section 4.3). By the agreement properties of the consensus protocol, all correct validators obtain the same sequence of certificates. They then use those ordered certificates to construct the same checkpoints since creating checkpoints is a deterministic process given the certificate sequence input. Correct validators execute all transactions within the checkpoints they assemble. Additionally, at the end of every epoch validators revert the execution of any owned-object transaction not included in a checkpoint (see Section 4.1), hence only owned-object transactions included in a checkpoint persist. Reverting these transactions is safe as only transactions included in a checkpoint are final (see Theorem 3). □

Lemma 3 (Shared-Object Execution Set)**.** *No correct validators have executed a different set of shared object transactions by the end of epoch e.*

Proof. Correct validators execute all shared object transactions sequenced by the consensus protocol before the last checkpoint of epoch e (see Section 4.2). By the agreement property of the consensus protocol, all correct validators obtain the same sequence and thus execute the same set of shared-object transactions. □

Execution order. We now show that correct authorities execute conflicting transactions in the same order.

Lemma 4 (BCB Consistency). *No two conflicting transactions, namely transactions sharing the same owned inputs objects, object version, and epoch, are certified.*

Proof. The proof of this lemma directly follows from the consistency property of Byzantine consistent broadcast (BCB) over the label $(e, \mathsf{ObjID}, \mathsf{Version})$. Let's assume two conflicting transactions Tx_A and Tx_B taking as input the same owned object Obj with version Version are certified during the same epoch e. Then $f+1$ correct validators performed check (1.4) of Algorithm 1 and produced TxSign_A and $f+1$ correct validators did the same and produced TxSign_B. A correct validator performing check (1.4) cannot successfully process both (conflicting) Tx_A and Tx_B; indeed it will return an error at Line 21. As a result, a set of $f+1$ correct validators produced TxSign_A but not TxSign_B, and a distinct set of $f+1$ correct validators produced TxSign_B but not TxSign_A. Hence there should be $f+1+f+1 = 2f+2$ correct validators additionally to the f byzantine. However $N = 3f+1 < 3f+2$ hence a contradiction. □

Lemma 4 operates over the label $(e, \mathsf{ObjID}, \mathsf{Version})$ rather than only $(\mathsf{ObjID}, \mathsf{Version})$ because of check (1.4) of Algorithm 1. This check relies on the integrity of the store OwnedLock, which is dropped upon epoch change (Section 4.3). This is however not a problem because certificates carry their epoch number and are only valid for a single epoch (see check (4.1) of Algorithm 4).

Lemma 5 (Shared-Locks Consistency). *The shared lock store SharedLock of correct validators are never conflicting; that is, the shared lock stores of correct validators are either identical or a subset of each other.*

Proof. Let's assume two correct validators update their store SharedLock by assigning different version numbers to a shared object Obj of a certificate TxCert. Correct validators only assign a version number to Obj after sequencing TxCert through consensus (Section 3.4). They then call AssignSharedLocks (Algorithm 4) and Line 36 of Algorithm 4 assigns a version number to Obj. The function AssignSharedLocks is deterministic and thus the version number assigned to Obj depends only on the consensus output sequence. As a result, two correct validators assign a different version to Obj only if they receive a different consensus output sequence. However, by the agreement property of the consensus all correct validators received the same output sequence, hence a contradiction. □

Lemma 6 (Owned Objects Sequential Execution). *If two certificates TxCert and TxCert′ both take as input the same owned object* Obj *(and no shared objects), all correct validators execute TxCert and TxCert′ in the same order.*

Proof. Let's assume two correct validators v and v' execute in different orders TxCert and TxCert′ taking the same input object Obj. That is, v executes TxCert then TxCert′, and v' executes TxCert′ then TxCert. We argue this lemma by contradiction of Lemma 4. Check (4.2) of Algorithm 4 ensures that a correct validator only executes certificates by following the sequence of monotonically increasing version numbers (i.e., Lamport timestamps Line 33 of Algorithm 4). As a result, since v executes TxCert before TxCert′, it follows that *version*(Obj) of TxCert is strictly lower than *version*(Obj) of TxCert′. Similarly, since v' executes TxCert′ before TxCert it follows that *version*(Obj) of TxCert′ is strictly lower than *version*(Obj) of TxCert. This implies that both TxCert and TxCert take as input Obj with the same version number. We finally note that correct validators only execute certificates valid for the current epoch (check (4.1) of Algorithm 4), thus TxCert and TxCert′ share the same epoch number. As a result, TxCert and TxCert share the input Obj for the same version and epoch number and are thus conflicting. This is a direct contradiction of Lemma 4. □

Lemma 7 (Shared Objects Sequential Execution). *If two certificates TxCert and TxCert′ both take as input the same shared object* Obj*, all correct validators execute TxCert and TxCert′ in the same order.*

Proof. Let's assume two correct validators v and v' execute in different orders TxCert and TxCert′ taking the same input shared object Obj. That is, v executes TxCert then TxCert′, and v' executes TxCert′ then TxCert. We note $\mathsf{TxDigest} = \mathit{digest}(\mathsf{TxCert})$, $\mathsf{TxDigest}' = \mathit{digest}(\mathsf{TxCert}')$, and call ObjID the identifier of Obj. We argue this lemma by contradiction of Lemma 5. First, we note that Lamport timestamps (see Line 33 of Algorithm 4) ensure correct validators always assign strictly increasing version numbers to shared objects. Since v executes TxCert before TxCert′, its SharedLock store holds the following two entries:

$$\begin{aligned}\mathsf{SharedLock}[(\mathsf{TxDigest}, \mathsf{ObjID})] &= \mathsf{Version}\\ \mathsf{SharedLock}[(\mathsf{TxDigest}', \mathsf{ObjID})] &= \mathsf{Version}'\end{aligned}$$

with $\mathsf{Version} < \mathsf{Version}'$. Similarly, since v' executes TxCert′ before TxCert, its SharedLock store holds the following two entries:

$$\begin{aligned}\mathsf{SharedLock}[(\mathsf{TxDigest}', \mathsf{ObjID})] &= \mathsf{Version}'\\ \mathsf{SharedLock}[(\mathsf{TxDigest}, \mathsf{ObjID})] &= \mathsf{Version}\end{aligned}$$

with $\mathsf{Version}' < \mathsf{Version}$. This however means that the stores of v and v' conflict, which is a direct contradiction of Lemma 5. □

State transistions. We finally show that correct authorities executing the same sequence of certified transitions end up in the same state.

Lemma 8 (Objects Identifiers Uniqueness). *No polynomial-time constrained adversary can create two objects with the same identifier ObjID without two successful invocations of exec(TxCert) over the same certificate TxCert.*

Proof. We argue this lemma by the construction of the object identifier ObjID. Section 3.1 derives each objects identifier ObjID by hashing the digest *digest*(TxCert) of the certificate creating the object along with an index unique to each input object of TxCert. The adversary thus needs to find a hash collision to generate the same ObjID twice through the invocation of *exec*(TxCert) and *exec*(TxCert′), where TxCert $\neq$ TxCert′. □

Lemma 9 (Deterministic Execution)**.** *Every correct validator executing the same set of certificates makes the same state transitions.*

Proof. Every certificate TxCert in the sequence is executed by calling the function PROCESSCERT of Algorithm 4. The function *exec*(TxCert) (Line 17) calls the Move VM to produce the set of the newly created or mutated objects $[\mathsf{Obj}_{out}]$. The determinism of the Move VM and the correctness of its type checker ensures that every correct validator calling *exec*(TxCert) with the same input TxCert produces the same $[\mathsf{Obj}_{out}]$. ATOMICPERSIST (Line 18) then persists the state transition atomically (preventing crash-recoveries from corrupting the state). □

SUI LUTRIS safety. We now prove the safety of SUI LUTRIS using the previous lemmas.

Theorem 2 (SUI LUTRIS Safety)**.** *At the start of every epoch, all correct validators have the same state.*

Proof. We argue this property by induction. Assuming a history of $n-1$ epochs for which this property holds we consider epoch n. Lemma 2 and Lemma 3 prove that all correct validators will execute the same set of transactions by the start of epoch $n+1$. Then Lemma 6 and Lemma 7 show that correct validators can only execute those transactions in the same order (whenever order matter). Finally, Lemma 9 shows that the execution of those transactions causes the same state transition across all correct validators. As a result, every correct validator will have the same state at the start of epoch $n+1$. The inductive base case is argued by construction: all correct validators start in the same state during the first epoch (i.e., genesis). □

Client-perceived safety. Clients consider a transaction Tx final if there exists an effect certificate EffCert over Tx. We thus show that the existence of EffCert implies that Tx is never reverted (Lemma 15 shows that Tx will eventually be executed). Thus all final transactions will be in a checkpoint within the epoch.

Theorem 3 (Client-Perceived Safety)**.** *If there exists an effect certificate EffCert over a transaction Tx, the execution of Tx is never reverted.*

Proof. Let's assume there exists an effect certificate EffCert over a transaction Tx and that the execution of Tx is reverted. The execution of Tx is reverted if and only if Tx is not included in a checkpoint by the end of the epoch. However, correct validators only sign EffCert after including Tx in the list of certificates to be sequenced and eventually observed into a checkpoint c. By the liveness property of consensus within an epoch, a correct validator will eventually be able to sequence the certificate as long as the epoch is ongoing. The epoch ending before the certificate being sequenced and included in a checkpoint implies that a set of $f+1$ correct validators signed EffCert and did not see it included in a checkpoint within the epoch, and a disjoint set of $f+1$ correct validators did not sign EffCert and participated in the reconfiguration protocol to to move to the next epoch. This implies a total of $f+1+f+1+f=3f+2>3f+1$ validators, hence a contradiction. □

Theorem 4 (No Conflicts)**.** *No two conflicting effect certificates exist. That is, two different effect certificates sharing the same input objects and object version.*

Proof. Let's assume two conflicting effect certificates EffCert and EffCert′ exist. We distinguish two (exhaustive) cases, (i) EffCert and EffCert′ share an input owned object with the same version number, and (ii) EffCert and EffCert′ share an input shared object with the same version number. Case (i) implies there must exist two conflicting certificates TxCert and TxCert (remember effect certificate are created by signing certificates). This is however a direct contradiction of Lemma 4. Case (ii) implies that the $f+1$ correct validators who signed EffCert persisted $\mathsf{SharedLock}[(\mathsf{TxDigest}, \mathsf{ObjID})] = \mathsf{Version}$, and the $f+1$ correct validators who signed EffCert′ persisted $\mathsf{SharedLock}[(\mathsf{TxDigest}, \mathsf{ObjID}')] = \mathsf{Version}$. Since Lemma 5 ensures the shared lock store of correct validators do not conflict, the set of $f+1$ correct validators who signed TxCert is disjoint from the set of $f+1$ correct validators who signed TxCert′. As a result, there must be a total of $f+1+f+1+f=3f+2>3f+1$ validators, hence a contradiction. □

A.3. Liveness

We prove the liveness of the SUI LUTRIS protocol. We start by showing that correct users can always obtain a certificate over their valid transactions, even across epochs.

Lemma 10 (Dependencies Availability)**.** *Given a certificate TxCert a correct user can always retrieve all the dependencies (i.e. parents) of TxCert.*

Proof. We argue this property by induction on the serialized retrieval of the direct parent certificates. Assuming a history of $n+1$ certificate dependencies for which this property holds, we consider certificate n noted TxCert. TxCert is signed by $2f+1$ validators, out of which at least $f+1$ are correct. Correct validators only sign a transaction after ensuring they hold all its input objects (check (1.1) of Algorithm 1). This means that $f+1$ correct validators have executed (and persisted) certificate $n-1$ that created the inputs of TxCert. A correct user can thus query any of those $f+1$ correct validators for certificate $n-1$. The inductive base case assumes that the first dependency of every certificate is a fixed genesis (which we ensure axiomatically). □

Lemma 11 (Certificate Creation)**.** *A correct user can obtain a certificate* TxCert *over a valid transaction* Tx.

Proof. A correct validator always signs a transaction Tx if it passes all 4 checks of Algorithm 1. Lemma 10 proves that a correct user can always ensure check (1.1) passes by providing all the transaction's dependencies the validator missed. Correct transactions always pass check (1.2). Check (1.3) always passes for the first copy of Tx received by the validator (at any given time). Finally correct users do not equivocate. Thus Tx is the first and only transaction referencing its owned object, and always passes check (1.4). As a result, if Tx is disseminated to $2f+1$ correct validators by a correct user, they will eventually all return a signature TxSign to the user. The user then aggregates those TxSign into a certificate TxCert over Tx. □

Lemma 12 (Certificate Renewal)**.** *A correct user holding a certificate over transaction* Tx *for an old epoch* e *that did not finalize in* e *can get a new certificate over* Tx *for the current epoch* e'.

Proof. A correct user holding a certificate over transaction Tx for an old epoch e can re-submit Tx to $2f+1$ correct validators to obtain a new certificate for the current epoch e'. Indeed, correct validators sign Tx if it passes all 4 checks of Algorithm 1 like in Lemma 11. If the validator did not already execute Tx, the correct user can ensure check (1.1) passes by providing all the transaction's dependencies the validator missed (Lemma 10). Check (1.2) and (1.3) will pass exactly as described in Lemma 11. Finally, check (1.4) passes since correct users do not attempt equivocation during epoch e' and correct validators drop the store $\mathsf{OwnedLock}$ upon epoch change (and thus $\mathsf{OwnedLock}[\mathsf{ObjKey}] ==$ None, for every input of Tx). As a result, if Tx is disseminated to $2f+1$ correct validators by a correct user, they will eventually all return a signature TxSign to the user. The user then aggregates those TxSign into a certificate TxCert over Tx. □

The existence of a certificate implies that every owned object used as input of a certified transaction is locked for a particular version number. We now prove that all the shared objects of the certificate are also eventually locked for a version number.

Lemma 13 (Shared Locks Availability)**.** *A correct user can always ensure that all correct validators eventually assign shared locks to all shared objects of a valid transaction* Tx.

Proof. Lemma 11 shows that a correct user can always assemble a certificate TxCert over a valid transaction Tx. The correct user can then forward the TxCert to an honest authority who submits it to the consensus engine. By the liveness property of the consensus, TxCert is eventually sequenced by all correct validators. When TxCert is sequenced, correct validators call ASSIGNSHAREDLOCKS (Algorithm 4). Line 36 of Algorithm 4 then assigns locks to all shared objects of TxCert. □

We finally show that the existence of a certificate ensures the transactions of a correct user are eventually included in a checkpoint, and thus eventually executed.

Lemma 14 (Effect Certificates Availability)**.** *A correct user can always ensure an effect certificate* $\mathsf{EffCert}$ *over transaction* Tx *will eventually exist if a certificate* TxCert *over* Tx *exists.*

Proof. A correct validator signs an effect $\mathsf{EffSign}$ if it passes all 3 checks of function PROCESSCERT of Algorithm 4. A correct user can ensure that check (4.1) passes by either providing the validator with the certificate TxCert during the same epoch of its creation or by re-creating a certificate for the current epoch (Lemma 12). A correct user can ensure check (4.2) passes by providing all the certificate's dependencies the validator missed (Lemma 10). Lemma 13 ensures that a correct user can make correct validators assign shared locks to all shared objects of a certificate TxCert, thus validating the first part of check (4.3) Line 10. It can then ensure the second part of check (4.3) Line 14 succeeds by providing the validator with all dependencies it missed. As a result, a correct user can collect at least $2f+1$ effects $\mathsf{EffSign}$ over Tx and assemble them into an effect certificate $\mathsf{EffCert}$. □

Lemma 15 (Checkpoint Inclusion)**.** *If an effect certificate over transaction* Tx *exists within an epoch,* Tx *will be included in a checkpoint within the same epoch.*

Proof. If an effect certificate $\mathsf{EffCert}$ exists, at least $f+1$ correct validators executed its corresponding transaction Tx. When correct validators execute a transaction they include it in the list of certificates to sequence and checkpoint (Section 4.1). Since $f+1$ correct validators are also needed to close the epoch, and a correct validator will not do so until it witnesses all listed certificates being sequenced, and by the liveness of consensus within an epoch, it follows that eventually the certificate will be sequenced (similar to Theorem 3). Since all certificates on which a certificate depends must also have been executed (in case of owned objects) or sequenced and executed (in case of shared objects) before an honest validator executes the transaction and signs it, it follows that if an $\mathsf{EffCert}$ exists then an $\mathsf{EffCert}$ for all dependencies will also exist and also be eventually sequenced. Since the certificate and all its causal dependencies will eventually be sequenced within the epoch, they will be included in a checkpoint within the epoch. □

Lemma 16 (Checkpoint Execution)**.** *All correct validators eventually execute all transactions included in all checkpoints.*

Proof. Correct validators assemble checkpoints out of certificates sequenced by consensus. By the liveness property of the consensus protocol, all correct validators can eventually sequence all the certificates they are executed (or observe others do so) and assemble them into checkpoints. We conclude the proof by noting that correct validators execute all transactions within all checkpoints they assemble. □

Theorem 5 (SUI LUTRIS Liveness). *A correct user can always ensure its transaction* Tx *will eventually be finalized. That is, all correct validators execute it and never revert it.*

Proof. Lemma 11 ensures that a correct user can eventually obtain a certificate TxCert over their valid transaction Tx. Lemma 14 then ensures the user can get an effect certificate $\mathsf{EffCert}$ using TxCert. Lemma 15 shows that the existence of $\mathsf{EffCert}$ implies the transaction is eventually included in the a checkpoint. Finally Lemma 16 shows that all transactions included in all checkpoints are executed. To conclude the proof we note that the execution of transactions included in checkpoints is never reverted (Section 4.3). □

Theorem 6 (Client-Perceived Starvation Freedom). *Let's assume two correct validators respectively set* $\mathsf{OwnedLock}[\mathsf{ObjKey}] = sign(\mathsf{Tx}_1)$ *and* $\mathsf{OwnedLock}[\mathsf{ObjKey}] = sign(\mathsf{Tx}_2)$ *(with* $\mathsf{Tx}_1 \neq \mathsf{Tx}_2$*) during epoch* e*. A correct user can eventually obtain an effect certificate over transaction* Tx' *accessing* ObjKey *at epoch* $e' > e$*.*

Proof. All owned objects locked by transactions at epoch e are freed upon entering epoch $e+1$ (see Section 4.2); that is, correct validators drop all $\mathsf{OwnedLock}[\cdot]$ upon epoch change. Correct validators thus sign a correct transaction Tx' accessing ObjKey at epoch $e' > e$ submitted by correct clients (who do not equivocate). Lemma 11 then ensures the client eventually obtains a certificate over Tx' and Lemma 14 ensures the client eventually obtains an effect certificate over Tx'. □

Appendix B. Reproducing Experiments

We provide the orchestration scripts[15] used to benchmark on AWS the codebase evaluated in this paper.

Deploying a testbed. The file ' /.aws/credentials' should have the following content:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

configured with account-specific AWS *access key id* and *secret access key*. It is advise to not specify any AWS region as the orchestration scripts need to handle multiple regions programmatically.

A file 'settings.json' contains all the configuration parameters for the testbed deployment. We run the experiments of Section 6 with the following settings:

```
{
    "testbed_id": "${USER}-sui",
    "cloud_provider": "aws",
    "token_file": "/Users/${USER}/.aws/credentials
        ",
    "ssh_private_key_file": "/Users/${USER}/.ssh/
        aws",
    "regions": [
        "us-east-1",
        "us-west-2",
        "ca-central-1",
        "eu-central-1",
        "ap-northeast-1",
        "ap-northeast-2",
        "eu-west-1",
        "eu-west-2",
        "eu-west-3",
        "eu-north-1",
        "ap-south-1",
        "ap-southeast-1",
        "ap-southeast-2"
    ],
    "specs": "m5d.8xlarge",
    "repository": {
        "url": "https://github.com/mystenlabs/sui.
            git",
        "commit": "sui-lutris"
    }
}
```

15. https://raw.githubusercontent.com/asonnino/sui/sui-lutris/crates/orchestrator

where the file '/Users/$USER/.ssh/aws' holds the ssh private key used to access the instances.

The orchestrator binary provides various functionalities for creating, starting, stopping, and destroying instances. For instance, the following command to boots 2 instances per region (if the settings file specifies 13 regions, as shown in the example above, a total of 26 instances will be created):

```
cargo run --bin orchestrator -- \
    testbed deploy --instances 2
```

The following command displays he current status of the testbed instances

```
cargo run --bin orchestrator testbed status
```

Instances listed with a green number are available and ready for use and instances listed with a red number are stopped. It is necessary to boot at least one instance per load generator, one instance per validator, and one additional instance for monitoring purposes (see below). The following commands respectively start and stop instances:

```
cargo run --bin orchestrator -- testbed start
cargo run --bin orchestrator -- testbed stop
```

It is advised to always stop machines when unused to avoid incurring in unnecessary costs.

Running Benchmarks. Running benchmarks involves installing the specified version of the codebase on all remote machines and running one validator and one load generator per instance. For example, the following command benchmarks a committee of 100 validators (non faulty) under a constant load of 1,000 tx/s for 10 minutes (default), using 3 load generators:

```
cargo run --bin orchestrator -- benchmark \
    --committee 100 fixed-load --loads 1000 \
    --dedicated-clients 3 --faults 0
    --benchmark-type 0
```

The parameter 'benchmark-type' is typically set to "0" to instruct the load generators to submit individual payment transactions. It can also be set to "batch" to instruct them to submit bundles of 100 payment transactions, as experimented in Figure 3. When benchmarking individual transactions, we select the number of load generators by ensuring that each individual load generator produces no more than 350 tx/s (as they may quickly become the bottleneck). We set the number of load generators to 40 when benchmarking bundles of 100 transactions (Figure 3).

Monitoring. The orchestrator provides facilities to monitor metrics. It deploys a Prometheus instance and a Grafana instance on a dedicated remote machine. Grafana is then available on the address printed on stdout when running benchmarks with the default username and password both set to `admin`. An example Grafana dashboard can be found in the file 'grafana-dashboard.json'[16].

Troubleshooting. The main cause of troubles comes from the genesis. Prior to the benchmark phase, each load generator creates a large number of gas object later used to pay for the benchmark transactions. This operation may fail if there are not enough genesis gas objects to subdivide or if the total system gas limit is exceeded. As a result, it may be helpful to increase the number of genesis gas objects per validator in the 'genesis_config' file[17] when running with very small committee sizes (such as 10).

16. https://github.com/asonnino/sui/blob/sui-lutris/crates/orchestrator/assets/grafana-dashboard.json
17. https://github.com/asonnino/sui/blob/7f3d922432b185e6977513ea577929ea06097102/crates/sui-swarm-config/src/genesis_config.rs#L361