Cuttlefish: Expressive Fast Path Blockchains with FastUnlock

Lefteris Kokoris-Kogias[1,2], Alberto Sonnino[1,3], and George Danezis[1,3]

[1] MystenLabs
[2] IST Austria
[3] University College London

Abstract. Cuttlefish addresses several limitations of existing consensus-less and consensus-minimized decentralized ledgers, including restricted programmability and the risk of deadlocked assets. The key insight of Cuttlefish is that consensus in blockchains is necessary due to contention, rather than multiple owners of an asset as suggested by prior work. Previous proposals proactively use consensus to prevent contention from blocking assets, taking a pessimistic approach. In contrast, Cuttlefish introduces collective objects and multi-owner transactions that can offer most of the functionality of classic blockchains when objects transacted on are not under contention. Additionally, in case of contention, Cuttlefish proposes a novel 'Unlock' protocol that significantly reduces the latency of unblocking contented objects. By leveraging these features, Cuttlefish implements consensus-less protocols for a broader range of transactions, including asset swaps and multi-signature transactions, which were previously believed to require consensus.

1 Introduction

Consensus is not required for implementing decentralized asset transfers [13]. This insight led to the design of cryptocurrencies based on consistent or reliable broadcast [2,3,8], which offer several advantages. They exhibit exceptionally low latency, operate purely asynchronously, and are highly scalable.

However, consensus-less systems suffer from two significant limitations. Firstly, they have limited programmability, since to maintain liveness transactions must be submitted in a valid and race-condition-free manner. Failure to do so can result in deadlocked assets that become forever inaccessible to their owners. Thus, programmability is restricted to simple transactions involving objects owned by a single entity, such as asset transfers or payments. Attempts to support more complex transactions involving multiple users (e.g., asset swaps) or authorization (e.g., multi-signature) risk causing deadlocks, rendering assets unusable indefinitely. As a result, existing consensus-less cryptocurrencies [2,3,8] are only suited for basic operations.

The second limitation arises from the strong requirement imposed on clients in consensus-less systems to never issue conflicting transactions. Even minor bugs in client implementations can lead to deadlocked assets. For instance, a faulty wallet that unintentionally sends a transaction with a randomized signature twice may be interpreted as two conflicting transactions, resulting in locked assets. Similar issues occur when a client underestimates the required gas for a transaction and attempts to reissue it with a higher gas amount.

To address these limitations, the recent Sui blockchain [4], introduces a hybrid model that combines a consensus-less *fast path* with a consensus-based fallback. Sui supports general-purpose smart contracts by utilizing broadcast in the fast path, while transactions involving shared objects at risk of race conditions are processed through the consensus fallback path. Consensus is also employed for a daily system reconfiguration which drops all the locks of the fast path and thus implicitly solves potential deadlocks. While Sui theoretically overcomes both limitations, it does so at the cost of increased latency and reduced usability. Accessing shared state necessitates sequencing transactions through consensus in all scenarios, resulting in higher latency ranging from seconds instead of fractions of a second, and prohibiting the use of parallel broadcast protocols for scalability. Furthermore, deadlocked objects are only reset and made available once a day upon reconfiguration, which proves impractical for objects that may legitimately be used by multiple non-coordinating users and get accidentally locked. The fear of losing access to resources for an entire day terrifies developers as it happens on a daily basis due to honest mistakes. As a result, developers on Sui forfeit the use of the low-latency path of Sui and fallback on implementing their smart contract using consensus [22].

This paper introduces Cuttlefish, a solution that addresses both challenges mentioned above. Extending the high-level design of Sui, Cuttlefish combines a fast path reliable broadcast with a consensus fallback while enabling the execution of consensus-less transactions, without introducing additional latency unless there is actual contention. The first contribution of Cuttlefish is the enhancement of fast path object programmability. It achieves this by introducing collective objects, which allow for complex access control and enable the execution of any type of transaction on the fast path. The transaction model is extended to support multi-owner transactions on the fast path, thereby expanding the range of programmable actions beyond single-owner operations. The second and core contribution of Cuttlefish is the design of FastUnlock, a mechanism that facilitates the concurrent execution of conflicting transactions on the fast path, ensuring liveness and enabling rapid unlocking of fast path objects. This advancement makes Cuttlefish suitable for a broader spectrum of transactions, including asset swaps and multi-signature transactions. Cuttlefish is currently considered for adoption by the Sui blockchain team.

2 Motivating Applications of Cuttlefish

Cuttlefish addresses real-world needs raised by existing blockchains.

Deadlocks due to multiple or buggy clients. Cuttlefish tackles the common challenge of locked objects in consensus-less blockchains, that leads to a poor user experience. Using multiple wallets for the same account or objects can result in concurrent conflicting transactions due to bugs, lack of synchronization, or being offline. Even, a single wallet may send a transaction with insufficient gas, only to later attempt to rectify the mistake by updating the gas value, and leading to two conflicting transaction on the same account or objects. Unfortunately, these innocent slip-ups or bugs are interpreted as equivocation attempts within the context of consistent broadcasts, potentially deadlocking the assets involved. Unlike previous solutions, Cuttlefish enabled users to swiftly regain control of their assets through FastUnlock and retry their transactions safely.

Atomic swaps. Atomic swaps allow two parties to exchange digital assets without the need for a trusted intermediary. While consensus-based blockchains can achieve this through smart contracts, the risk of deadlock arises in consensus-less environments due to the possibility of a Byzantine user issuing a concurrent transaction. Such a situation would effectively deadlock the assets of both parties. However, this risk only materializes when an active attacker intentionally causes contention. In rare cases like these, the FastUnlock protocol enables participants in the swap to quickly recover their assets. This safety net allows Cuttlefish to support multi-owner transactions in the fast path, allowing fast path atomic swaps and other multi party smart contracts, enhancing the programmability of consensus-less transactions.

Regulated stablecoins. Regulated stablecoins, require the issuer to be able to block accounts or balances for regulatory reasons, besides their owner spending them, which eludes consensus-less systems. Since multiple parties need to operate on such objects they need to use consensus to sequence these potentially conflicting operations, even though the issue nearly never execises their ability to block objects (creating no practical contention). Cuttlefish allows for collective objects, that may be used by more than one owner, or any pattern or complex access control, and can be used in the fast path.

3 Background

A number of consensus-less systems have been proposed in the literature, including FastPay [2], Astro [8], Zef [3], and Linera [18]. We will specifically describe and extend Sui (the Sui Lutris mechanism [17]) as a basis for the Cuttlefish design, as it is the only currently deployed mechanism with a consensus-less fast path. Cuttlefish extends the expresivity of both object authentication and transactions in the Sui fast path, and also extends that Sui consensus path to support FastUnlock.

Object Types. All Sui blockchain state is composed on a set of objects. There are three types of objects, and their use in a transaction determines whether the fast path or the consensus path is to be used.

– *Read-only objects* cannot be mutated or deleted and may be used in any type of transactions concurrently and by all users.

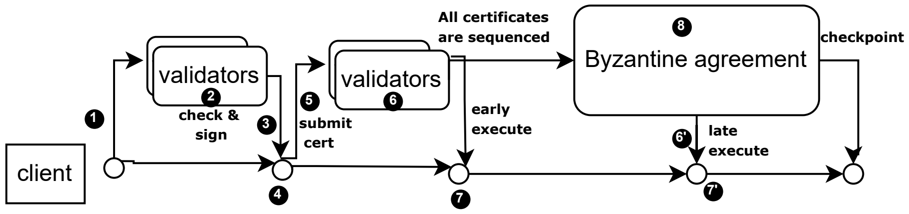

Fig. 1. General protocol flow of Sui Lutris [4] fast-path & consensus failover system.

- *Owned objects* have an owner field that determines access control. When owner is an address representing a public key, a transaction may access the object, if it is signed by that address (which can also be a multi-signature). When the owner of an object (called a child object) is another object ID (called the parent object), the child object may only be used if the root object (the first one in a tree of possibly many parents) is included in the transaction and authorized. This facility is used to construct efficient collections.
- *Shared objects* do not specify an owner. They can instead be included in transactions by anyone, and do not require any authorization. Instead, they perform their authorization logic (enforced by the smart contract).

Transactions. A transaction is a signed command that specifies several input objects, a version number per object, and a set of parameters. If valid it consumes the input object versions and constructs a set of output objects at a fresh version—which can be the same objects at a later version or new objects. Owned objects versions need to be the latest versions in validator databases, and not be re-used across transactions. Shared objects need not specify a version, and the version on which the transaction is executed is assigned by the system. A transaction is signed by a single address and therefore can use one or more objects owned by that address. A single transaction cannot use objects owned by more than one address.

Certificates. A *certificate* (Cert) on a transaction contains the transaction itself as well as the identifiers and signatures from a quorum of at least $2f + 1$ validators. A certificate may not be unique, and the same logical certificate may be signed by a different quorum of validators. 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 (i.e., accountable signatures [5]) to identify validators ready to process the certificate, or that can serve past information required to process the certificate.

Processing in the Fast Path and Consensus. Figure 1 provide an overview of Sui-Lutris and by extension Cuttlefish's common-case. A transaction is sent by a user to all validators (❶), that ensure it is correctly signed for all owned objects and versions, and also that all objects exist (❷); a correct validator rejects any conflicting transaction using the same owned object versions, in the same epoch (so the first transaction using an object acquires a *lock* on it). They then countersign it (❸) and returns the signature to the user. A quorum of signatures constitutes a *certificate* for the transaction (❹). Anyone may submit the certificate to the validators (❺) that check it.

At this point execution may take the fast path: if the certificate only references read-only and owned objects it is executed immediately (❻) and a signature on the effects of the execution returned to the user to create an effects certificate (❼) and the transaction is final. If any shared objects are included execution must wait. In all cases, certificates are input into consensus and sequenced (❽). Once sequenced, the system assigns a common version number to shared objects for each certificate, and execution can resume (steps ❻' and ❼') to finalize the transaction. The common sequence of certificates is also used to construct checkpoints, which are guaranteed to include all finalized transactions (❽).

Checkpoints and Reconfiguration. Sui ensures transaction finality either before consensus for owned object transactions (❼) and after consensus for shared object transactions (❼'). Its reconfiguration protocols ensure that if a transaction could have been finalized it will eventually be included in a checkpoint before the end of the epoch. At the end of the epoch, all locks are reset (❷). Appendix B summarizes the reconfiguration protocol of Sui that Cuttlefish directly adopts.

Limitations of Sui. Misconfigured clients may create and submit concurrently conflicting transactions in step (❶) and (❷), that reuse the same owned object versions. In that case, neither transaction may be able to construct a certificate (❹), and the owned object becomes locked until the end of the epoch. Due to the risk that owned objects can become locked through conflicting transactions, Sui restricts transactions to only contain objects from a single owner, thus limiting the applicability of the fast path—to avoid mistrusting users from locking each others' objects for a day. For similar reasons, objects may also have at most one owner. Cuttlefish addresses all these limitations.

4 Overview

Cuttlefish adopts the high-level design of Sui, namely using reliable broadcast for a fast path with a fall-back to consensus (see Appendix A for definitions of distributed systems primitives), but augments it in the following ways.

1. It allows for *multi-owner transactions* on the fast path that use objects with different 'owners'. In Sui this can only be expressed with shared objects in transactions using the higher-latency consensus path.
2. It introduces *collective objects* that allow for complex authorization involving different users or combinations of users, or even time or external events. Collective objects extend owned objects and may be used on the fast path.

3. It adds a *FastUnlock protocol* that allows for fast path objects blocked due to concurrent conflicting transactions to recover liveness within seconds, whereas Sui would recover only within a day.

Collective objects and multi-owner transactions allow for more expressive transactions in the fast path, but risk increasing the incidence of conflicting transactions and locked owned objects. To alleviate this issue, Section 6 presents a simple design for FastUnlock: it performs a no-op on locked objects using the consensus path making them available again within seconds. Section 7 extends the FastUnlock protocol to force a specific transaction instead of a no-op, which is necessary when objects are under continuous contention.

FastUnlock leverages the consensus protocol to signal that an owned object is suspected of being under contention and should not be processed by the fast path. Following invocations, the current version of the object is blocked, and consensus is used to determine whether a transaction on it might have been final; if not, in the simple FastUnlock the version of the object is increased with a no-op. As a result, the object has a new version that can be accessed via the fast path once again. Since the version is always updated, the transactions that blocked the object are no longer valid, removing any replay attack opportunity. This is not true in Sui as even after the end of the epoch a malicious client can resubmit the equivocated transactions and try to re-lock the object.

Threat Model. Cuttlefish operates in the same threat model as Sui. It assumes a message-passing system with a set of n validators and a computationally bound adversary that controls the network and can 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. Similarly to Sui [17], Cuttlefish additionally uses a consensus protocol as a black box that takes some valid inputs and outputs a total ordering [9,11,21], possibly operating within a partially synchronous model [10].

5 Enhancing Programmability

Cuttlefish provides greater objects programmability on the fast path than existing consensus-less systems using two main ingredients: (i) multi-owner transactions, and (ii) collective objects.

Multi-owner transactions. Sui Lutris requires all owned objects in a transaction to be 'owned' by the same address [17]. Cuttlefish lifts this restriction: a transaction can reference owned objects with any root authenticator term. The transaction contains the authentication evidence used to authorize all objects, such as a set of signatures over the transaction, potentially from multiple addresses.

Validators must ensure that all owned objects referenced by a transactions are correctly authorized before signing a transaction, which ensures that a valid certificate represents an authorized transaction. Transactions that only contain owned objects, even when they have different owners, can be executed on the fast path. Then in addition they contain shared objects their execution needs to be deferred after the certificate has been sequenced by consensus.

Multi-owner transactions make Cuttlefish more susceptible to owned-objects being locked through error or malicious behaviour. For example, consider an atomic swap T transaction that takes objects A owned by Alice and object B owned by Bob and exchanges their ownership. If Alice signs T first, Bob may refuse to sign initially denying Alice access to her object. If Alice loses patience and tries to use A in another transaction T', then Bob can sign T and race Alice's attempt to build a certificate. Now both T and T' contain A and conflict which can lead to a locked A (and B). To resolve such situations it is necessary for Cuttlefish to implement FastUnlock described in Section 6.

Collective Objects. Collective objects are owned objects with a more complex authenticator, than the usual address or object ID that Sui Lutris supports. Complex authenticators allow conjunction, disjunction, and weighted thresholds thresholds of authentication terms to be used as authenticators. Authentication terms include the traditional address and object ID, but also conditions on time or events that have occured in the environment of the execution. Due to the fact that multiple non-coordinating or even mutually distrustful parties can use the object in transaction, as well as the fact that some authorization terms are non-deterministic, complex authenticators can lead to conflicting transactions being authorized on objects and thus require the FastUnlock protocols to be practical.

More specifically Cuttlefish extends the authorization logic of an owned object to be a root authentication term $\langle T \rangle$ from the grammar in Figure 2:

$$\begin{aligned}
\langle T \rangle &:= \textsc{PublicKey}(pk) \mid \textsc{ObjectID}(oid) \\
&:= \textsc{BeforeTime}(t) \mid \textsc{AfterTime}(t) \\
&:= \textsc{EventOccured}(c, e) \\
&:= \textsc{Threshold}(W, [(w_i, \langle T \rangle_i)]) \\
&:= \bigwedge_i \langle T \rangle_i \qquad (\textsc{And}) \\
&:= \bigvee_i \langle T \rangle_i \qquad (\textsc{Or})
\end{aligned}$$

Fig. 2. Grammar defining the authrorization logic for collective objects

- A PublicKey term is true if the transaction is signed by the public key *pk*. Using a single such term as an authenticator for an object expresses the authentication logic of a traditional single owner object in Sui.
- A ObjectID term requires the object with id *oid* to be included (and authenticated) as part of the transaction. A single object id authenticator expresses the traditional parent-child relation, and ownership rules in Sui.
- The BeforeTime and AfterTime are true if the (local) time the transaction is received by the validator is respectively before or after t. Note that since even honest validators cannot have perfectly synchronized clocks, it is possible that a transaction with such a term becomes 'stuck'.
- The EventOccured term becomes true if in the trace of finalized executions a specific event was emitted on chain c. Note that the chain may be different chain than the one operated by Cuttlefish effectively making authorization conditional on an oracle for another chain. Such an event may be described by type or content and we abstract this in e. A reference to the transaction that emitted the event can be provided as an authenticator to help validators check this term.
- The Threshold defined a threshold W and a weight w_i for a set of terms. It is true if the sum of weights of the true terms exceed the threshold. It allows the definition of flexible policies such as requiring a threshold of signature or other conditions to be present to authorize the object being used.
- The And and Or define a number of terms, and are true if all or any of these terms are true, respectively.

A transaction needs to provide evidence that all authenticator terms for all objects in its input set are true. For each input object it specifies the path(s) in the authentication term tree that are true supporting the overall authenticator term, collectively called *authentication paths*. It also contains a set of signatures (as a list ordered by public key) signing the transaction. To allow for greater flexibility the authentication paths are not signed (conceptually they are part of the signature not the transaction), and therefore a transaction cannot get information about the logic that authorized its execution through this mechanism.

We note that an authorization path may be expressed in a very succinct manner as a one bit per Threshold, And or Or branch pursued to demonstrated the root authenticator term to be true. A single signature is required to satisfy any number of PublicKey terms with the same *pk*. ObjectID terms can be demonstrated as satisfied implicitly by including the *oid* as an input. EventOccured, BeforeTime and AfterTime terms are satisfied (or not) through the validator comparing their specified time with the current time or consulting a chain for an event, and incur no additional overhead in terms of evidence in the transaction.

We represent the authenticator logic as a tree, with AND/OR, k-out-of-n connectives as branches and identities, time conditions and object IDs as leafs. In this representation, we can augment each branch and leaf with an optional nonce, compute a Merkle tree over them, and only store a hash of the root as the authenticator. In this way transfering to a complex authenticator is no different than transferring an object to an address, and one cannot tell the difference until the object is used in a transaction. A transaction then reveals only the paths necessary to show that the condition for access is satisfied. This allows objects to preserve secret authenticators until they are accessed, and even upon access only reveal the information required. We leave using zero-knowledge proofs as evidence all authenticators are satisfied in a transaction, allowing us to hide all information besides authorization, for future work.

6 Baseline FastUnlock Protocol

Both multi-owner transactions and collective objects can result in deadlocks in the fast path when correct clients attempt to access objects concurrently. To remedy this issue Cuttlefish introduces a FastUnlock functionality. For simplicity, we show how to unlock a single object by either executing a preexisting transaction to finality or executing a no-op which only increases the version. Section 7 extends the basic protocol to execute a new transaction instead of a no-op. Sui [17] provides detailed specifications and implementations of its system model and Cuttlefish largely extends it with the additional FastUnlock protocol.

New Persistent Data Structures. Each Cuttlefish validator maintains a set of persistent tables abstracted as key-value maps, with the usual $\mathsf{contains}$, get, and set operations. The map

$$\textsc{LockDb}[\mathsf{ObjectKey}] \rightarrow \mathsf{Cert} \text{ or } \mathbf{None}$$

maps each object's identifier and version, $\mathsf{ObjectKey} = (\mathsf{ObjectId}, \mathsf{Version})$, to a certificate Cert or **None** if the object's version exist by the validator does not hold any certificate. The map

$$\textsc{UnlockDb}[\mathsf{ObjectKey}] \rightarrow \mathbf{Unlocked}, \mathbf{Confirmed}, \text{ or } \mathbf{None}$$

records whether a transaction over the specified object version is involved in a current FastUnlock instance (**Unlocked**), has been sequenced by the consensus engine (**Confirmed**), or none of the above (**None**).

All new owned object entries start with $\textsc{UnlockDb}[\mathsf{ObjectKey}]$ set to **None**. Once a transaction certificate is sequenced through consensus it is always executed (whether it is for a shared object transaction or an owned object only transaction) and all owned object entries have $\textsc{UnlockDb}[\mathsf{ObjectKey}]$ set to **Confirmed**.

FastUnlock Protocol Description. In order to safely unlock an object, the user interactively constructs a proof, called a *no-commit certificate*, that no transaction modifying that object has been committed or will be committed on the fast path. This proof consists of a message signed by a quorum of validators attesting that they have not already executed a transaction over the $\mathsf{ObjectKey}$, and promising that they will not execute any transaction over the $\mathsf{ObjectKey}$

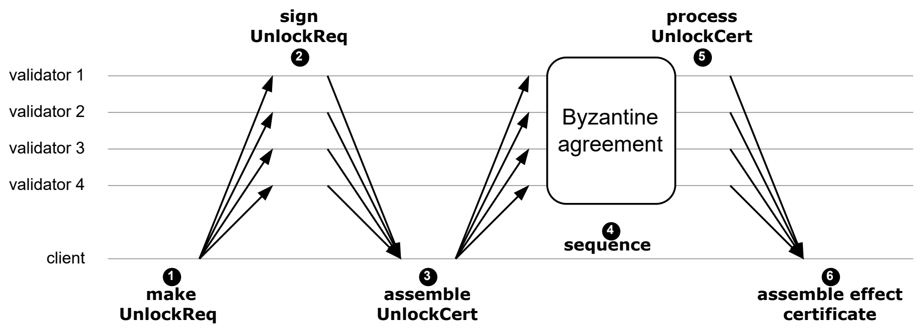

Fig. 3. FastUnlock interactions between a user and validators to unlock an object.

in the fast path. Only certificates sequenced over consensus may affect such an ObjectKey going forward.

Figure 3 illustrates the fast-unlock protocol allowing a user to instruct validators to unlock a specific object. A user first creates an *unlock request* specifying the object they wish to unlock:

$$\mathsf{UnlockRqt}(\mathsf{ObjectKey}, \mathsf{Auth})$$

This message contains the object's key ObjectKey to unlock (accessible as UnlockRqt.ObjectKey) and an authenticator Auth ensuring the user is authorized to unlock ObjectKey. The authenticator is composed of two parts: (i) a transaction that mutates the object in question and potentially additional objects, which is signed by the object owner, and (ii) a proof that the party requesting an unlocking can modify at least one of the objects in the transaction. The authenticator prevents rogue unlock requests for objects that are either not under contention (the transaction shows there exists a transaction that uses the object) or by parties not authorized to act on the objects. The user broadcasts this UnlockRqt message to all validators (❶).

Each validator handles the UnlockRqt as follows (Algorithm 1). A validator first performs the following check:

– **Check (1.1)** It ensures the validity of UnlockRqt by verifying the authenticator Auth with respect to the ObjectKey to unlock. Specifically, it should contain a valid transaction including ObjectKey and evidence that the unlock is authorized given the owner of ObjectKey. Otherwise stop processing.

The validator attempts to retrieve a certificate Cert for a transaction on ObjectKey exists (**Step (1.2)**) or sets Cert to **None**. Then, the validator records that the object in UnlockRqt can only be included in transaction in the consensus path (Line 11) by setting its entry in the UnlockDb[ObjectKey] to **Unlocked** (**Step (1.3)**). It finally returns a signed *unlock vote* UnlockVote to the user:

$$\mathsf{UnlockVote}(\mathsf{UnlockRqt}, \mathbf{Option}(\mathsf{Cert}))$$

This message contains the UnlockRqt itself, the (possibly **None**) certificate Cert leading to the execution of the object key referenced by UnlockRqt (❷).

Algorithm 1 Process unlock requests

```
// Handle UnlockRqt messages from clients.
procedure ProcessUnlockTx(UnlockRqt)
    // Check (1.1): Check Auth. (Section 6).
    if !valid(UnlockRqt) then return error

    // Step (1.2): No conflicting executions.
    ObjectKey ← UnlockRqt.ObjectKey
    Cert ← LockDb[ObjectKey]  ▷ Can be None

    // Step (1.3): Record the decision to unlock.
    UnlockVote ← sign(UnlockRqt, Cert)
    UnlockDb[ObjectKey] ← Unlocked

    return UnlockVote
```

Algorithm 2 Process unlock certificates

```
// Handle UnlockCert message from consensus.
procedure ProcessUnlockCert(UnlockCert)
    // Check (2.1): Check if we can execute (Section 6).
    if UnlockDb[ObjectKey] = Confirmed then
        return

    // Check (2.2): Check cert validity (Section 6).
    if !valid(UnlockCert) then return error

    // Execute Cert or None (2.3)
    Cert ← UnlockCert.Cert
    if Cert ≠ None then
        Tx ← Cert.Tx
    else
        Tx ← No-Op
    EffectSign ← exec(Tx, UnlockCert)

    // Prevent execution overwrite.
    UnlockDb[ObjectKey] ← Confirmed

    return EffectSign
```

The user collects a quorum of $2f+1$ UnlockVote over the same (UnlockRqt, Cert) fields and assembles them into an *unlock certificate* UnlockCert:

$$\mathsf{UnlockCert}(\mathsf{UnlockRqt}, \textbf{Option}(\mathsf{Cert}))$$

where UnlockRqt is the certified abort message created by the user and Cert is the (possibly **None**) certificate leading to the execution of the objects referenced by UnlockRqt. There are two cases leading to the creation of UnlockCert:

1. At least one UnlockVote carries a certificate. This scenario indicates that a correct validator already executed a transaction, which implies the object is not locked. However this is not a proof of finality and subsequent steps may invalidate this execution.
2. No UnlockVote carries a certificate. This scenario is a 'no-commit' proof as there are $f+1$ honest validators that will not process certificates (UnlockDb holds **Unlocked**) thus no certificate execution in the fast path will ever become final.

The user submits this UnlockCert for sequencing by the consensus engine (❸). All correct validators observe a consistent sequence of UnlockCert messages output by consensus (❹) and process them in order as follows (Algorithm 2). A validator performs the following checks, and if any fails they ignore the certificate:

– **Check (2.1)** They ensure they did not already process another transaction to completion (i.e. UnlockDb is not **Confirmed**) or a different UnlockCert for the same objects keys.

– **Check (2.2)** They check UnlockCert is valid, that is, the validator ensures (i) it is correctly signed by a quorum of authorities, and (ii) that the certificate Cert it contains is valid or **None**.

The validator then executes the transaction referenced by Cert (step 2.3) if one exists. Otherwise, if Cert is empty, the validator undoes any local transaction executed on the object[4], then executes a no-op, that is, the object contents remain unchanged but its version number increases by one. The validator finally marks every object key as **Confirmed** to prevent future unlock certificates or checkpoint certificates from overwriting execution (Line 18) and returns an EffectSign to the user (❺). The user assembles a quorum of $2f+1$ EffectSign messages into an *effect certificate* EffectCert that determines finality (❻).

Appendix C details the use of gas objects within the context of FastUnlock and D proves the safety and liveness of the protocol. The key insight is that an UnlockCert forces transactions on the owned object to go through consensus sequencing. There, either a transaction certificate or an unlock certificate will be sequence first and consistently executed. An unlock certificate for a finalized transaction will always result in the execution of the same transaction.

Auto-Unlock. The basic FastUnlock scheme presumes that the request to unlock an object is authenticated by the owner(s) of the object. This ensures that only authorized parties can interfere with the completion of a transaction, but it also restricts who can initiate unlocking in case of loss of liveness. Alternatively, an 'Auto Unlock' scheme may use a synchrony assumption instead to initiate unlock: each validator upon signing a transaction associates with each input object the current timestamp. An Auto Unlock request is identical to a FastUnlock request, but is not authenticated by the object owner. Instead, its validity is checked (checks (1.1) and (2.1)) by ensuring that a sufficient delay Δ has passed since the object was locked. To ensure liveness the delay Δ should be long enough to allow for the creation of transaction certificates if there is no contention. FastUnlock and Auto Unlock can be combined: an authenticated request can be processed immediately, but an unauthenticated request is only valid after Δ.

7 Contention Mitigation

The basic FastUnlock protocol speeds up recovery from loss of liveness due to mistakes. However, Cuttlefish aims to support workloads on the fast path that are truly under contention. In this case, the basic protocol in Section 6 is insufficient, since it can result in multiple rounds of locking and no-op unlocking without any user transaction being committed. We present a protocol that proposes a new transaction during the unlock phase that is executed once the unlock

[4] The UnlockCert with Cert being **None** ensures such an execution could not have been final; only a single layer of execution can ever be undone, and no cascading aborts can happen.

is sequenced, ensuring liveness. Additionally, we show how to generalize the basic protocol to unlock multiple objects at once.

Algorithm 3 Process unlock requests (multi)

```
    // Handle UnlockRqt messages from clients.
 1: procedure ProcessUnlockTx(UnlockRqt)
 2:     // Check (3.1): Check authenticator.
 3:     if !$valid$(UnlockRqt) then return error
 4:
 5:     // Collect certificates.
 6:     $c \leftarrow$ None
 7:     for ObjectKey $\in$ UnlockRqt.ObjectKeys do
 8:         $c \leftarrow c \cup$ LockDb[ObjectKey]
 9:     UnlockVote $\leftarrow$ $sign$(UnlockRqt, $c$)
10:
11:     // Record the decision to unlock.
12:     if $c ==$ None then
13:         for ObjectKey $\in$ UnlockRqt.ObjectKeys do
14:             UnlockDb[ObjectKey] $\leftarrow$ Unlocked
15:
16:     return UnlockVote
```

Algorithm 4 Process unlock certificates (multi)

```
    // Handle UnlockCert messages from consensus.
 1: procedure ProcessUnlockCert(UnlockCert)
 2:     // Check (4.1): Check message validity.
 3:     for ObjectKey $\in$ UnlockCert.ObjectKeys do
 4:         if UnlockDb[ObjectKey] = Confirmed then
 5:             return
 6:
 7:     // Check (4.2): Check message validity.
 8:     if !$valid$(UnlockCert) then return error
 9:
10:     // Check (4.3): Can we execute the tx?
11:     $v \leftarrow [\,]$
12:     if UnlockCert.Cert = [ ] then
13:         Tx $\leftarrow$ UnlockCert.UnlockRqt.Tx
14:         EffectSign $\leftarrow$ $exec$(Tx, UnlockCert)
15:         $v \leftarrow$ EffectSign
16:         for ObjectKey $\in$ UnlockCert.ObjectKeys do
17:             UnlockDb[ObjectKey] = Confirmed
18:     else
19:         for Cert $\in$ UnlockCert.Cert do
20:             EffectSign $\leftarrow$ $exec$(Cert)
21:             $v \leftarrow v \cup$ EffectSign
22:             for ObjectKey $\in$ Cert.ObjectKeys do
23:                 UnlockDb[ObjectKey] = Confirmed
24:     return $v$
```

The multi-objects unlock protocol follows the same general flow as the single-object unlock protocol described in Section 6. We thus describe the protocol referring to the steps ❶-❻ depicted in Figure 3.

Protocol description. The user first creates an *unlock request* specifying a set of objects to unlock:

$$\text{UnlockRqt}([\text{ObjectKey}], \text{Tx}, \text{Auth})$$

This message contains a list of the object's keys [ObjectKey] to unlock (accessible as UnlockRqt.ObjectKeys), a new transaction Tx to execute if the unlock process succeeds, and an authenticator Auth ensuring the sender is authorized to access all objects in [ObjectKey]. The user broadcasts this message to all validators (❶).

Algorithm 3 describes how each validator handles this unlock request UnlockRqt. They first perform Check (3.1) Line 3 to check the authenticator Auth is valid with respect to all objects. This check ensures that the user is authorized to mutate all the objects referenced by UnlockRqt and to lock all owned object referenced by Tx. The validator then collects any certificates for the objects referenced by UnlockRqt (Line 8) and adds them to the response as Cert. The validator then marks object in UnlockRqt as reserved for transaction executed through consensus only (Line 14).

The validator finally returns an *unlock vote* UnlockVote to the user:

$$\text{UnlockVote}(\text{UnlockRqt}, [\textbf{Option}(\text{Cert})])$$

This message contains the unlock message $\mathsf{UnlockRqt}$ itself and possibly a set of certificates [Cert] on transactions including the object keys referenced by $\mathsf{UnlockRqt}$ (possible empty) (❷). If Cert is not empty the certified transactions may have been finalized, and should be executed instead of the new transaction.

The user collects a quorum of $2f + 1$ $\mathsf{UnlockVote}$ over the same $\mathsf{UnlockRqt}$ message and assembles them into an *unlock certificate* $\mathsf{UnlockCert}$:

$$\mathsf{UnlockCert}(\mathsf{UnlockRqt}, \mathsf{Cert})$$

where $\mathsf{UnlockRqt}$ is the user-created certified unlock message and $U\mathsf{Cert}$ is the unions of all set of certificates received in $\mathsf{UnlockRqt}$ responses. The user submits this message to the consensus engine (❸) The consensus engine sequences all $\mathsf{UnlockCert}$ messages; all correct validators observe the same output sequence (❹).

Algorithm 4 describes how validators process these $\mathsf{UnlockCert}$ messages after they are sequenced by the consensus engine. The validator first ensures they did not already process another $\mathsf{UnlockCert}$ or Cert through checkpoint for the same objects keys (Line 4). They then check $\mathsf{UnlockCert}$ is valid, that is, the validator ensures (i) it is correctly signed by a quorum of authorities, and (ii) that all certificates [Cert] it contains are valid (Line 8). The validator can only execute the transaction Tx specified by the user if $\mathsf{UnlockCert.Cert}$ is empty (Line 12). The validator then marks every object key of [$\mathsf{ObjectKey}$] as **Confirmed** to prevent any future unlock requests on the $\mathsf{ObjectKey}$ from overwriting execution with a different transaction (Line 23) and returns a set of $\mathsf{EffectSign}$ to the user (❺).

The user assembles an $\mathsf{EffectSign}$ from a quorum of $2f + 1$ validators into an *effect certificate* $\mathsf{EffectCert}$ that determines finality (❻).

8 Related and Future work

The Cuttlefish's fast path is based on Byzantine consistent broadcast [6]. Previous works suggested using this weaker primitive to build payment systems [1–3, 8, 12, 14, 17] or even as an exclusion-based locking mechanism for optimistic state-machine replication [15]. Zzyzx specifically uses a two-mode unlock mechanism that checks if all replicas have a matching history over the object and retracts the lock or runs full consensus to find the best state to adopt. Unlike Zzyzx, Cuttlefish provides the machinery to not only abort but also directly execute a new transaction and exploits the idea of shared objects to allow for easy execution when there is true contention. Addtionally, Cuttlefish comes with a full set of proofs.

Section 3 extensively discussed Sui [17], the closest systems to Cuttlefish. Notably, Sui includes a restricted variant of multi-owner transactions to support sponsored transactions, and a restricted variant of complex authenticators allowing only weighted thresholds of signatures as an authenticator.

Sui additionally, supports a batch execution mechanism called Programmable Transaction Blocks (PTB). In a PTB a user can bundle multiple of their transactions together for execution and allows for a significant increase in operations per second Sui can process. Unfortunately, this is currently only available for a single owner largely due to the risk of deadlocks if one of the bundled operations is under a race condition. With Cuttlefish we envision providing this significant advantage in terms of throughput efficiency for general-purpose workloads as dapp operators will be able to bundle transactions of many users in a single certificate workflow knowing that if something goes wrong, they could invoke FastUnlock and seamlessly regain liveness.

Another closely related work is FastPay which implements a payment system using a Byzantine consistent broadcast primitive and a lazy synchronizer to achieve *totality* [6]. Zef combines FastPay with the Coconut anonymous credentials scheme [20] to enable confidential and unlinkable payments. Astro relies on an eager implementation of *Byzantine reliable broadcast* [6] to achieve totality without relying on an external synchronizer at the cost of higher communication in the common case. Similarly, ABC [19] proposes a relaxed notion of consensus where termination is only guaranteed for honest users. All these systems lack an integration with a consensus path making them 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). If integrated, then Cuttlefish would apply directly to allow more use-cases on the low latency consensusless path without the risk of locking assets forever due to race conditions.

9 Conclusion

Cuttlefish proposes a novel approach to decentralized ledgers that addresses the shortcomings of previous consensus-minimized systems. By realizing that the requirement for consensus in blockchains is driven by contention rather than the number of owners, Cuttlefish challenges traditional wisdom and provides an alternative perspective. When objects are not under contention, the use of collective objects and multi-owner transactions, combined with the right authentication mechanism enables Cuttlefish to give the majority of the functionality seen in traditional blockchains within two round-trips of communication. To properly deal with deadlock when the objects are under contention, Cuttlefish proposes the novel FastUnlock protocol allowing users to quickly regain access to locked assets. As a result, Cuttlefish allows for the consensus-less execution of a broader set of transactions, including asset swaps and multi-sig transactions that were previously thought to need consensus.

Acknowledgment

This work is supported by Mysten Labs. We thank the Mysten Labs Engineering teams for valuable feedback broadly, and specifically to Xun li and Mark Logan for advising on a design that would best fit the Sui codebase.

References

1. Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous incentive-compatible payment channels. In *Financial Cryptography and Data Security (FC)*, 2021.
2. Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.
3. Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: Low-latency, scalable, private payments. *arXiv preprint arXiv:2201.05671*, 2022.
4. Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus, 2023.
5. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.
6. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
7. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, 1999.
8. Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.
9. George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.
10. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
11. Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 296–315. Springer, 2022.
12. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. AT2: asynchronous trustworthy transfers. *CoRR*, 2018.
13. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.
14. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Principles of Distributed Computing (PODC)*, 2019.
15. James Hendricks, Shafeeq Sinnamohideen, Gregory R Ganger, and Michael K Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 363–372. IEEE, 2010.

16. Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.
17. Mysten Labs. Build without boundaries. https://sui.io, 2022.
18. Linera. Unlocking the power of decentralization. https://linera.io, 2022.
19. Jakub Sliwinski and Roger Wattenhofer. Abc: Asynchronous blockchain without consensus. ArXiv preprint, 2019.
20. Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
21. Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.
22. Sui. Private conversation with the Sui team, 2023.

A Distributed Systems Primitives

This section presents the fundamental definitions of the distributed primitives we use as black boxes.

Consensus. Consensus is the process of agreeing on a value or decision among a group of nodes, each of which has its local input and can communicate with other nodes over a network. Consensus protocols satisfy the following properties:

- Termination: Eventually, every honest node decides on a value.
- Agreement: All honest nodes decide on the same value.
- Validity: If all nodes have the same input value, then any node that decides must decide on that value.
- Integrity: Nodes only decide on values proposed by some nodes.

State Machine Replication. A Byzantine fault-tolerant state machine replication protocol commits client transactions into a sequential log akin to a single non-faulty server, and provides the following two guarantees:

- Safety: Any two honest replicas that commit a transaction at a log position commit the same transaction.
- Liveness: A transaction submitted through an honest replica is eventually committed by all honest replicas at some log position.

SMR typically uses a consensus instance per log position. In this work, we use the term SMR and consensus interchangeably and assume a black box construction. Any type of consensus that provides safety under asynchrony is sufficient, whether classic [7, 11] or DAG-based [9, 16, 21].

Reliable Broadcast. Reliable Broadcast is a weaker version of consensus that provides liveness only in the presence of an honest *source* node that drives the protocol to completion. As we will discuss later in the paper this is sufficient for the safety and liveness of transactions that do not experience contention on the objects they operate. Unfortunately in the presence of a faulty or buggy source node objects can lose liveness forever. A reliable broadcast algorithm should satisfy the following properties:

- Validity: If an honest node broadcasts a message, then every honest node eventually delivers that message.
- Integrity: If an honest node delivers a message, then that message was previously broadcast by the source.
- Agreement: If a honest node delivers a message m, then every honest node delivers m.

B Epoch Change

Sui divides time into a sequence of *epochs*, each comprising an approximately equal number of checkpoints. At the end of each epoch, validators release all their locks to allow users with equivocated objects to regain access to them in the next epoch[5]. However, this design choice introduces a potential risk: what happens to transactions that were in progress during the epoch change?

To mitigate this risk, Sui implements an incremental epoch change process. As a first step, validators pause transaction processing and focus solely on generating checkpoints. This ensures that validators can eventually terminate once all executed transactions are secure. In a second step, validators submit all the certificates they have executed for checkpointing. Once all the certificates known to the validator have been checkpointed, the final step is for the validator to send an *end-of-epoch* message to be sequenced by the consensus protocol. The epoch change is considered complete when $2f+1$ *end-of-epoch* messages are sequenced.

The safety of transactions in progress right before the start of the epoch change process is guaranteed by the requirement that an honest validator must not send an *end-of-epoch* message until all the certificates it executed are sequenced (either by itself or others inserting them into the consensus protocol). This ensures that if a transaction has been finalized in the fast path, at least one honest validator will either include it in a checkpoint or prevent the epoch from completing. As a result, no finalized transaction is reverted or 'forgotten' during the epoch change. The preservation of this property can be extended to apply directly to the unlock transactions.

C Handling Gas Objects

Typical transactions not only mutate objects but also consume a gas object to spend for the computation. If, however, the transaction is locked then this gas is blocked as well. For this reason Cuttlefish requires a fresh gas-object in order for

[5] Transactions are valid only within the epoch they were signed to prevent replays.

consensus to process the unlock request. Specifically together with Algorithm 1, the parties should provide a fresh gas object for their request. This gas object is checked for validity along with the check in Line 3 and locked for the unlock transaction in Line 11. When the user collects the no-commit proof in the second step of the protocol, the $2f + 1$ collected signatures also serve as a certificate for the gas object. The consensus then checks the validity of the certificate and spends it locally before entering Algorithm 2. Then when consensus executes the transaction three scenarios may happen:

- The unlock request is valid and includes a certificate. Then the execution happens as usual and both the gas object for the unlock and the gas object for the execution are consumed.
- The unlock requests is valid and comes with a no-op. Then the gas object for unlock is consumed. If there was some locked transaction racing the FastUnlock then the accompanying gas object is potentially blocked. The user can then explicitly unlock that gas object by running FastUnlock.
- The unlock request is not processed because a checkpoint certificate already executed a transaction. Then the gas object is still consumed without altering the state of the ObjectKey.

D Security Arguments

We argue about the safety and liveness of Cuttlefish. Intuitively, Cuttlefish does not invalidate the finality guarantees of the normal fast path operations. That is, a client holding an effect certificate can be assured that its transaction will not be reverted.

Theorem 1. *If there exists an effect certificate EffectCert over a transaction Tx, the execution of Tx is never reverted.*

Proof. We assume that the execution of Tx is reverted and lead to a contradiction. The transaction cannot be reverted at the end of the epoch as it will contradict the properties we inherit from Sui which Cuttlefish did not modify. Hence, the transaction can only be reverted if there exist an UnlockCert over an ObjectKey modified by Tx. For this to happen there should be an UnlockCert over that ObjectKey carrying an empty certificate. From Check (1.2) of Algorithm 1 a correct validator only provides and UnlockVote with an empty Cert if it has not executed anything for ObjectKey. From our assumption that ObjectKey did admit a no-op there should be $f+1$ honest validators that did not partake in the generation of the EffectCert of Tx and hence passed the check. Additionally, for the EffectCert to exist by definition it has $2f + 1$ signatories over the ObjectKey in question, at least $f + 1$ of them being honest. This implies a total of at least $f + 1 + f + 1 + f = 3f + 2 > 3f + 1$ validators, hence a contradiction.

The dual also applies meaning that if an UnlockCert exists then no EffectCert over the ObjectKey will be generated in the fast path. The proof analogue by add an extra check during the EffectCert generation that correct validators refuse to process certificates when the recorded **Unlocked** in their UnlockDb.

Next we show that validators that might process on the consensus path both a Cert (through checkpointing) and UnlockCert will arrive at the same execution result. We prove the case where an UnlockCert is ordered first. For this, we need to enhance the protocol of checkpointing in Sui to check the value of UnlockDb[ObjectKey] and ignore a *cert* that tries to process a **Confirmed** *Okey*, which is a straightforward change.

Theorem 2. *If a correct validator executes an UnlockCert certificate over ObjectKey as sequenced by the SMR engine, no correct validator will subsequently execute a conflicting a Cert as sequenced by the SMR engine .*

Proof. The proof directly follows from the safety property of the SMR engine that all validators will process certificates in the same order. Hence, upon processing UnlockCert, all honest validators mark the execution of ObjectKey as confirmed by setting UnlockDb[ObjectKey] $\leftarrow$ **Confirmed** (Line 18 of Algorithm 2). Then, check (2.1) of Algorithm 2 (and its dual added at the checkpoint algorithm) ensures that if any further Cert or UnlockCert with a conflict is given as input to the execution engine it is rejected.

The dual can be proven in the same manner since we enhance the execution of Cert during the checkpoint process with updating UnlockDb[ObjectKey] $\leftarrow$ **Confirmed** after processing. Then all UnlockCert on the ObjectKey will be rejected at the Check (2.1) of Algorithm 2.

Liveness argument. Intuitively, we argue that Cuttlefish—and its composition with normal fast path operations—neither deadlocks nor enables unjustified aborts (which could starve an object from progress).

Lemma 1 (Unlock Certificate Availability). *A correct user can obtain an unlock certificate UnlockCert over a valid ObjectKey.*

Proof. A correct validator always signs UnlockVote if it passes the check of Algorithm 1. Well formed UnlockRqt always come with a valid authentication path (Check (1.1)), and Check (1.2) always returns an UnlockVote. As a result, if UnlockRqt is disseminated to $2f + 1$ correct validators by a correct user, they will eventually all return an UnlockVote. The user then aggregates those votes into a unlock certificate UnlockCert over ObjectKey.

Theorem 3 (Cuttlefish Liveness). *If a correct and authorized user initiates a fast-unlock protocol, the ObjectKey in question will eventually admit a new transaction.*

Proof. A correct and authorized user will eventually generate an unlock certificate by Lemma 1. Additionally from the liveness property of SMR the unlock certificate will either eventually be added as part of the SMR output or the epoch will end. If the first happens by agreement of consensus the UnlockCert will be executed by all validators, leading to the termination of the fast-unlock protocol and an updated ObjectKey. If the epoch ends, all locks are dropped and liveness of all ObjectKey are automatically available for processing.

Theorem 3 is sufficient for correct users as either they will manage to no-op an incorrect invocation of ObjectKey, drive the tranasction of a correct Tx to completion, or the epoch end will automatically unblock them. This means that there will always be an available ObjectKey to be modified.

Now that we proved that an authorized user will succeed into unblocking the ObjectKey we also need to show that an unauthorized user will not succeed into starving legitimate users from progress through abusing fast-unlock.

Theorem 4 (Starvation Freedom). *An user cannot successfully initiate a fast-unlock on any ObjectKey it is cannot produce an Auth.*

Proof. All honest validators check the authorization vector Auth of the requesting user (Line 3 in Algorithm 1). This means that no honest party will lock an object without an authorization, including slow parties that have not yet seen the ObjectKey which will reject or cache the request for later processing. As a result, by the model, there will never be sufficient UnlockVote to generate an UnlockCert driven by an unauthorized user.

Generalization to multi-object unlock. The multi-object unlock protocol can be seen as a composition of many single-object unlock protocols (one per object) as well as a single commit protocol (for the accompanied transaction). As a result, the safety of the protocol follows from the fact that objects are independent of each other so if at least one has a prior certificate then the commit flow will lead to committing that prior certificate (which iteratively applies to all objects with prior certificates). If on the other hand, no object has a prior certificate then the workflow is the combination of the simple FastUnlock per object together with the shared-object path of committing the transactions of Sui which is safe as proven in the original Sui paper [4].

Second, we explore liveness. There are two cases: (1) all objects can be unlocked, (2) one or more objects are already certified. The first case is exactly the same as the simple protocol of Section 6 and a proof would follow exactly the same structure. For the second case, we first look into the base case of a single object that is already certified which is already proven in the previous sections. For more than one objects we can see that since the validator adds all certificates in their reply and then processes each certificate separately when handling the unlock cert then there is no interaction between certificate processing and can be considered a batch of independent requests.

Finally, for liveness the accompanied transaction might need to acquire locks. This is also an independent invocation of the Sui fast-path. As a result if the transaction is valid it will either succeed or blocks. In the latter case, the user will have to invoke fast-unlock again including in the set of to-unlock objects the newly blocked objects of the transaction. Given that there is a finite number of objects a user holds an unlock request will eventually succeed.