Pallas and Aegis: Rollback Resilience in TEE-Aided Blockchain Consensus

Jérémie Decouchant
Delft University of Technology
j.decouchant@tudelft.nl

David Kozhaya
ABB Corporate Research
david.kozhaya@ch.abb.com

Vincent Rahli
University of Birmingham
v.rahli@bham.ac.uk

Jiangshan Yu
University of Sydney
jiangshan.yu@sydney.edu.au

***Abstract*—Several Byzantine Fault-Tolerant (BFT) consensus algorithms leverage trusted components to boost resilience and reduce communication overhead. However, recent findings expose a critical vulnerability to rollback attacks when trusted components crash, lose state, or be cloned. Existing defenses either treat crashed replicas as Byzantine, increasing replica count, or duplicate trusted state across components, incurring substantial performance costs and offering limited crash tolerance.**

We propose a robust alternative: a secure state-preservation mechanism for trusted components that eliminates costly duplication of trusted states across replicas. At its core is `Aegis`, the first efficient view synchronizer specifically designed for BFT protocols that utilize trusted components. `Aegis` enforces that only one trusted component instance per replica may vote in any view, even when trusted components restart following a crash or are cloned by an adversary. On top of `Aegis`, we introduce `Pallas`, the first BFT consensus protocol that preserves safety against a strong adversary that controls a fixed set of Byzantine replicas and can cause a potentially unbounded and varying number of trusted components to crash. We determine the adversarial conditions under which `Pallas` ensure liveness under partial synchrony.

Extensive geo-distributed evaluations on Amazon AWS show that `Pallas` delivers high performance with negligible overhead in stable conditions, outperforming existing protocols by up to 41% in throughput and 29% in latency. More importantly, it sustains liveness and graceful degradation under adversarial conditions where other protocols fail.

I. Introduction

Trusted Execution Environments (TEEs) [1–3] are increasingly being incorporated into Byzantine Fault-Tolerant (BFT) consensus protocols to enhance resilience without increasing system size, or to achieve comparable fault tolerance with fewer replicas, thereby improving performance. By leveraging TEEs, recent protocols can operate with as few as $2f+1$ replicas [4–7] to tolerate f Byzantine replicas, a significant improvement over the traditional $3f+1$ requirement [8–10]. In addition, TEE-based consensus algorithms [5, 6] benefit from a reduced number of communication phases, which reduces latency. However, TEEs introduce new security vulnerabilities, most notably rollback attacks [11], where a TEE is reset to a previously recorded state or is duplicated. These attacks allow a single replica to emit conflicting votes, a problem known as equivocation, which leads to consensus safety violations.

While rollback attacks have long been recognized as a security concern in both TEE and cloud computing literature [11], their implications in the context of BFT consensus protocols remain inadequately addressed [12]. Current solutions often fall short mainly due to impractical countermeasures that dilute or negate the very performance gains TEEs are designed to provide. One commonly proposed mitigation strategy involves replicating the internal state of TEEs across multiple replicas or trusted parties [13–16]. However, this approach introduces considerable computational and communication overhead. The burden of synchronizing sensitive TEE state across replicas weakens their performance efficiency, negating in some aspects the reasons for their deployment in the first place. Another commonly suggested approach is to treat TEE failures, including crashes and restarts, as manifestations of Byzantine behavior [12]. The downside of such an approach is that it inflates the minimum system size required to tolerate a given number of faults. As a result, protocols would often need significantly more replicas, thereby offsetting the scalability and low-latency advantages TEEs otherwise promise [17].

More critically, the literature on rollback-resistant consensus algorithms tends to only model weak adversarial capabilities. For instance, much of the prior work assumes a Byzantine attacker that can revert a TEE to a previously valid but stale state [18]. This overlooks far more sophisticated attack vectors, such as the creation of cloned TEEs (a.k.a. forking [11, 19]) with divergent states [13], or remote denial-of-service (DoS) attacks that cause host-level crashes or reboots. Since most TEEs do not persist enclave state across restarts, such events result in the loss of volatile state. Under a mobile adversary with such capabilities [20], all previous consensus algorithms would eventually lose liveness because they do not allow crashed replicas to safely rejoin the protocol. Supporting dynamic membership is essential but challenging, as improper handling risks violating safety.

This paper addresses these challenges by introducing a rollback-resilient consensus framework that maintains both safety and liveness under a significantly powerful adversarial model, one that encompasses static Byzantine faults with a wide TEE attack vector as well as mobile TEE crashes [20]. We identify that maintaining liveness under rollback-prone conditions requires a carefully coordinated interaction between

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242443
www.ndss-symposium.org

the consensus algorithm and its view synchronizer. Specifically, the consensus must support safe rejoining of new TEE instances, while the view synchronizer must maintain accurate membership knowledge.

To this end, we first introduce `Aegis`[1], a robust view synchronizer designed to guide correct replicas in reaching agreement on the current view number. It tightly regulates TEE voting privileges during view changes, allowing only one active TEE instance per replica in any session. Traditional view synchronizers lack provisions for securely reinserting or replacing a replica's TEE following its crash. In contrast, `Aegis` is the first synchronizer that is purpose-built to efficiently support trusted hardware, resolving TEEs joining the system via a session-oriented approach that decouples join approval from activation. This design enables robust recovery from TEE failures without jeopardizing the complexity of the system or the number of replicas.

We integrate `Aegis` within a high-performance BFT consensus protocol equipped with rollback-aware safeguards and TEE membership restoration capabilities, forming a cohesive system that tolerates a broad range of adversarial conditions. Our protocol, which we call `Pallas`[2], ensures progress under partial synchrony [21] and adversarial conditions, while retaining a minimal configuration of $N = 2\mathcal{F} + 1$ replicas, where $\mathcal{F}$ denotes the total number of faulty replicas, accounting for both Byzantine failures and crash-induced TEE faults (see Section II-B). Through rigorous experimentation on real-world infrastructure, we demonstrate that `Pallas` delivers high performance with negligible overhead in stable conditions, outperforming state-of-the-art BFT consensus systems by up to 41% in throughput and 29% in latency. Importantly, it sustains liveness and low latency despite frequent TEE crashes, degrading gracefully and recovering robustly in adversarial conditions where competing protocols cease to be live.

In summary, we provide the first practical BFT consensus protocol that is both efficient and provably resilient to rollback attacks, enabling the safe use of TEEs in critical distributed systems. Overall, we make the following **contributions**:

- We present a powerful adversarial model in the partially synchronous setting that captures both static Byzantine faults and mobile TEE crashes, expanding prior threat assumptions to more realistically reflect attack surfaces encountered in practical deployments.
- We introduce `Aegis`, a view synchronizer that enforces the participation of at most one active TEE instance per replica in each view. While `Aegis` shares high-level structural elements with Cogsworth [22], specifically its phased dissemination of votes, it departs from conventional exponential-backoff approaches and, to the best of our knowledge, is the first synchronizer to support TEEs that provide rollback resistance, clone protection, and increased Byzantine robustness.
- We present `Pallas`, the first BFT consensus protocol to guarantee safety and liveness under rollback attacks without replicating TEE state or increasing the replica count beyond $N = 2\mathcal{F} + 1$. By combining secure TEE-join handling, rollback recovery, and view synchronization into a single efficient protocol, `Pallas` maintains robust operation even under repeated TEE crashes. A provably safe recovery mechanism enables eventual recovery from an unbounded number of crashes, setting `Pallas` apart from prior protocols that lose liveness or require manual intervention under repeated or mobile faults.
- We implemented `Pallas` and `Aegis` on top of Damysus [6], a cutting-edge streamlined TEE-based BFT protocol using Intel SGX [23] and evaluated it through AWS EC2 geo-distributed deployments. Our results demonstrate that `Pallas` outperforms existing solutions under normal and frequent crash-recovery conditions.

[1]The Aegis is the single most powerful and characteristic piece of armor or equipment associated with Pallas Athena.

[2]The Greek goddess Athena, who is a symbol of wisdom and strategic warfare, is frequently referred to in classical texts as Pallas Athena or simply Pallas.

II. Models and Objectives

A. System Model

We consider a system of N replicas that are each equipped with a trusted component that executes code securely within an isolated memory region and supports remote attestation. A trusted component can save its state encrypted on disk, and upon crashing, use it to recover its state. Replicas implement a Byzantine fault tolerant (BFT) state-machine replication protocol.

We adopt the standard communication model that has been assumed by most BFT protocols [6, 14, 24]. Replicas exchange messages over a partially synchronous and fully connected network that might duplicate, delay or drop messages. The network being partially synchronous [21] means that there may be unstable periods of time where messages exchanged between correct processes are arbitrarily delayed. However, there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all messages are guaranteed to be delivered within Δ time, which enables liveness. In this model, consensus safety has to be ensured at all times, while liveness has to be guaranteed after GST. Trusted components do not have direct access to the network, and rely on their hosts to transmit and receive messages. We also assume the existence of a Public Key Infrastructure used by participants to distribute the keys required for authentication and message signing. Replicas have access to classical cryptographic primitives, including secure hash functions and digital signature schemes.

B. Threat Model

We consider a *hybrid fault model* consisting of a *static Byzantine adversary*, which causes *Byzantine faults*, and a *mobile crash adversary*, which causes *crash faults* of TEEs. Out of the N replicas in the system, we tolerate $\mathcal{F} = f+u$ faulty replicas, where f denotes the maximum number of Byzantine replicas and u denotes the maximum number of crashed TEEs. We assume that the two adversaries can collude and coordinate perfectly.

The *Byzantine adversary* is capable of compromising up to f replicas over the system's lifetime. Once compromised, a replica may behave arbitrarily and its internal components can be tampered with, except the TEE components. However, the adversary retains the ability to fork the TEE, i.e., instantiate multiple clones of a TEE at any previously observed state, thereby undermining assumptions about the uniqueness or freshness of TEE state. Additionally, the adversary can also mount a rollback attack by supplying a restarting TEE with stale encrypted state retrieved from disk [11, 19].

The *mobile crash adversary* can crash the TEE of replicas at any given time by resetting their trusted components, e.g., by crashing the replica through DoS attacks, thereby erasing the internal state of its TEE component. However, in practice, while the TEE may lose its state and become non-functional, the replica may still resume operations that do not require TEE access. In such cases, the replica continues to function, albeit without its TEE, until the TEE is eventually restored. The term "mobile" reflects the adversary's ability to shift control from one replica to another over time. This movement is constrained by a security parameter λ, which bounds the minimum time between successive compromises. We assume that after GST, the mobile adversary's speed, which is dependent on λ, may allow it to crash up to u replicas simultaneously. This, for example, can be guaranteed by regulating the speed at which those crashed replicas can recover compared to the adversary's speed for crashing new ones. Although initially the adversary can only control at most u replicas simultaneously, it is possible that it may crash additional replicas if some previously compromised replicas are taking "too long" to recover. As a result, the total number of replicas in a crashed state may temporarily exceed u. This can happen, for instance, during an extended period of asynchrony before GST, where typical membership and recovery protocols lose liveness, giving the adversary enough time to move to new targets before earlier victims have recovered. In the worst case, an attacker can crash the TEE of all correct replicas, forcing them to restart their TEEs, affecting liveness. A detailed analysis of the parameter λ for our protocol is provided in Appendix B.

C. Security goal

Our goal is to allow the system to guarantee consensus safety at any time and consensus liveness after GST, while tolerating $\mathcal{F}$ faults with $N \geq 2\mathcal{F}+1$ replicas.

III. Solution Overview and Key Ideas

Rollbacks in Trusted Execution Environments (TEEs) introduce a serious threat to safety in TEE-based consensus protocols. If a TEE reverts to a prior state while retaining voting privileges, effectively functioning as a duplicate of its former self, it may violate consensus safety by equivocating, potentially enabling a single replica to vote more than once. To mitigate this risk, we implement a design that enforces strict voting eligibility: each replica is restricted to a single active TEE instance per view. This instance is uniquely identified and authorized to cast votes. Any additional TEE instances, such as outdated clones from earlier replica states, are explicitly prohibited from participating in the voting process.

We achieve this by organizing protocol execution into sessions, each comprising multiple consensus views, and use a periodic view synchronizer to apply membership changes for enforcing TEE membership consistency, while maintaining a system size of $N \geq 2\mathcal{F}+1$. In this design, the synchronizer acts as a reliable broadcast channel to announce and activate new TEE instances, each tagged with a unique, freshly generated[3], and agreed-upon nonce. Agreement on these identifiers is established through consensus executions carried out between two consecutive synchronizer invocations. In this way, a synchronizer execution relies on the membership that has been decided by the consensus instances completed since the last successful synchronizer execution. This process guarantees that all correct replicas recognize exactly one valid TEE instance per replica per view, thereby preventing multiple clones from participating concurrently.

Interestingly, deciding which replicas to include in a view without getting the normal consensus execution involved, i.e., by relying solely on the view leader in the view synchronizer, is not possible. Modern synchronizer designs use multiple leaders to trigger a view change (for liveness). Consequently, if distinct leaders were to initiate the same view while selecting different memberships, correct replicas may have inconsistent perspectives on the participating nodes, violating agreement. This could happen, for example, if different leaders were to accept different TEE instances of the same replica in concurrent executions of the synchronizer.

Alternative mechanisms that attempt to enforce a single valid TEE instance per replica and per view, such as relying on local decisions or deferring membership agreement to in-progress consensus executions, are also ineffective. Local uncoordinated decisions cannot guarantee that all correct replicas agree on view membership (including the leader and voting set) prior to view initiation, a critical safety requirement. Meanwhile, determining membership on the fly solely via consensus executions (by reading the latest agreed upon membership changes) is possible, but requires increasing the replica count to ensure a quorum always remains available despite Byzantine faults and crashes.

With $N = 2f+2u+1$, our solution ensures that, after GST, at least $f+1+u$ replicas remain continuously online, each with a unique non-forked TEE instance. This design guarantees that the majority of active replicas in any given view have distinct TEE identities, a property that is fundamental to ensuring both safety and liveness.

A. Replica Restarts

Whenever a replica initiates a new TEE instance, such as following a restart, it must issue a join request encoded as a

[3]We force TEEs to generate a new nonce when restarting, preventing the existence of two identical TEE instances (see Fig. 3).

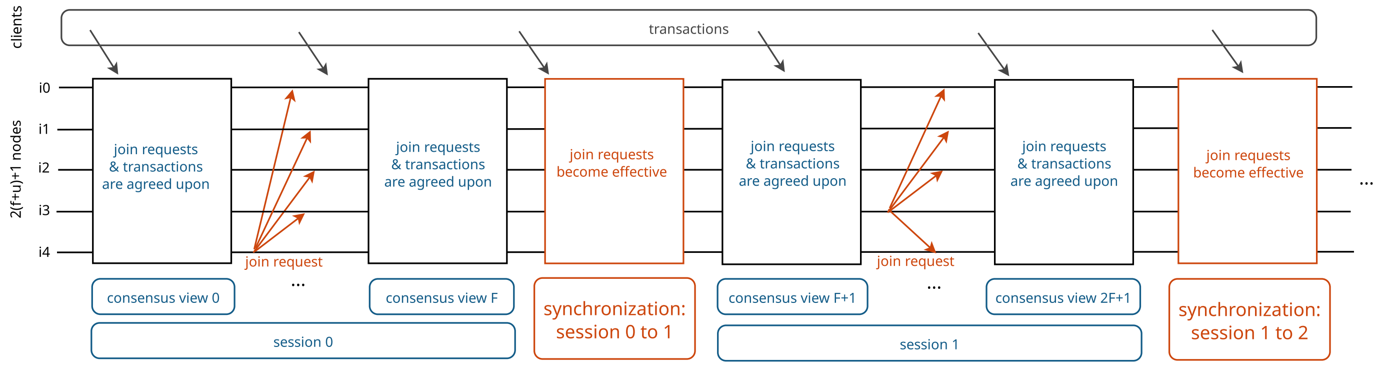
clients
transactions
2(f+u)+1 nodes
i0
i1
i2
i3
i4
join requests & transactions are agreed upon
join request
join requests & transactions are agreed upon
join requests become effective
join requests & transactions are agreed upon
join request
join requests & transactions are agreed upon
join requests become effective
...
consensus view 0
...
consensus view F
synchronization: session 0 to 1
consensus view F+1
...
consensus view 2F+1
synchronization: session 1 to 2
session 0
session 1

Fig. 1: High-level view of the protocol

special transaction to be processed through standard consensus. To ensure that no replica has multiple active TEE instances within a single view, if multiple join requests from the same replica are generated for the same view, only the first request is accepted. Subsequent requests for the same view and from the same replica are discarded. The consensus leader of the view governs the filtering and ordering of these join requests, while other replicas have to vote on them, enforcing the uniqueness of a TEE instance per replica per view. Once the join request is accepted by consensus, the new TEE instance can start being involved only after the next synchronizer execution. As previously discussed, while immediate integration without synchronizer execution might be technically feasible, it would demand additional adaptations of the consensus protocol. Our approach favors generality and wider applicability.

B. Solution Components and Their Interaction

Our rollback-prevention mechanism combines a consensus algorithm (built on top of Damysus [6]) and a view synchronizer (inspired by Cogsworth [22]). When replicas restart with new TEE instances, they broadcast TEE join requests, which are validated by consensus (like standard transactions). The synchronizer periodically activates committed TEE join requests letting approved replicas rejoin and become active voting participants. Immediate rejoining, i.e., in the next view as opposed to periodic activation, is unsafe without safeguards to prevent equivocation - only one TEE per replica is allowed per view. We argue that enforcing these safeguards becomes particularly challenging when view v cannot be completed, due to factors such as network delays or a faulty leader, and replicas must leap ahead to view $v{+}1$ from an earlier point to preserve system liveness. Hence, our approach decouples join request approval from execution for modularity and safety.

Our approach makes the system progress in successive sessions, each potentially encompassing several consensus views. At the end of a session, the synchronizer activates approved join requests, allowing replicas to rejoin. Deferring membership changes to session boundaries guarantees that no two clones of a replica operate within the same session. Even if a view within a session fails, the synchronizer ensures transition to the next session without violating this rule.

As illustrated in Fig. 1, for parameters $\mathcal{F} = 2$ and $f = u = 1$, permitting one Byzantine fault and one recoverable crash. Each session spans $\mathcal{F} + 1$ views. During these views, replicas agree on client transactions as well as on join requests. For example, replica i_4 restarts in session 0, and its join request is passed to consensus and is approved during that session. It reenters the system only at view $\mathcal{F}{+}1$, marking the start of session 1. Likewise, i_3 restarts in session 1 and resumes participation at view $2(\mathcal{F}{+}1)$. With $u = 1$, our system maintains safety and liveness after GST with at most one crash at any time. However, our mobile crash model permits distinct replicas to crash and recover over the course of time: safety is always maintained and liveness is ensured after GST if the recovery rate outpaces the failure rate.

View Synchronizer-Consensus Interactions: The pseudo code in Fig. 2 outlines how replicas manage join requests and view changes by interacting with both the view synchronizer (Fig. 3) and the consensus protocol (Fig. 4).

Each replica maintains a local session number $session$ and view number $view$, outside the TEE, to record the latest session and view it has reached. Correct replicas and those under rollback attacks, store their own $session$ and $view$ correctly, whereas Byzantine replicas can modify them arbitrarily. The role of the synchronizer is to periodically synchronize all correct and recovering replicas to a common session and view.

Every $sync\_period$ views, replicas initiate view synchronization by calling `wish_to_advance()`. When a restarted replica spins up a new TEE instance, it triggers `wish_to_join()` (l. 12, Fig. 2a) to generate a join request. A correct replica does so immediately upon reboot, while there is no guarantee when a Byzantine replica might call this function. When `wish_to_join()` is called, the replica's TEE creates a join request for the next session number. Specifically, since each replica stores the last session it participated in on permanent storage outside the TEE (l. 50, Fig. 3a), calling the TEE function `TEErequestjoin` to generate a join request produces a request for $session{+}1$, where $session$ is the replica's last session before it crashed. `TEErequestjoin` is defined at l. 13 of Fig. 3b and invoked at l. 14 of Fig. 2a.

When a replica attempts to rejoin the system, it may not know which replica is acting as leader and hence it broadcasts

(a) Non-trusted code at replica i

```
1:  Common variables:
2:  • $view := 0$ // current view
3:  • $session := 0$ // current session number
4:  • $\phi_{store}$ // last store certificate
5:  • $received\_joins := \emptyset$ // set of received join requests
6:  • $sessions$ // array of last joined sessions per replica
7:  • $sync\_period$ // synchronization period
8:  • $last\_sync$ // number of views since last synchronization
9:  • $blocks$ // maps views to blocks
10:
11: // for replicas to (re)join the system
12: function wish_to_join()
13:     if a join certificate for session $session+1$ has not been generated then
14:         $\phi_{join}$ := TEErequestjoin($session+1$)
15:         record on disk that a $\phi_{join}$ certificate was generated
16:         send join($\phi_{join}$) to all
17:     else
18:         send the join certificate for session $session+1$ to all
19:     end if
20:
21: // to handle join requests
22: all replicas
23:     wait for a join request $\phi_{join}$
24:     $joins$ := join requests in $blocks$ from beginning of session up to $\phi_{store}$'s view
25:     update $received\_joins$ with $\phi_{join}$ if its session exceed those in $joins$, the one
    in $received\_joins$ and the one in $sessions$
26:
27: // for replicas to start a new view or synchronize
28: function start_new_view()
29:     if $last\_sync > sync\_period$ then
30:         wish_to_advance()
31:         $last\_sync := 0$
32:     else
33:         send $\phi_{nv}$ := TEEnewview($\phi_{store}$) to $view$'s leader
34:         $last\_sync$++
35:     end if
36:
```

(b) TEE code

```
1:  Common variables:
2:  • $id$ // fixed, hardware based
3:  • $syncing$ := false // consensus/sync state, volatile
4:  • $view := 0$ // current view, volatile
5:  • $session := 0$ // current session, volatile
6:  • $prepv := 0$ // view of latest prepared block, volatile
7:
8:  // generic accumulator:
9:  function TEEaccum($\phi, \vec{\phi}$) where $\phi$ is $\langle \texttt{TAG}, p \rangle_\sigma$ and $\vec{\phi}$ has size $\mathcal{F}$
10:     if all certificates are from different replicas $\wedge \forall \phi' \in \vec{\phi}.\phi' \leq \phi$ then
11:         return $acc := \langle \texttt{ACC}|\texttt{TAG}, p \rangle_{\sigma'}$
12:     end if
13:
```

Fig. 2: High-Level pseudo-code of the glue code between the synchronizer and Damysus

its join request to all other replicas. Notably, the replica does not need precise knowledge of the current session number. For example, replica j may attempt to join $session+1$ even if the system has already progressed to $session+2$. Such a request remains valid and will be considered by the consensus protocol. To guard against replay of stale join requests, which could be exploited to eject a previously rejoined TEE, each replica maintains a record of the last session joined by every other replica in an array $sessions$ (l. 6, Fig. 2a and maintained in Fig. 3a).

Upon receiving a join request from replica j, the system performs two checks: (1) it checks that the request's session number is strictly greater than $sessions[j]$; and (2) it verifies that no pending or prepared join from j in the current session has a higher or equal session number. A join request that passes those two checks is accepted and added to $received\_joins$. Otherwise, the request is discarded as stale or duplicate, ensuring that outdated messages cannot be replayed to trigger unintended re-join. During regular consensus executions, pending join requests are treated as special transactions. These requests reside in the $received\_joins$ set until they are incorporated into a prepared block, at which point they transition into the standard block ledger.

At the end of each consensus instance (or view), replicas must determine whether to remain in the current session or initiate synchronization for a new session. This decision hinges on the system's synchronization schedule. Specifically, if the view number satisfies $view \mod sync\_period = 0$, it indicates the session has reached its end. In such cases, the replica triggers `wish_to_advance()` to activate the synchronizer (l. 7, Fig. 3). If the session is not due for synchronization, the replica proceeds to the next view within the same session. It does so by sending a new-view message to the designated leader of that view (l. 33, Fig. 2a).

IV. View-Synchronization, TEE Membership Update and Recovery

This section outlines the mechanism by which replicas, whose TEEs may have restarted, can safely rejoin the system.

Traditional synchronizer designs, which coordinate view transitions among replicas, are not suited for TEE-assisted protocols and lack provisions for handling rejoining replicas in a secure and consistent manner. In our approach, replicas that restart, whether due to benign failures or adversarial resets, submit a special join request to re-enter the system. These requests are treated as transactions and subject to consensus voting. At regular intervals, `Aegis`, our modified synchronizer (Sec. IV-A) is invoked to incorporate replicas whose join requests have been approved, thereby updating the system membership in a secure and coordinated fashion.

A. `Aegis`: View Synchronizer and Rollback-Prevention

Fig. 3 presents our synchronizer, which prevents rollback attacks by using random nonces to uniquely identify TEE instances. When a replica's TEE restarts and issues a join request via `TEErequestjoin` (l. 13, Fig. 3b), it includes a freshly generated nonce and the target session. This nonce uniquely identifies the TEE instance and is embedded in the join request, which is then voted on by the consensus protocol.

To prevent concurrent participation by multiple TEE instances from the same replica, the synchronizer enforces a strict rule: within each synchronization interval, only one nonce per replica per session may be accepted. Upon restart, a replica generates a single nonce during its initialization (via `TEErejoin`). If multiple join requests with differing nonces are submitted for the same session, only one is accepted and

(a) Non-trusted code at replica i

```
Common variables from Fig. 2a
  • $view := 0$          • $\phi_{store}$
  • $session := 0$       • $sessions$
                         • $blocks$

1: Additional variables:
2: • attemptedTC := 0
3: • attemptedQC := 0
4: • $acc_{sync}(s)$ // latest request to synchronize
5:
6: // for replicas to synchronize
7: function wish_to_advance()
8:     $\phi_{sync(s)}$ := TEEsync($\phi_{store}$)
9:     send $\phi_{sync(s)}$ to leader($s$)
10:
11: as a leader($r$)
12:     receive $\mathcal{F}$+1 $\phi_{sync(s)}$ or 1 $acc_{sync}(s)$, such that $s \le r \le s+\mathcal{F}$,
13:         for the first time (different $acc_{sync}(s)$ from different leaders are allowed)
14:     if received $\mathcal{F}$+1 synchronization requests then
15:         $\phi_{sync(s)}$ := wish certificate with highest view
16:         $\vec{\phi}_{sync(s)}$ := $\mathcal{F}$ other wish certificates
17:         $acc_{sync}(s)$ := TEEaccum($\phi_{sync(s)}, \vec{\phi}_{sync(s)}$)
18:     end if
19:     send $acc_{sync}(s)$ to all
20:
21: all replicas
22:     receive a valid $acc_{sync}(s)$ from leader($r$) such that $s \le r \le s+\mathcal{F}$
23:     $acc_{sync}(s)$ is of the form $\langle \texttt{ACC|SYNC}, s, v, h\rangle_{\sigma_j}$
24:     send $acc_{sync}(s)$ to {leader($s$), ..., leader($s+\mathcal{F}$)}
25:     $joins$ := join requests in $blocks$ from beginning of the session up to $v$
26:     $J$ := [] // pairs of ids/nonces
27:     for each $\langle s', nonce\rangle_{\sigma_k} \in joins$ do $J[k] := nonce$
28:     $\phi_{vote(s)}$ := TEEvotejoin($acc_{sync}(s), J$)
29:     send $\phi_{vote(s)}$ to leader($r$)
30:
31: all replicas
32:     when timeout after $2\Delta$ from sending a $\phi_{sync(s)}$
33:         and not receiving $acc_{sync}(s)$
34:         and attemptedTC $\le view+\mathcal{F}+1$
35:     send $\phi_{sync(s)}$ to leader(attemptedTC)
36:     attemptedTC++
37:
38: as a leader($r$)
39:     receive $\mathcal{F}$+1 $\phi_{vote(s)}$ such that $s \le r \le s+\mathcal{F}$,
40:         with the same first four fields $s, v, h, J$,
41:         and has not sent a $\phi_{cert(s)}$
42:     $\vec{\sigma}$ := the collection of $\mathcal{F}$+1 signatures
43:     $\phi_{cert(s)}$ := $\langle \texttt{VOTE}, s, v, h, J\rangle_{\vec{\sigma}}$
44:     send $\phi_{cert(s)}$ to all
45:
46: all replicas
47:     receive a valid $\phi_{cert(s)}$ from leader($r$) for $s \le r \le s+\mathcal{F}$
48:     $\phi_{cert(s)}$ is of the form $\langle \texttt{VOTE}, s, v, h, J\rangle_{\vec{\sigma}^{\mathcal{F}+1}}$
49:     attemptedTC := $s$; attemptedQC := $s$; $session$ := $s$; $view$ := $v$
50:     store $session$ on disk
51:     update $sessions$ so that the replicas in $J$ are mapped to $s$
52:     $\phi_{store}$ := TEErejoin($\phi_{cert(s)}$)
53:     start_new_view()
54:
60: all replicas
61:     when timeout after $2\Delta$ from sending a $\phi_{vote(s)}$
62:         and not receiving $\phi_{cert(s)}$
63:         and attemptedQC $\le view+\mathcal{F}+1$
64:     // votes are sent when receiving a TC ll. 22-29
65:     send $\phi_{vote(s)}$ and $acc_{sync}(s)$ to leader(attemptedQC)
66:     attemptedQC++
67:
```

(b) TEE code

```
Common variables from Fig. 2b
  • $id$                      • $view := 0$
  • $syncing$ := false        • $session := 0$
                              • $prepv := 0$

1: Additional variables:
2: • $nonce := \perp$ // nonce used to join a view, volatile
3: • $initialized$ := false // set to true once initialized
4:
5: // Freshness generation mechanism
6: function TEEsync($\phi_{store}$) where $\phi_{store}$ is $\langle \texttt{PCOM}, s, v, h\rangle_{\sigma'}$
7:     if $initialized \wedge nonce \neq \perp \wedge s = session \wedge v = prepv$ then
8:         $syncing$ := true
9:         return $\phi_{sync(session+1)} := \langle \texttt{SYNC}, session+1, prepv, h\rangle_\sigma$
10:    end if
11:
12: // $s$ is a session number provided by the OS:
13: function TEErequestjoin($s$)
14:     if $nonce = \perp$ then
15:         $nonce$ := generate new nonce
16:         return $\phi_{join} := \langle \texttt{JOIN}, s, nonce\rangle_\sigma$
17:     end if
18:
19:  function TEEvotejoin($acc_{sync}(s), J$)
       where $acc_{sync}(s)$ is $\langle \texttt{ACC|SYNC}, s, v, h\rangle_{\sigma_j}$
20:     // only allow voting on "future" views:
21:     if $s = session+1 \wedge syncing$ then
22:         return $\phi_{voted(session)} := \langle \texttt{VOTE}, s, v, h, J\rangle\sigma$
23:     end if
24:
25: // View progressing mechanism
26: function TEErejoin($\phi_{cert(s)}$) where $\phi_{cert(s)}$ is $\langle \texttt{VOTE}, s, v, h, J\rangle_{\vec{\sigma}^{\mathcal{F}+1}}$
27:     $c_1 := (id, nonce) \in J \wedge s > session$
28:     $c_2 := (session+1 = s) \wedge id \notin J$
29:     if $c_1 \vee c_2$ then // $c_1$ for "safety" and $c_2$ for "liveness"
30:         $session := s$
31:         $view := v$
32:         $prepv := v$
33:         $syncing$ := false
34:         $initialized$ := true
35:         return $\phi_{store} := \langle \texttt{PCOM}, s, v, h, H_\perp\rangle_\sigma$
36:     end if
37:

Accumulator from Fig. 2b
  function TEEaccum($\phi, \vec{\phi}$) where $\phi$ is $\langle \texttt{TAG}, p\rangle_\sigma$ and $\vec{\phi}$ has size $\mathcal{F}$
      if all certificates are from different replicas $\wedge \forall \phi' \in \vec{\phi}. \phi' \le \phi$ then
          return $acc := \langle \texttt{ACC|TAG}, p\rangle_{\sigma'}$
      end if
```

Fig. 3: Aegis: TEE-aided view synchronizer and rollback prevention

the others are rejected. This prevents multiple TEE clones from joining consensus simultaneously.

Replicas periodically call wish_to_advance() to initiate synchronization. This triggers a call to the TEE function TEEsync (l. 6, Fig. 3b), with the replica passing in the last store certificate it produced during consensus. A store certificate guarantees that a block proposed by a leader in a given session and view has been verified and accepted (i.e., committed but not necessarily yet decided).

TEEsync only generates a certificate if the TEE was *initialized* with a current nonce (indicating that the TEE is active and not restarting), and if its session and view numbers match the ones in the store certificate (ensuring consistency between the replica's TEE state and the provided store certificate, i.e., this store certificate was the last one generated by the TEE). If all conditions hold, the TEE sets an internal flag *syncing* to *true* (indicating that this replica is now committing to synchronizing), and outputs a certificate $\phi_{sync(s)}$ to synchronize and transition from session s to session $s+1$. Setting the *syncing* flag to true forces the replica to stop participating in the normal consensus protocol while synchronization is in progress, preventing it from accidentally preparing new blocks in the old session during the transition. The replica sends this certificate to the leader of session s. If no reply from the leader is received after some time, the replica re-sends the synchronization request to the next leader (l. 35, Fig. 3a), and repeats this for $\mathcal{F}$+1 views to ensure liveness. A reply from the leader is a $acc_{sync}(s)$, also called a TC for “Time Certificate” in Cogsworth.

Once a leader collects a quorum of $\mathcal{F}$+1 synchronization requests (l. 12, Fig. 3a), it creates a TC of the form $acc_{sync}(s)$ out of those requests by calling the accumulator (l. 17, Fig. 3a), and broadcasts this TC to all replicas. If a TC was already received, the leader simply relays it.

The $syncing$ flag serves a critical role in maintaining protocol safety. Without this safeguard, replicas might initiate synchronization for a triple (s, v, h), h being the latest block prepared at view v, while concurrently participating in consensus during session s. If a consensus decision advances to an extended block b' in view v+1 within the same session, Byzantine participants can issue synchronization messages for both (s, v, h) and $(s, v{+}1, \mathbf{H}(b'))$, undermining the accumulator's ability to ensure the uniqueness of the leader's TC.

In session s, leaders (whose IDs belong in $[s, s{+}\mathcal{F}]$) may concurrently generate TCs to ensure liveness, possibly from different synchronization requests. These requests target session s+1 but may differ due to delays or malicious faults. To resolve discrepancies, the stateless accumulator (l. 9, Fig. 2b) selects the request with the highest view, producing a TC $acc_{sync}(s)$ that preserves committed blocks.

Safety is preserved even with concurrent synchronization runs. The request selected by the accumulator defines membership uniquely through its view, guiding join computations and avoiding conflicting views. The highest view encoded in the TC specifies the join requests used to compute the new replica membership (l. 25, Fig. 3a). This uniqueness is essential to avoid diverging memberships across replicas and is reinforced by consensus-based voting on join requests. While the TC governs membership changes, session block content determines the operational semantics. Thus, replicas defer processing synchronization requests and TCs until they verify the presence of all preceding blocks in the ledger, fetching any missing ones if necessary (ll. 3 and 14, Fig. 4a).

Upon receiving a TC (l. 22, Fig. 3a), a replica first disseminates it to all leaders of the corresponding session (l. 24, Fig. 3a) to prevent losing liveness due to faulty participants. Each replica subsequently computes the set $joins$ of join requests prepared in the session, based on the TC's view number (l. 25, Fig. 3a). From $joins$, an array J of replica IDs and nonces is extracted and passed, together with the TC, to `TEEvotejoin` (l. 28, Fig. 3a).

The function `TEEvotejoin` verifies that the session number in a certificate matches the expected next session (l. 21, Fig. 3b). If valid, it issues a vote certificate (l. 22, Fig. 3b) comprising the session number, view number, hash of the last prepared block h, and a list J of joining replica IDs and nonces. These certificates (1) authorize approved replicas to safely join, and (2) contain enough context, i.e., view v and hash value h, to derive a new store certificate (l. 35, Fig. 3b) from the original one that triggered synchronization (l. 6, Fig. 3b).

Leaders gather votes from `TEEvotejoin` (l. 39, Fig. 3a). Once $\mathcal{F}$+1 consistent votes are collected, they form a quorum certificate (QC) $\langle \texttt{VOTE}, s, v, h, J\rangle_{\vec{\sigma}}$ and broadcast it (l. 44, Fig. 3a). This QC guarantees at least one correct replica processed the synchronization logic and created a valid J.

Upon receiving a QC (l. 47), each node updates its TEE state (session and view number) by calling `TEErejoin` (l. 26). A replica joins the session if either: (a) it is a joining replica, i.e., its ID and nonce are in J, and s is larger than its local session (condition c_1, l. 27, Fig. 3b); or (b) it is a continuing replica, i.e., its session is s−1 and it is not in J (condition c_2, l. 28, Fig. 3b).

When joining upon such a QC, both $view$ and $prepv$ are set to (s, v), and $syncing$ is set to `false` to indicate completion. A new store certificate $\langle \texttt{PCOM}, s{+}1, v, h\rangle_{\sigma'}$ (l. 35, Fig. 3b) is produced, enabling view transitions in the next session.

Replicas adopt the QC view both inside and outside the TEE. Since QC views may be lower than local views but still greater than or equal to the last prepared view, replicas may “rollback” to v, e.g., after timeouts where only some replicas incremented to v+1. This rollback is safe due to the incremented session and synchronized ledger state.

B. TEE Membership Recovery

In extreme failure scenarios, progress in the main consensus protocol may stall if too many TEEs from the last agreed membership simultaneously crash. Specifically, if u+1 or more TEEs fail simultaneously, f Byzantine nodes can prevent the formation of a quorum of $f{+}u{+}1$ TEEs by remaining silent. Therefore, to prevent loss of liveness, we design a recovery protocol that does not depend on any TEE-based operations.

Recovering from this liveness failure poses two critical non-trivial challenges. First, the system must establish agreement on a new valid membership configuration that enables quorum formation for safe decision-making. Second, this reconfiguration must be derived from a minimal subset of surviving nodes that retain the most recent, coherent system state. This state must be reliably propagated and adopted by the new membership to preserve consensus continuity and prevent forks or inconsistencies in future decisions.

To recover from such a loss of liveness, which can occur before or after GST, each replica monitors for prolonged consensus inactivity, using a timeout of $\mathcal{F}$+1 unsuccessful views. When detecting a timeout, a replica broadcasts a trigger for TEE membership election and halts participation in consensus. Once f+1 such votes are received, a correct replica joins the recovery process, votes to initiate it, and exits the main consensus loop. After GST, this liveness-loss condition only happens if consensus progress remains blocked, but eventually all correct replicas will converge on this election process.

The TEE membership election process must fulfill two goals: (i) establish a unique new TEE membership; and (ii) ensure this membership extends the latest prepared block, avoiding conflicts with committed history.

To prevent replay of old blocks after recovery that might lead to conflicts, session numbers are embedded in recovery messages. Nodes wait for $f{+}u{+}1$ messages with a fresh membership ID (a nonce) before transitioning. These nodes do not vote on blocks and thus form a safe quorum to initiate recovery.

Once in recovery mode, each replica reliably broadcasts its view number, TEE nonce, and latest known prepared block. Replicas retain the freshest such message from each peer. To determine the latest prepared block, each replica collects $2f{+}u{+}1$ signed responses. While Damysus requires $f{+}u{+}1$ for quorum, intersection with Byzantine replicas may lead to stale QCs. Messages contain signed QCs, including a reference to previous memberships ensuring consistency. A necessary condition for obtaining the freshest QC is $2f{+}u{+}1 \leq N{-}f$. This leads to $f \leq u$, a bound required for safe recovery.

To ensure that only one TEE membership is elected, a quorum size of $Q_m = \lceil \frac{3}{2} f \rceil {+} u {+} 1$ replicas must agree. This guarantees intersection between any two quorums in at least $f{+}1$ replicas, sufficient to break ties and avoid conflicts.

This bound is derived as follows:

- For any two sets of size Q_m to intersect in at least $f{+}1$ replicas:
$$2Q_m - (f{+}1) \geq N$$
- Since the system size satisfies $N \geq 2f{+}2u{+}1$:
$$2Q_m{-}(f{+}1) \geq 2f{+}2u{+}1$$
- Rearranging yields:
$$2Q_m \geq 3f{+}2u{+}2 \quad \Rightarrow \quad Q_m \geq \left\lceil \frac{3}{2} f \right\rceil {+}u{+}1$$

This quorum ensures that any pair of election quorums overlap, preserving agreement and preventing divergence in membership selection.

Upon initiation of membership recovery, replicas:

- Switch to TEE-only voting: they continue running the partially synchronous consensus but now exclusively vote on TEE membership proposals.
- Require a valid proposal: a correct leader constructs a proposal with (i) the latest prepared block (carrying a QC with $2f{+}u{+}1$ signatures), and (ii) a proposed nonce per QC replica.

Nodes only vote if the proposed nonce matches their intended TEE state. Once a TEE membership is committed, a node agrees to resume voting on blockchain blocks and on membership joins messages.

The rest of the recovery procedure is outlined as follows.

1) **Replica Broadcasts**. Each replica uses a reliable broadcast to transmit to all others the following information:
   - Its current TEE nonce
   - The latest view and session numbers where it saw a committed block
   - A monotonically increasing counter indicating the current membership recomputation round
2) **Consensus on Nonces**. Upon receiving $N{-}f$ nonce messages (suggesting at least $f{+}1$ correct sources), each replica joins up to N binary consensus instances:
   - For each replica, agree on its nonce or decide $\perp$
   - Nonces not delivered are treated with $\perp$ (absence or invalidity)
3) **Finalizing Membership**.
   - All nodes reach agreement on which nonces to include
   - If too few (less than a quorum) valid (non-$\perp$) nonces are chosen, membership must be recomputed
4) **Synchronizing Pacemakers**. Replicas that are part of the newly agreed membership adopt the smallest view number strictly greater than the highest one delivered via broadcast.

V. Pallas: A Rollback-Resilient BFT Consensus

A. Protocol Description

Fig. 4 presents `Pallas`, which implements rollback-prevention safeguards on top of the Damysus [6] consensus protocol. In the original Damysus protocol, replicas vote on a tuple (v, h), representing the view number and block hash. Our modified version extends this to a triple (s, v, h) by incorporating the session number s, which helps distinguish progress across sessions and prevent replays.

Blocks contain client transactions and any pending join requests, the latter treated as special transactions from replicas. By embedding join requests directly into proposed blocks (l. 8, Fig. 4a), we eliminate the need for a separate membership voting round. Thus, the protocol avoids maintaining or voting on a distinct "join request list", any relevant membership changes are already captured within the block itself.

a) Prepare phase: New-view messages from backups to the leader now include a session number (l. 3, Fig. 4a) alongside the current view, last prepared view, and the hash of the last prepared block. Upon receiving a quorum of new-view messages, the leader invokes the TEE accumulator `TEEaccum` over these messages to generate a new-view certificate acc_{nv} (l. 7, Fig. 4a), attesting to the latest view. It then proposes a new block extending the certified block hash and includes any unprepared join requests as special transactions (l. 8, Fig. 4a). The new block's hash is passed to `TEEprepare` for signing by the leader's TEE, generating a session- and view-unique vote (l. 10). To ensure at most one proposal per view, `TEEprepare` checks that *prepared* = `false`, sets it to `true`, and allows reset only via `TEEnewview` upon view increment.

Backups validate the leader's proposal, which includes the proposed block b, prepare certificate ϕ_{prep}, and accumulator certificate acc_{nv}. They verify that any join request J in b carries a session number higher than prior joins from the same node (l. 19, Fig. 4a), that b extends the block hashed in acc_{nv}, and that its hash matches the one in ϕ_{prep}. Only if all conditions hold will backups invoke `TEEprepare` to vote for b.

(a) Non-trusted code at replica i

Common variables from Fig. 2a

- $view := 0$
- $session := 0$
- ϕ_{store}
- $received\_joins := \emptyset$
- $sessions$
- $blocks$

```
1:  // prepare phase
2:  as a leader
3:      wait for $\mathcal{F}+1$ new-view certificates of the form $\langle \texttt{NV}, session, view, v', h' \rangle_\sigma$ such that all $b' \preceq h'$ have been received // + fetch missing blocks // $v', h', \sigma$ can vary
4:      $\phi_{nv}$ := certificate in the new view certificates with highest $v'$
5:      $\phi_{nv}$ is of the form $\langle \texttt{NV}, session, view, v', h' \rangle_\sigma$
6:      $\vec{\phi}_{nv}$ := $\mathcal{F}$ other certificates
7:      $acc_{nv} := \texttt{TEEaccum}(\phi_{nv}, \vec{\phi}_{nv})$
8:      $b := \texttt{createLeaf}(h'$, client transactions & $received\_joins)$
9:      $blocks[view] := b$
10:     $\phi_{prep} := \texttt{TEEprepare}(\mathbf{H}(b))$
11:     send $\langle b, \phi_{prep}, acc_{nv} \rangle$ to all backups
12:
13: as a backup
14:     wait for $\langle b, \phi_{prep}, acc_{nv} \rangle$ from the leader such that all $b' \preceq b$ have been received // + fetch missing blocks
15:     $acc_{nv}$ is $\langle \texttt{ACC|NV}, session, view, v', h' \rangle_\sigma$
16:     $\phi_{prep}$ is $\langle \texttt{PREP}, session, view, h \rangle_{\sigma'}$
17:     $\vec{\phi}_{join}$ := the join requests in $b$ // possibly empty
18:     $joins$ := join requests in $blocks$ from beginning of session up to $v'$
19:     check that the sessions in $\vec{\phi}_{join}$ are strictly higher than the ones in $joins$ and in $sessions$
20:     if $b \succ h' \wedge \mathbf{H}(b) = h$ then
21:         $blocks[view] := b$
22:         send $\phi'_{prep} := \texttt{TEEprepare}(h)$ to leader
23:     end if
24:
25: // pre-commit phase
26: as a leader
27:     wait for $\mathcal{F}+1$ prepare certificates of the form $\langle \texttt{PREP}, session, view, h, v' \rangle_\sigma$ // $\sigma$s differ
28:     send $\vec{\phi}_{prep} := \langle \texttt{PREP}, session, view, h, v' \rangle_{\vec{\sigma}}$ to all
29:
30: all replicas
31:     wait for $\langle \texttt{PREP}, session, view, h, v' \rangle_{\vec{\sigma}}$ from the leader
32:     $\phi_{store} := \texttt{TEEstore}(\langle \texttt{PREP}, session, view, h, v' \rangle_{\vec{\sigma}})$
33:     send $\phi_{store}$ to leader
34:
35: // decide phase
36: as a leader
37:     wait for $\mathcal{F}+1$ store certificates of the form $\langle \texttt{PCOM}, session, view, h \rangle_\sigma$ // $\sigma$s differ
38:     send $\vec{\phi}_{store} := \langle \texttt{PCOM}, session, view, h \rangle_{\vec{\sigma}}$ to all
39:
40: all replicas
41:     wait for $\vec{\phi}_{store} := \langle \texttt{PCOM}, session, view, h \rangle_{\vec{\sigma}}$ from leader
42:     execute $b$ corresponding to $h$ & reply to clients
43:     for each node with a join request in $b$ remove its entry in $received\_joins$ if the session number in the request in $b$ is greater than or equal to the one in $received\_joins$
44:     // starting new view
45:     $view$++
46:     start_new_view()
47:
48: // new-view after a timeout
49: all replicas
50:     if timeout
51:     $view$++
52:     start_new_view()
53:
```

(b) TEE code

Common variables from Fig. 2b

- id
- $syncing := \texttt{false}$
- $view := 0$
- $session := 0$
- $prepv := 0$

```
1:  Common variables, plus:
2:  • $prepared := \texttt{false}$
3:
4:  function TEEnewview($\phi_{store}$) where $\phi_{store}$ is $\langle \texttt{PCOM}, s, v, h \rangle_{\sigma'}$
5:      if $s = session \wedge v = prepv$ then
6:          $view$++
7:          $prepared := \texttt{false}$
8:          return $\phi_{nv} := \langle \texttt{NV}, session, view, v, h \rangle_\sigma$
9:      end if
10:
11: function TEEprepare($h$)
12:     if $\neg prepared$ then
13:         $prepared := \texttt{true}$
14:         return $\phi_{prep} := \langle \texttt{PREP}, session, view, h \rangle_{\sigma'}$
15:     end if
16:
17: function TEEstore($\vec{\phi}_{prep}$) where $\vec{\phi}_{prep}$ is $\langle \texttt{PREP}, s, v, h \rangle_{\vec{\sigma}^{\mathcal{F}+1}}$
18:     if $s = session \wedge v = view \wedge \neg syncing$ then
19:         $prepv := v$
20:         return $\phi_{store} := \langle \texttt{PCOM}, session, view, h \rangle_\sigma$
21:     end if
22:
```

Accumulator from Fig. 2b

```
function TEEaccum($\phi, \vec{\phi}$) where $\phi$ is $\langle \texttt{TAG}, p \rangle_\sigma$ and $\vec{\phi}$ has size $\mathcal{F}$
    if all certificates are from different replicas $\wedge \forall \phi' \in \vec{\phi}.\phi' \leq \phi$ then
        return $acc := \langle \texttt{ACC|TAG}, p \rangle_{\sigma'}$
    end if
```

Fig. 4: `Pallas`: BFT consensus where replicas handle and verify the validity of TEE join requests, built on top of Damysus [6]

b) Pre-commit phase: The rest of the pipeline proceeds as in Damysus, with a small adjustment for rollback prevention. Once a proposed block b accumulates a quorum of votes, the leader disseminates a prepare certificate, prompting backups to emit pre-commit votes, ultimately leading to a commit.

The pre-commit step in Fig. 4 deviates slightly from standard Damysus. Specifically, `TEEstore` (l. 17, Fig. 4b) executes only when *syncing* is false.

Upon block commitment, each replica updates its local list of pending join requests (l. 43, Fig. 4a). Once a node's join request is accepted, earlier requests from that node are considered obsolete and discarded. However, any newer join request with a higher session number remains active for future processing, ensuring the protocol preserves the latest join.

B. Proof Intuition

Our system inherits foundational safety and liveness properties from two well-established components that it builds upon: the Damysus consensus protocol and the Cogsworth pacemaker. These components have been proven to be safe and live under partial synchrony and bounded Byzantine faults. We build on their guarantees while introducing new mechanisms to handle rollback-prone environments and dynamic replica recovery.

a) Intuition behind safety: Safety in our system is preserved through three key mechanisms:

- Consensus-based Membership Installation. Any change in replica membership, such as a TEE rejoining after a crash, is only accepted if agreed upon through consensus. This ensures that no conflicting or stale replica state can influence the protocol.

- Single-Voter Enforcement per Replica. Our view synchronizer, Aegis, guarantees that at most one trusted component per replica can vote in a given view. This prevents equivocation, even in the presence of rollback attacks or cloned TEEs
- Damysus Safety Guarantees. Since our protocol builds directly on Damysus, it inherits its Byzantine fault tolerance and quorum-based safety properties, ensuring that conflicting decisions cannot be made.

Together, these mechanisms prevent rollback-induced safety violations and ensure that all decisions are made by a consistent and authenticated set of replicas.

b) Intuition behind liveness: Liveness in our system is ensured through a combination of the following:

- Aegis Progress Guarantees. Because it builds on Cogsworth [22], Aegis guarantees timely view changes under partial synchrony, even when leaders are faulty. We adapt Cogsworth's claims 1 and 2 [22] to our Aegis design showing that all honest nodes eventually enter the same view within bounded time (see Appendix A for detailed proofs).
- Damysus Consensus Progress. Once in a common view, Damysus ensures that honest replicas can reach agreement, provided a correct leader is eventually selected.
- Recovery Protocol for Crashed TEEs. We introduce a secure rejoin mechanism that allows crashed replicas to safely catch up and re-enter the system without violating safety. This dynamic recovery ensures that liveness is not lost even under excessive TEE crashes, i.e., when more than $\mathcal{F}$ TEEs simultaneously crash.

By tightly coordinating the consensus layer, view synchronizer, and recovery logic, we ensure that honest replicas make progress even under frequent crashes and adversarial conditions.

VI. Performance Evaluation

A. Implementation

Pallas and Aegis are implemented in C++. Pallas is built on top of Damysus [6], whose code is publicly available[4], while we implemented Aegis from scratch based on the Cogsworth [22] paper. Like Damysus we use Intel SGX [23] TEEs to run trusted services. While SGX has known security vulnerabilities [25], our implementation focuses on evaluating the performance of Pallas with and without rollback attacks. Moreover, our trusted services are generic enough to be potentially implemented in other trusted execution environments, such as AMD SEV [2] or ARM TrustZone [26], if needed. Replicas use ECDSA signatures with prime256v1 elliptic curves (available in OpenSSL [27]), and are connected using Salticidae [28].

B. Baselines

We evaluate the performance of Pallas and compare it with the following state-of-the-art protocols:

[4]https://github.com/vrahli/damysus

- **Basic-Damysus+ROTE**: A version of Damysus [6] with $2(f+u)+1$ nodes that uses ROTE [13] to replicate the state of trusted components, in particular every time TEE counters are incremented.
- **Flexi-Basic-Damysus**: A version of Flexi-BFT [12] with $3f+1$ nodes that execute Damysus as its consensus protocol and Cogsworth [22] as its synchronizer.
- **Basic-Damysus**: A version of Damysus with $2(f+u)+1$ nodes and that uses Cogsworth [22] as its synchronizer.
- **Achilles**: A version of Achilles [18] with $2f+1$ nodes that execute a variant of Damysus as its consensus protocol and a classical exponential backoff mechanism as its synchronizer.

C. Settings

We deploy Pallas and our baselines on AWS EC2 machines with one t2.micro instance per node. Blocks contain 400 transactions, and use payloads of 0 B and 256 B. In addition to the payload, a transaction contains 2×4 B for metadata (a client id, and a transaction id), as well as the hash value of the previous block of size 32 B, thereby adding 40 B to each transaction in addition to its payload. For example, the experiments with payloads of 0 B involve blocks of size 400×40 B=15.6 KB. Payloads of 0 B are used to evaluate only the protocols' overhead, while payloads of 256 B have been selected to observe the impact of increasing block size. This choice reflects the typical block size observed in blockchain systems such as Ethereum [29]. With 256 B payloads, blocks are then of size $400 \times (256+40)$ B=115.6 KB, which as shown below leads to a significant latency increase. We measure throughput and latency in various scenarios, and report the average of 100 repetitions. In our regional experiments, nodes are equally distributed over 4 AWS regions: Ireland, London, Paris, Frankfurt; while in our worldwide experiments, nodes are equally distributed over 8 AWS regions: North Virginia, Ohio, North California, Oregon, Ireland, London, Paris, Frankfurt.

D. Comparison Between the Protocols

a) Evaluation in failure-free runs: We first quantify Pallas's overhead by evaluating all baselines in stable settings, i.e., without TEE crashes or rollback attacks.[5] Since we do not evaluate TEE crashes here, we set $u = 0$. The baselines then have different system sizes: Flexi-Basic-Damysus requires $3f+1$ nodes, while Basic-Damysus+ROTE, Basic-Damysus, and Pallas require $2f+1$ nodes. We plot the throughput (in Kops/s) and the latency (in ms) of each protocol in function of f with 0 B-payloads (Fig. 5 and Fig. 7) and with 256 B-payloads (Fig. 6 and Fig. 8). As expected, the throughput of all protocols decreases when the system size increases, while their latency slightly increases. The performance of Pallas and Basis-Damysus are the best, and are very similar to each other, which demonstrates Pallas's negligible overhead in stable regimes. Furthermore, Pallas outperforms Flexi-Basic-Damysus and Basic-Damysus+ROTE in all scenarios. According to Fig. 5 (EU experiment with 0 B payloads) Flexi-Basic-Damysus's

[5]We omit Achilles in these experiments as Achilles is merely Damysus in stable settings.

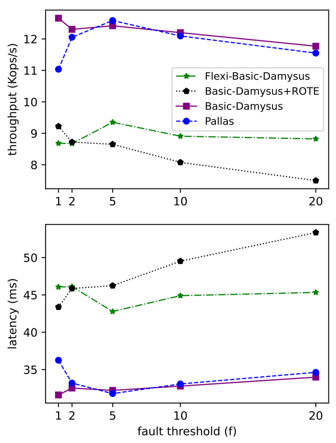

Fig. 5: Throughput and latency of all protocols under stable settings with 0 B-payloads (EU).

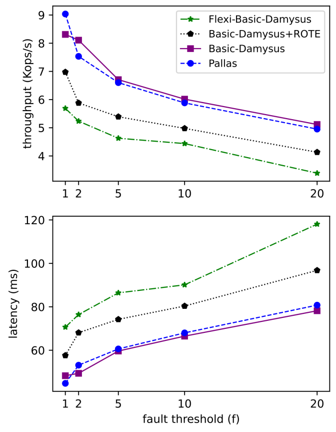

Fig. 6: Throughput and latency of all protocols under stable settings with 256 B-payloads (EU).

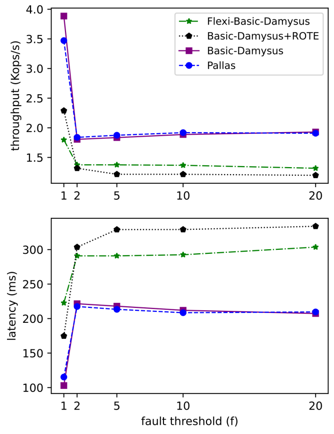

Fig. 7: Throughput and latency of all protocols under stable settings with 0 B-payloads (Worldwide).

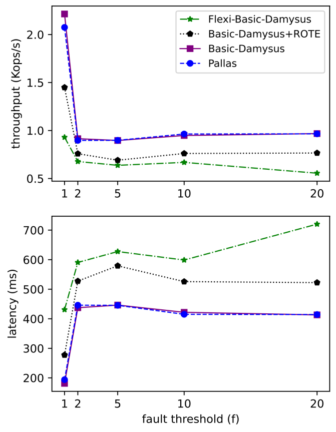

Fig. 8: Throughput and latency of all protocols under stable settings with 256 B-payloads (worldwide).

throughput is 33% lower than Pallas's, while its latency is 25% higher, and Basic-Damysus+ROTE is the slowest protocol, with a 41% lower throughput than Pallas's, while its latency is 29% higher. According to Fig. 6 (EU experiment with 256 B payloads) Flexi-Basic-Damysus's throughput is 45% lower than Pallas's, while its latency is 31% higher, and Basic-Damysus+ROTE is the slowest protocol, with a 24% lower throughput than Pallas's, while its latency is 18% higher. Furthermore, as we can see in the worldwide experiments, the added network latency exacerbates the difference between the protocols. According to Fig. 7 (worldwide experiment with 0 B payloads) Flexi-Basic-Damysus's throughput is 50% lower than Pallas's, while its latency is 32% higher, and Basic-Damysus+ROTE is the slowest protocol, with a 52% lower throughput than Pallas's, while its latency is 34% higher. According to Fig. 8 (worldwide experiment with 256 B payloads) Flexi-Basic-Damysus's throughput is 63% lower than Pallas's, while its latency is 36% higher, and Basic-Damysus+ROTE is the slowest protocol, with a 29% lower throughput than Pallas's, while its latency is 22% higher.

b) Evaluation under rollback attacks/mobile crashes: The three protocols Flexi-Basic-Damysus, Basic-Damysus, and Basic-Damysus+ROTE cannot tolerate mobile crashes, compromising liveness under our strong attacker model. Flexi-Basic-Damysus and Basic-Damysus+ROTE prevent rollback attacks but loses liveness under mobile crashes, while Basic-Damysus does not prevent rollback attacks at all. Therefore, we compare against Basic-Damysus+ROTE as well as Achilles, which adds rollback prevention to Damysus and enables nodes to recover and rejoin after crashes. We exclude Flexi-Basic-Damysus and Basic-Damysus because neither supports recovery after crashes.

To quantify the impact of rollback attacks and crashes/restarts on Pallas, Basic-Damysus+ROTE, and Achilles, we assess performance degradation as crashed TEEs rejoin the system. We exclude mobile crashes since only Pallas retains liveness in heavy mobile crashes. Fig. 9 illustrates scalability as the number of rejoining nodes increases within a system of size $2\mathcal{F}+1 = 2(f+u)+1 = 7$, where $f = 0$ and $u = 3$.

In this setup, the synchronizer is executed every $\mathcal{F}+1$ views, i.e., every 4 views, and replicas that have crashed and recovered rejoin at the beginning of each session (i.e., immediately following a synchronizer run). We evaluate the performance of Pallas, Basic-Damysus+ROTE, and Achilles under varying degrees of recovery pressure by measuring throughput and latency as the number of restarting nodes per session ranges from 0 to 3.

Fig. 9 shows that Pallas's performance decreases gradually as more nodes are rejoining. Nonetheless, the degradation remains modest: throughput declines by up to 13% (from 13.6 Kops/s down to 11.78 Kops/s) and latency increases by no more than 17% (from 29 ms up to 34 ms). These results demonstrate that Pallas maintains robust performance and operational stability even under frequent TEE restarts, validating its resilience against rollback-induced rejoin overhead. In contrast, Basic-Damysus+ROTE and Achilles exhibit an exponential decay both in throughput an latency. This is attributed to timeouts resulting from leader crashes in those algorithms, which is circumvented in Pallas's design.

As illustrated in Fig. 10, a larger system ($2\mathcal{F}+1 = 2(f+u)+1 = 41$ replicas with $f = 0$ and $u = 20$) results in more pronounced performance degradation under rejoin stress. As the number of rejoining replicas increases from 0 to 20, throughput declines by up to 69% (from 68 Kops/s to 21 Kops/s), while latency surges by as much as 200% (from 633 ms to 1.9 s). Despite this significant degradation, Pallas continues to outperform existing protocols, which would eventually lose liveness under increasing restarts. These results demonstrate Pallas's ability to preserve progress and guarantee termination in rollback-heavy scenarios.

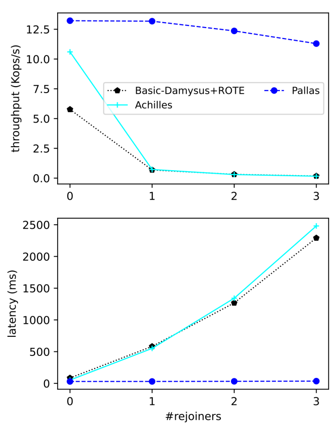

Fig. 9: Throughput and latency of `Pallas` with joining nodes, 0 MB payloads, $f = 0$ and $u = 3$.

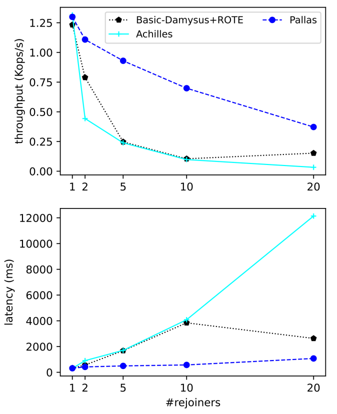

Fig. 10: Throughput and latency of `Pallas` with joining nodes, 0 MB payloads, $f = 0$ and $u = 20$.

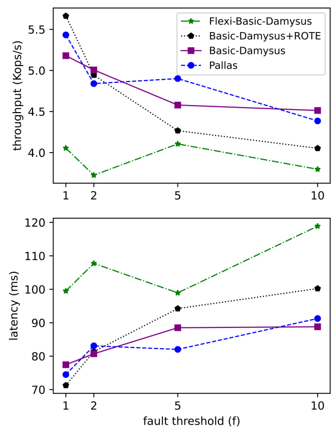

Fig. 11: Throughput and latency of all protocols with 0 B-payloads and 5% message loss (EU).

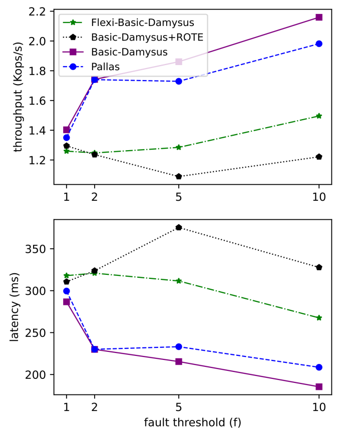

Fig. 12: Throughput and latency of all protocols with 256 B-payloads and 5% message loss (EU).

E. Evaluation Under Message Losses

Fig. 11 and Fig. 12 compare the protocols with 0 B and 256 B payloads, respectively, in the EU regions and with a heavy message loss of 5% [30]. As the figures show, `Pallas` and Basic-Damysus have similar performance, superior to that of Flexi-Basic-Damysus and Basic-Damysus+ROTE with increasing system size. While this difference between the protocols follows a trend similar to that of Fig. 5 and Fig. 6, it is however not as pronounced as in those figures because of the re-transmission of lost messages.

VII. Related Work

A. Rollback Attacks and Defenses

TEE rollback attacks lead an enclave to adopt a previous state or launch several instances of the same enclave [11]. To ensure the freshness of a TEE state, some works relied on non-volatile memory associated to a TEE [31–33]. In these approaches, the rate at which a TEE can save its state is limited by wear of the non-volatile memory, or there is a dependency on specialized hardware or an uninterruptible power supply [34]. Another approach consists in storing freshness information in a separate trusted server, or in a set of servers that can only be partially corrupted by an adversary. ROTE [13] replicates the state of a modified enclave among other enclaves, and allows a restarting enclave to detect a rollback attack and recover its state. ENGRAFT [14] and TEEMS [34] run a crash-tolerant consensus algorithm inside TEEs and design mechanisms to tolerate rollback attacks. NARRATOR [15] aims to allow state continuity protection for cloud TEEs by leveraging an external blockchain. In particular, NARRATOR addresses a vulnerability of ROTE during its initialization time by using a blockchain. Nimble [16] provides a TEE-based service for cloud users, applications run inside TEEs and Nimble enforces that the system prevents rollback attacks. It uses an external consensus protocol to update the state of the TEEs. CloneBuster [19] allows an enclave to detect whether a cloned enclave is running on the same machine. Rollbaccine [35] detects that a rollback happened by maintaining replicated versions of a host's disk and, upon detecting a rollback attack, reverts an application to a state it could have reverted to in the absence of rollback attacks. Rollbaccine does not tolerate Byzantine faults. We present the first defense against rollback attacks for BFT consensus algorithms that avoids replicating a TEE state.

B. Hybrid BFT protocols

Hybrid Byzantine Fault Tolerant (BFT) protocols may integrate trusted components for several purposes, including reconfiguration [36–38], proactive recovery [39], fault tolerance or performance. Several techniques have been proposed in the literature to leverage trusted hardware, such as trusted logs [40–42], attested append-only memory (A2M) [43], and several trusted incrementers (TrInc) [44]. MinBFT [5] uses a single trusted monotonic counter to reduce the number of communication phases in PBFT with improved resilience. CheapBFT [4] and ReBFT [45] further optimized performance by enabling the system to operate optimistically with only $f{+}1$ active replicas, keeping the remaining f replicas passive during normal-case execution. Hybster [46] extended this line of work by enabling parallel instance execution using TrIncX, a variant of the trusted monotonic counter. Similarly, FastBFT [47] adopted an optimistic approach with $f{+}1$ active nodes and incorporated a TEE-based secret sharing mechanism to reduce message complexity. More recently, TBFT [48] moves away from PBFT's all-to-all $O(N^2)$ communication to a leader-centric $O(N)$ design similar to HotStuff. It uses trusted counters and secure message sharing to build quorum certificates from just $f{+}1$ replicas. Damysus [6] defines two trusted components, the Checker and the Accumulator, to improve the resilience and the latency of HotStuff [9]. FlexiTrust [12] uses $3f{+}1$ replicas to support parallel voting on several blocks. Zhao et al. [50] propose a TEE-based leaderless protocol. None of these protocols tolerate rollback attacks. Achilles [18]

TABLE I: Comparison with related TEE-aided BFT systems

Protocol	**Rollback-Resilient**	**Clone-Resilient**	**Mobile Crash adversary**	**Min. # Nodes**	**Synchronizer**
TEE-aided BFT [5, 6, 49, 50]	✗	✗	✗	$2f+1$	Exp. backoff
TEE-aided BFT + ROTE [13]	✓	✓	✗	$2f+2u+1$	Exp. backoff
Flexi-BFT [12]	✗	✗	✗	$3f+1$	Exp. backoff
Achilles [18]	✓	✗	✗	$2f+1$	Exp. backoff
`Pallas` - Byz. only (u=0)	✓	✓	✗	$2f+1$	`Aegis`
`Pallas` - Byz.+static crash	✓	✓	✗	$2f+2u+1$	`Aegis`
`Pallas` - Byz.+mobile crash	✓	✓	✓	$2f+2u+1$ w. $u \geq f$	`Aegis`

and Engraft [14] explore how to tolerate a weaker form of rollback attacks, i.e., replaying a previously recorded state to the TEE, thus violating freshness, without considering TEE forking, where a Byzantine replica creates multiple TEE clones Appendix C presents a detailed comparison between our solution and both protocols. In addition, all these protocols would lose liveness under the mobile crash adversary.

Table I compares `Pallas` against prior work. For clarity, we distinguish between rollback via state replay (i.e., rolled-back TEEs) and TEE forking (i.e., cloned TEEs). `Pallas` is the only protocol that comprehensively addresses the rollback threat landscape, covering both attack vectors under our advanced threat model. Moreover, `Aegis` is the only synchronizer that remains correct with just $2f+1$ replicas.

C. View Synchronization

A view synchronizer is a crucial component of partially synchronous consensus protocols [21] that ensures that all correct nodes stay in a view long enough to reach agreement when the leader is honest. Theoretical bounds dictate that any deterministic Byzantine consensus algorithm requires $O(n^2)$ messages [51], and that they have a $O(n)$ latency lower bound [52]. PBFT [8]'s synchronizer uses an exponential back-off mechanism for liveness and incurs $O(n^2)$ expected message complexity, $O(\delta) = O(1)$ expected latency and unbounded worst-case latency. Naor et al. [22] described a minimal straw-man synchronizer with no messaging and unbounded latency. FastSync [53] relies on Bracha's reliable broadcast and has similar performance. HotStuff [9] assumes a synchronizer (called pacemaker) that uses exponentially increasing timeouts, though its design was left unspecified in the original publication and later elaborated upon in the LibraBFT white paper [54]. After GST, all correct replicas converge to a given view within a bounded delay δ, incurring $O(n^2)$ message complexity. In the worst case, however, the complexity may escalate to $O(f \cdot n^2) = O(n^3)$. Cogsworth [22] achieves linear message complexity with constant expected latency. In failure-free cases, the protocol maintains $O(n)$ message complexity and $O(1)$ latency. Under a coordinated attack by f Byzantine leaders, complexity degrades to $O(n^3)$ with $O(n)$ latency. Other view synchronizers, like Lewis-Pye's synchronizer [55], approach the theoretical message complexity lower bound by grouping views into epochs of $f+1$ consecutive views. FastSync [53] is a view-synchronizer that does not rely on digital signatures and instead relies on Bracha's reliable broadcast [56]. RareSync [57] meets the theoretical worst-case message bound of $\Theta(n^2)$ and synchronizes within $O(f)$ after GST. Fever [58] balances optimistic latency with worst-case complexity of $O(fn+n)$. Lumiere [59] adapts to faults, with message complexity proportional to the actual fault count f'. Our view synchronizer, `Aegis`, is built on top of Cogsworth and is, to the best of our knowledge, the first to operate with TEEs. We also highlight the need for a tight interplay between the synchronizer and consensus protocols to efficiently tolerate rollback attacks.

D. Mobile Adversary

Schmiedel et al. [20] propose a mobile crash Byzantine adversary model, where crashed nodes can recover without losing their states or messages. Crashed nodes may be temporarily unresponsive but are otherwise correct. We also consider a mobile rollback adversary, along with a static Byzantine adversary, however, the TEE of the crashed replicas may lose their state and cannot recover without a recovery mechanism. The classic static threshold model that "tolerates f Byzantine nodes", who can be Byzantine or victim of a rollback attack, does not properly represent a mobile attacker's capacity. Under this model, it it is not necessary to introduce additional defense mechanisms against rollback attacks, as the remaining $f+1$ correct nodes always guarantee safety and liveness. However, under mobile rollback attacks crashed machines must be allowed to rejoin the system. Our mobile adversary can rollback the TEE of a selected node, and later move to a different node to launch the same attack. However, if we allow the attacker to move faster than the required time a node needs to recover, then the attacker can stall the liveness of the system by corrupting more nodes than what the system can tolerate. To model this behavior, we consider a λ-mobile adversary, where λ is a security parameter representing the pace of the attacker to move its rollback attack from one node to the other. For example, λ can be represented by a specific duration such as an upper bound of communication delay Δ. Hence, the resiliency provided by a BFT protocol against a λ-mobile adversary is inversely proportional to the value of λ: the smaller the value of λ the higher the resiliency achieved. See Appendix B for our analysis of λ.

VIII. Conclusion

This paper presents a novel approach to addressing rollback vulnerabilities in TEE-aided BFT consensus protocols. By tightly coupling a consensus algorithm with a purpose-built synchronizer, we ensure that each replica maintains at most one active TEE instance per view. Our protocol supports safe recovery from TEE crashes without requiring expensive state replication or increasing the number of replicas beyond $2f+2u+1$, thereby preserving efficiency. Through careful session management and the integration of join requests into consensus, our design enforces safety by preventing equivocation and ensures liveness under mobile rollback and Byzantine adversaries. Extensive evaluations demonstrate that our protocol performs competitively with, and often outperforms, existing solutions, even as the number of restarting TEEs increases.

Acknowledgment

This work was partially supported by the Engineering and Physical Sciences Research Council (EPSRC) grant number EP/W034514/1, and by the Australian Research Council (ARC) under projects DE210100019. We gratefully acknowledge Nils van den Honert, whose MSc thesis [60] at TU Delft was a preliminary exploration of rollback attacks on Damysus.

References

[1] V. Costan and S. Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016).

[2] A. Sev-Snp. "Strengthening VM isolation with integrity protection and more". In: *White Paper, January* 53.2020 (2020), pp. 1450–1465.

[3] I. TDX. *Intel CPU Architectural Extensions Specification*. https://cdrdv2.intel.com/v1/dl/getContent/733582. 2021.

[4] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. "CheapBFT: resource-efficient byzantine fault tolerance". In: *EuroSys*. 2012.

[5] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE TC* 62.1 (2013), pp. 16–30.

[6] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. "DAMYSUS: streamlined BFT consensus leveraging trusted components". In: *EuroSys*. 2022.

[7] L. Zhao, H. Schmiedel, Q. Wang, and J. Yu. "Janus: Enhancing Asynchronous Common Subset with Trusted Hardware". In: *Annual Computer Security Applications Conference, IEEE ACSAC*. 2024, pp. 488–504.

[8] M. Castro and B. Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM TC*. 20.4 (2002), pp. 398–461.

[9] M. Yin, D. Malkhi, M. K. Reiter, G. Golan Gueta, and I. Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *PODC* (2019).

[10] D. Malkhi and K. Nayak. "Hotstuff-2: Optimal two-phase responsive bft". In: *Cryptology ePrint Archive* (2023).

[11] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. "Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory". In: *DSN*. 2017.

[12] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi. "Dissecting BFT Consensus: In Trusted Components we Trust!" In: *EuroSys*. 2023.

[13] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. "ROTE: rollback protection for trusted execution". In: *USENIX Security*. 2017.

[14] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang. "ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes". In: *CCS*. 2022.

[15] J. Niu, W. Peng, X. Zhang, and Y. Zhang. "NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud". In: *CCS*. 2022.

[16] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. T. V. Setty, and S. Singanamalla. "Nimble: Rollback Protection for Confidential Cloud Services". In: *OSDI*. 2023.

[17] A. Bessani, M. Correia, T. Distler, R. Kapitza, P. E. Veríssimo, and J. Yu. "Vivisecting the Dissection: On the Role of Trusted Components in BFT Protocols". In: *arXiv preprint arXiv:2312.05714* (2023).

[18] J. Niu, X. Wen, G. Wu, S. Liu, J. Yu, and Y. Zhang. "Achilles: Efficient TEE-Assisted BFT Consensus via Rollback Resilient Recovery". In: *EuroSys*. 2025.

[19] S. Briongos, G. Karame, C. Soriente, and A. Wilde. "No Forking Way: Detecting Cloning Attacks on Intel SGX Applications". In: *ACSAC*. 2023.

[20] H. Schmiedel, R. Han, Q. Tang, R. Steinfeld, and J. Yu. "Modeling Mobile Crash in Byzantine Consensus". In: *CSF*. 2024.

[21] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323.

[22] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman. "Cogsworth: Byzantine View Synchronization". In: *Cryptoeconomic Systems* 1.2 (2021).

[23] *SGX*. URL: https://software.intel.com/en-us/sgx.

[24] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. "OneShot: View-Adapting Streamlined BFT Protocols with Trusted Execution Environments". In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. IEEE, 2024, pp. 1022–1033.

[25] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, and F. Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *SP*. 2020.

[26] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM CSUR* 51.6 (2019), 130:1–130:36.

[27] *OpenSSL*. URL: https://www.openssl.org/.

[28] *Salticidae*. URL: https://github.com/Determinant/salticidae.

[29] *Etherscan*. URL: https://etherscan.io/chart/blocksize.

[30] S. Chowdhury and K. Fatema. "Analysing TCP performance when link experiencing packet loss". In: 2014.

[31] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. "Memoir: Practical state continuity for protected modules". In: *S&P*. 2011.

[32] R. Strackx, B. Jacobs, and F. Piessens. "ICE: A passive, high-speed, state-continuity scheme". In: *ACSAC*. 2014.

[33] R. Strackx and F. Piessens. "Ariadne: A minimal approach to state continuity". In: *USENIX Security*. 2016.

[34] B. Dinis, P. Druschel, R. Rodrigues, and I. Superior Técnico. "RR: A Fault Model for Efficient TEE Replication". In: *NDSS*. 2023.

[35] D. Chu, A. Balasubramanian, D. Bao, N. Crooks, H. Howard, L. E. Katahanas, and S. Ponnapalli. "Rollbaccine: Herd Immunity against Storage Rollback Attacks in TEEs [Technical Report]". In: *arXiv preprint arXiv:2505.04014* (2025).

[36] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. A. Schultz. "Automatic Reconfiguration for Large-Scale Reliable Storage Systems". In: *IEEE TDSC* 9.2 (2012), pp. 145–158.

[37] D. S. Silva, R. Graczyk, J. Decouchant, M. Völp, and P. Esteves-Verissimo. "Threat Adaptive Byzantine Fault Tolerant State-Machine Replication". In: *SRDS*. 2021.

[38] P. Kuznetsov and A. Tonkikh. "Asynchronous reconfiguration with byzantine failures". In: *DISC* (2022).

[39] M. Castro. "Practical Byzantine Fault Tolerance". Ph.D. MIT, Jan. 2001.

[40] A. Haeberlen, P. Kouznetsov, and P. Druschel. "PeerReview: Practical accountability for distributed systems". In: 2007.

[41] S. B. Mokhtar, J. Decouchant, and V. Quéma. "Acting: Accurate freerider tracking in gossip". In: *SRDS*. 2014.

[42] A. Diarra, S. B. Mokhtar, P.-L. Aublin, and V. Quéma. “Fullreview: Practical accountability in presence of selfish nodes”. In: *SRDS*. 2014.
[43] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. “Attested append-only memory: making adversaries stick to their word”. In: *SOSP*. 2007.
[44] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. “TrInc: Small Trusted Hardware for Large Distributed Systems”. In: *NSDI*. 2009.
[45] T. Distler, C. Cachin, and R. Kapitza. “Resource-Efficient Byzantine Fault Tolerance”. In: *IEEE TC* 65.9 (2016), pp. 2807–2819.
[46] J. Behl, T. Distler, and R. Kapitza. “Hybrids on Steroids: SGX-Based High Performance BFT”. In: *EuroSys*. 2017.
[47] J. Liu, W. Li, G. O. Karame, and N. Asokan. “Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing”. In: *IEEE TC* 68.1 (2019), pp. 139–151.
[48] J. Zhang, J. Gao, K. Wang, Z. Wu, Y. Lan, Z. Guan, and Z. Chen. “TBFT: Understandable and Efficient Byzantine Fault Tolerance using Trusted Execution Environment”. In: *arXiv preprint arXiv:2102.01970* (2021).
[49] S. Xie, D. Kang, H. Lyu, J. Niu, and M. Sadoghi. “Fides: Scalable Censorship-Resistant DAG Consensus via Trusted Components”. In: *arXiv preprint arXiv:2501.01062* (2025).
[50] L. Zhao, J. Decouchant, J. K. Liu, Q. Lu, and J. Yu. “Trusted hardware-assisted leaderless byzantine fault tolerance consensus”. In: *IEEE TDSC* 21.6 (2024), pp. 5086–5097.
[51] D. Dolev and R. Reischuk. “Bounds on information exchange for Byzantine agreement”. In: *J. ACM* 32.1 (1985), pp. 191–204.
[52] D. Dolev and H. R. Strong. “Authenticated algorithms for Byzantine agreement”. In: *SIAM Journal on Computing* 12.4 (1983), pp. 656–666.
[53] M. Bravo, G. Chockler, and A. Gotsman. “Making byzantine consensus live”. In: *Distributed Computing* 35.6 (2022), pp. 503–532.
[54] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. *State machine replication in the libra blockchain*. Tech. rep.
[55] A. Lewis-Pye. “Quadratic worst-case message complexity for state machine replication in the partial synchrony model”. In: *arXiv preprint arXiv:2201.01107* (2022).
[56] G. Bracha. “Asynchronous Byzantine agreement protocols”. In: *Information and Computation* 75.2 (1987), pp. 130–143.
[57] P. Civit, M. A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, and M. Vidigueira. “Byzantine consensus is Θ (n^2): the Dolev-Reischuk bound is tight even in partial synchrony!” In: *Distributed Computing* 37.2 (2024), pp. 89–119.
[58] A. Lewis-Pye and I. Abraham. “Fever: Optimal responsive view synchronisation”. In: *OPODIS*. 2023.
[59] A. Lewis-Pye, D. Malkhi, O. Naor, and K. Nayak. “Lumiere: Making Optimal BFT for Partial Synchrony Practical”. In: *PODC*. 2024.
[60] N. van den Honert. “Rollback protection in Damysus”. Master’s thesis. Delft University of Technology, 2024.
[61] W. Wang, S. Deng, J. Niu, M. K. Reiter, Y. Zhang, and Y. Org. “Engraft: Enclave-guarded Raft on Byzantine Faulty Nodes”. In: *CSS* (2022).

Appendix

A. Aegis: Proof of Correctness

We adapt key liveness claims from Cogsworth [22] to our protocol’s synchronizer design, ensuring timely view transitions under partial synchrony. The first lemma (Lemma 1) shows that if an honest node enters session s at time t and the leader of session s is honest, then all honest nodes enter session s by $t{+}4\Delta$. The second lemma (Lemma 2) generalizes this guarantee: even if the leader is faulty, all honest nodes enter session s by $t{+}2\Delta(f{+}2)$, assuming honest nodes relay synchronization messages to multiple leaders. This modification ensures full liveness despite adversarial behavior, improving upon Cogsworth’s requirement of $f{+}1$ honest voters.

Lemma 1. *After GST, if an honest node enters session s at time t, and $\texttt{leader}(s)$ is honest then all honest nodes enter session s by $t + 4\Delta$.*

Proof. Since an honest node enter session s at time t, there is an honest node i that *first* entered session s at time $t' \leq t$. Node i entered session s because it received a $\phi_{cert(s)}$ certificate from $\texttt{leader}(r)$ such that $s \leq r \leq s{+}\mathcal{F}$ (l. 47, Fig. 3a).

If $r = s$ then since $\texttt{leader}(s)$ is assumed to be honest, and has sent this certificate to i, it must have sent it to all nodes (l. 44, Fig. 3a). Therefore, all honest nodes receive this certificate and enter session s by $t{+}\Delta$.

If $r > s$ then $\texttt{leader}(r)$ must have received $\mathcal{F}{+}1$ $\phi_{vote(s)}$ messages (l. 39, Fig. 3a). One of those messages was sent by a honest node, say j, which much have received a corresponding $acc_{sync}(s)$ (l. 22, Fig. 3a) from $\texttt{leader}(r')$ for $s \leq r' \leq s{+}\mathcal{F}$. Because j is honest, it must also have sent this $acc_{sync}(s)$ to $\texttt{leader}(s)$ (l. 24, Fig. 3a). Because $\texttt{leader}(s)$ is honest, it forwards it by $t + \Delta$ to all other nodes, which send their $\phi_{vote(s)}$ to $\texttt{leader}(s)$ (l. 29, Fig. 3a) by $t + 2\Delta$. By $t + 3\Delta$ the leader receives $\mathcal{F}{+}1$ votes from the honest nodes and sends $\phi_{cert(s)}$ to all nodes (l. 44, Fig. 3a). Finally, by $t + 4\Delta$ all honest nodes receive this certificate (l. 47, Fig. 3a) and enter session s, which concludes the proof.

The following lemma is Claim 2 from [22] adapted to our setting. One key difference is that Claim 2 states that $f{+}1$ honest nodes ($\mathcal{F}{+}1$ correct nodes in our setting) enter view s (session s in our setting), which is not possible to achieve without altering the synchronizer (l. 24, Fig. 3a). With this modification, we obtain that all honest nodes, i.e., at least $\mathcal{F}{+}1$, enter session s.

Lemma 2. *After GST, when an honest node enters session s at time t, all honest nodes enter session s by $t + 2\Delta(f + 2)$.*

Proof. Because an honest node enters session s at time t, there is an honest node i that *first* enters session s at time $t' \leq t$. Node i entered session s because it received a $\phi_{cert(s)}$ certificate from $\texttt{leader}(r)$ such that $s \leq r \leq s{+}\mathcal{F}$ (l. 47, Fig. 3a).

If $\texttt{leader}(r)$ is correct, then we can conclude since $\texttt{leader}(r)$ has sent this certificate to all nodes (l. 44, Fig. 3a), and all correct nodes have received it (l. 47, Fig. 3a) and entered session s by $t + \Delta$.

If $\texttt{leader}(r)$ is faulty, it might have sent the certificate to some nodes only, potentially to i only. However, $\texttt{leader}(r)$ must have received $\mathcal{F}{+}1$ $\phi_{vote(s)}$ messages (l. 39, Fig. 3a). One of those messages was sent by a honest node, say j, which much have received a corresponding $acc_{sync}(s)$ (l. 22, Fig. 3a) from $\texttt{leader}(r')$ for $s \leq r' \leq s{+}\mathcal{F}$.

Let us pause here and consider what happens in Cogsworth. While Cogsworth requires a set H of $f{+}1$ honest nodes to have sent votes, we cannot do this here because leaders receive at most $f{+}1$ messages, from at least 1 correct node, and possibly f Byzantine nodes (given $u = 0$). In Cogsworth, if one of the nodes in H does not receive a QC (a message of the form $\phi_{cert(s)}$ in our setting) for $f{+}1$ rounds of the synchronizer (which might be the case for example if $\texttt{leader}(r)$ sends the QC to i only, followed by a similar behaviour from subsequent leaders), it must have sent by then a TC (a message of the form $acc_{sync}(s)$ in our setting) to a correct leader, which leads the process to completion, i.e., to entering view v (session s in our setting). Otherwise, all $f{+}1$ nodes have received a QC and have changed view.

Going back to our variant, since H can only be of size 1, we require honest nodes to not only relay $acc_{sync}(s)$ (l. 24, Fig. 3a) to $\texttt{leader}(s)$, but to $\{\texttt{leader}(s), \dots, \texttt{leader}(s{+}\mathcal{F})\}$. Therefore, one of these leaders must be correct, and we can conclude as in the proof of Lem. 2.

B. Analysis of the λ Parameter

To prove consensus liveness, we have assumed that the *mobile crash adversary* can crash up to u TEEs within any time interval of duration λ after GST. Parameter λ therefore bounds the adversary's *movement speed* between successive crashes. To guarantee that no more than u replicas are simultaneously crashed after GST, it is required that $\lambda \geq T_R$, where T_R is the maximum *TEE recovery time*, measured as the maximum time a crashed TEE that instantly recovers would require to commit a rejoin request and rejoin the system.

In the following, we compute the minimal tolerated λ value. For simplicity, this analysis counts only network communication steps and assumes no extra local processing delays, as typically we expect the local computation to be negligible compared to network latency across the globe.

We use the following notations:

- Δ is the upper bound on message transmission delays after GST.
- T_{commit} is the time required to commit one block in the consensus pipeline (prepare $\rightarrow$ pre-commit $\rightarrow$ decide).
- T_{sync} is the time required by the synchronizer (`Aegis`) to activate new joins once invoked.

Best Case Join Operation. In the best case, the leader is correct and the join request arrives exactly at the beginning of an execution of `Pallas`, just before an execution of `Aegis`, its view-synchronizer.

1) Join message broadcast: 1 step (Δ).
2) Commit join in block: 5 message steps ($T_{\text{commit}} = 5\Delta$) in `Pallas`'s three-phase consensus pipeline (Fig. 4), including communications to exchange the leader's proposal, backups' prepare votes, the leader's prepare certificate, replicas' store certificates, and the leader's $\vec{\phi}_{store}$.
3) Synchronizer run: 4 message steps ($T_{\text{sync}} = 4\Delta$) in `Aegis` (Fig. 3) to actualize the membership with the accepted join requests. This includes communications to exchange messages including replicas' $\phi_{sync(s)}$, the leader's $acc_{sync}(s)$, replicas' votes, and the leader's $\phi_{cert(s)}$.

Thus:
$$T_R^{\text{best}} = \Delta + T_{\text{commit}} + T_{\text{sync}} = 10\Delta \qquad (1)$$
and:
$$\lambda_{\text{min}}^{\text{best}} = 10\,\Delta. \qquad (2)$$

Worst Case Join Operation. If the join is committed just after a session starts, it will be activated at the *next* session boundary. With sessions happening every $F{+}1$ views, the maximum wait is F additional views. Let T_{view} be the post-GST time to finish one view (one block decision), we have
$$T_R^{\text{worst}} \;\approx\; T_{\text{commit}} + F \cdot T_{\text{view}} + T_{\text{sync}}. \qquad (3)$$
Since $T_{\text{commit}} \approx T_{\text{view}}$ in steady state and T_{sync} is small:
$$T_R^{\text{worst}} \;\approx\; (F+1) \cdot T_{\text{view}}, \qquad (4)$$
yielding:
$$\lambda_{\text{min}}^{\text{worst}} \;\approx\; (F+1) \cdot T_{\text{view}}. \qquad (5)$$

Measured Values from Our Evaluation. Our geo-distributed AWS experiments report *latency per decision*, i.e., end-to-end time to commit a block. We use these measured latencies as $T_{\text{view}} \approx T_{\text{commit}}$, with T_{sync} negligible in comparison.

With our small system ($N = 7, f = 0, u = 3$), we measured: $T_{\text{view}} \approx 29{-}34\,\text{ms}$ (baseline to rejoin-heavy).
$$\lambda_{\text{min}}^{\text{best}} \;\approx\; 29{-}34\text{ ms}, \quad \lambda_{\text{min}}^{\text{worst}} \;\approx\; 0.12{-}0.14\text{ s}.$$

With a larger system ($N = 41$, $f = 0$, $u = 20$), we measured: $T_{\text{view}} \approx 0.633{-}1.9\,\text{s}$ (baseline to rejoin-heavy).
$$\lambda_{\text{min}}^{\text{best}} \;\approx\; 0.63{-}1.9\text{ s}, \quad \lambda_{\text{min}}^{\text{worst}} \;\approx\; 13{-}40\text{ s}.$$

TABLE II: Minimum λ values from theoretical and measured analysis.

System size	F	T_{view}	$\lambda_{\text{min}}^{\text{best}}$	$\lambda_{\text{min}}^{\text{worst}}$
$N = 7$	3	29–34 ms	29–34 ms	0.12–0.14 s
$N = 41$	20	0.633–1.9 s	0.63–1.9 s	13–40 s

In summary, for normal case, λ values are on the order of a *single block commit latency*, whereas for worst-case, λ grows linearly with $F{+}1$ (session length in views).

C. Comparison with Achilles and Engraft

We provide a detailed qualitative comparison between this work (`Pallas` and `Aegis`), Achilles and Engraft in Table III.

TABLE III: Detailed comparison of `Pallas` + `Aegis` with Achilles and Engraft.

	`Pallas` + `Aegis` (this work)	Achilles [18]	Engraft [61]
Number of Replicas	$N = 2(f + u) + 1$	$N = 2f + 1$.	$N = 2f + 1$.
Threat Model	Up to f Byzantine nodes. • TEEs of Byzantine hosts may be rolled-back (stale state) or cloned • all TEEs of non-Byzantine nodes may crash and restart (mobile crash adversary) Up to u crashed/restarting nodes.	Up to f Byzantine nodes. • TEEs of Byzantine hosts may be rolled-back **but not cloned** • TEEs of non-Byzantine nodes **cannot crash and restart**	Up to f Byzantine nodes. • TEEs of Byzantine hosts may be rolled-back **but not cloned** • TEEs of non-Byzantine nodes **cannot crash and restart**
Safety conditions	$\leq f$ Byzantine nodes	$\leq f$ Byzantine nodes	$\leq f$ Byzantine nodes
Liveness conditions	$\leq f+u$ simultaneously faulty nodes for long enough, which is ensured against a mobile crash adversary when $\lambda \geq T_R$ (cf. Appx. B) after GST (partial synchrony),	$\leq f$ simultaneously faulty nodes after GST (partial synchrony)	$\leq f$ simultaneously faulty nodes after GST (partial synchrony)
Synchronizer Design	**`Aegis`**: TEE-aware view synchronizer with linear expected message complexity and expected constant time complexity.	Exponential backoff synchronizer with unbounded message and time complexity.	No synchronizer, uses Raft's built-in leader election and timeout mechanism and Tiks, a key-value distributed storage mechanism, to maintain a correct state.
TEE Membership Changes	**`Aegis`** synchronizer restricts each replica to one active TEE per view. TEE joins are approved via consensus (**`Pallas`**). Dedicated recovery procedures that involves the hosts reconstructs a valid TEE membership if there are too many crashed TEEs.	Rollback-resilient recovery via quorum of $f + 1$ helper replicas. Assumes a single valid TEE per node without explicit enforcement (no TEE cloning).	Rollback-resilient recovery via quorum of $f + 1$ helper replicas. Assumes a single valid TEE per node without explicit enforcement (no TEE cloning).