Achilles: Efficient TEE-Assisted BFT Consensus via Rollback Resilient Recovery

Jianyu Niu[*][†]
niujy@sustech.edu.cn
SUSTech

Xiaoqing Wen[*]
xqwen@student.ubc.ca
University of British Columbia

Guanlong Wu[*]
santiscowgl@gmail.com
SUSTech

Shengqi Liu
liusq2020@mail.sustech.edu.cn
SUSTech

Jiangshan Yu
J.Yu.Research@gmail.com
The University of Sydney

Yinqian Zhang[†]
yinqianz@acm.org
SUSTech

Abstract

BFT consensus that uses Trusted Execution Environments (TEEs) to improve the system tolerance and performance is gaining popularity. However, existing works suffer from TEE rollback issues, resulting in a tolerance-performance tradeoff. In this paper, we propose Achilles, an efficient TEE-assisted BFT protocol that breaks the tradeoff. The key idea behind Achilles is removing the expensive rollback prevention of TEEs from the critical path of committing transactions. To this end, Achilles adopts a *rollback resilient recovery* mechanism, which allows nodes to assist each other in recovering their states. Besides, Achilles follows the chaining spirit in modern chained BFT protocols and leverages customized *chained commit rules* to achieve linear message complexity, end-to-end transaction latency of four communication steps, and fault tolerance for the minority of Byzantine nodes. Achilles is the first TEE-assisted BFT protocol in line with CFT protocols in these metrics. We implement a prototype of Achilles based on Intel SGX and evaluate it in both LAN and WAN, showcasing its outperforming performance compared to several state-of-the-art counterparts.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Keywords: BFT consensus, TEE, Rollback, CFT consensus

ACM Reference Format:
Jianyu Niu, Xiaoqing Wen, Guanlong Wu, Shengqi Liu, Jiangshan Yu, and Yinqian Zhang. 2025. Achilles: Efficient TEE-Assisted BFT Consensus via Rollback Resilient Recovery. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3717457

1 Introduction

Byzantine Fault Tolerant (BFT) consensus, as an important primitive in distributed computing, has recently gained renewed interest due to its applications in shared databases [59], distributed storage [60], and blockchains [26, 29]. BFT consensus enables a set of nodes to agree on the same ever-growing sequence of transactions, even if some nodes are *Byzantine* (*i.e.*, behaving arbitrarily). However, this promising tolerance comes at the cost of a lower tolerance threshold and performance, compared to Crash Fault Tolerant (CFT) consensus that handles nodes' crash behaviors. First, BFT protocols tolerate one-third of Byzantine nodes, whereas CFT protocols work with a minority of crashed nodes. Second, BFT protocols endure longer latency (*e.g.*, five communication steps including two for client interactions in PBFT [15]) [27, 45, 56] or have higher message complexity. In contrast, CFT protocols like Raft [48] can achieve latency of four communication steps and linear message complexity.

To reduce the cost, a line of work [18, 33, 62, 66] leverages Trusted Execution Environments (TEEs) such as Intel SGX [31] to design BFT consensus (referred to as TEE-assisted BFT consensus). TEEs enable applications to run in a hardware-protected environment with integrity and confidentiality guarantees by isolating applications' code and data from the OS or hypervisor. The integrity of TEEs can prevent Byzantine nodes from equivocating their messages by either equivocation prevention or detection methods [10]. As a result, authenticated nodes can reach consensus in the presence of a minority of Byzantine nodes (without equivocation) [19], keeping the tolerance threshold of TEE-assisted BFT protocols in line with their CFT counterparts. Besides, TEE-assisted BFT can also cut off some communication steps required for BFT consensus to prevent the leader from equivocating proposals (*e.g.*, PRE-PREPARE phase in PBFT).

Despite the promising improvement, Gupta *et al.* [28] have recently identified three alleged problems, *i.e.*, restrictive responsiveness, the lack of parallelism, and TEEs' rollback

[*] Both authors contributed equally to this research.

[†] Jianyu Niu and Yinqian Zhang are affiliated with Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering of SUSTech. Corresponding author: Yinqian Zhang.

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717457

issues, of existing TEE-assisted BFT protocols. Specifically, the rollback issues can be addressed by existing prevention methods [43, 46, 50, 57, 58], which are expensive for performance. To address these problems, the authors propose FlexiBFT, which relaxes the tolerance threshold from $n = 2f + 1$ to $n = 3f + 1$ (n and f denote the number of nodes and faulty ones, respectively). In other words, FlexiBFT trades the tolerance for better performance by achieving responsiveness, improving parallelism, and reducing access to TEEs (incurring expensive rollback prevention each time). However, the priority on performance has sparked significant controversy [10], as it conflicts with prior works that pursue higher fault tolerance [10, 19]. To avoid potential conflicts within the community, a straightforward question arises: *How to break the tolerance-performance tradeoff to design an efficient TEE-assisted BFT protocol?*

In this paper, we aim to break the performance-tolerance tradeoff by resolving the three problems. We find only rollback issues remain unsolved, since the other two can be addressed by slightly modifying the design (See more in Sec. 6.1.). To address rollback issues, one key is *removing* the associated prevention from the critical path of committing transactions, thereby ensuring that system performance is no longer affected. In existing TEE-assisted BFT protocols, when a node sends messages (*e.g.*, proposals and votes), it has to access trusted components (*e.g.*, message counter [62]) implemented within TEEs to certify them. For each access, invoked trusted components need to increment low-performance *persistent counters* (*e.g.*, TPM counter [42] or ROTE [43]) to *solely* prevent state rollback during state recovery. (See more details in Sec. 2.2.) Instead of recovering states from untrusted storage locally, our approach enables nodes to assist each other in recovering their trusted components. This is because exchanged messages among nodes allow them to infer the latest states of one another. Since recovery occurs rarely compared with frequently invoked trusted components in the consensus process, this approach can significantly reduce the overhead of rollback prevention.

We observe that trusted components that adopt equivocation prevention methods [8, 10, 20] are designed to enforce a node to send messages at most once in each view. Besides, when the node enters a new view, it stops sending messages for lower views. Therefore, the recovery should enable the node's trusted components to know sent messages for the highest view. At first glance, it is impossible to realize this *precise recovery* since a message certified by the nodes' trusted components may not reach others due to network asynchrony, or may be received but hidden by Byzantine nodes. To address the impossibility, one way is to relax the recovery requirement with the promise of not compromising security. That is, a node cannot send any messages until entering a higher view since it does not know whether it has sent messages in the current view.

We propose Achilles[1], an *efficient* TEE-assisted BFT protocol that adopts a *rollback resilient recovery* to break the performance-tolerance tradeoff. Except for rollback issues, existing TEE-assisted BFT protocols are inefficient in latency and message complexity. For example, FlexiBFT has $O(n^2)$ message complexity for committing transactions in four communication steps. Damysus [20] has linear message complexity for using chain structure; however, it commits transactions in six communication steps. Later, OneShot [21] optimizes Damysus by having four communication steps in the normal and piggyback execution (*i.e.*, all previously proposed transactions are committed), while still having six steps otherwise. Compared with CFT protocols like Raft, a gap in message complexity and latency exists (Sec. 2.2).

To eliminate the gap, we observe that the PREPARE phase in Damysus can be cut off because of equivocation prevention and chained commitment. The former is provided by trusted components within TEEs, and the latter is that the commitment of descendant blocks will lead to the commitment of its uncommitted parent blocks. By them, Achilles adopts customized *chained commit rules* to achieve linear message complexity and latency (not including view-change phase) of four communication steps. Achilles is the first TEE-assisted BFT consensus in line with their CFT counterparts in message complexity, latency, and tolerance threshold.

We implement a prototype of Achilles based on Intel SGX. We develop Achilles atop the Damysus implementation. We conduct extensive experiments on the public cloud platform to evaluate and compare Achilles with three counterparts, Damysus-R (*i.e.*, a variant of Damysus with rollback prevention), FlexiBFT, and OneShot-R (*i.e.*, a variant of OneShot with rollback prevention). We run experiments over LAN and WAN with up to $f = 30$ Byzantine nodes.

Our contributions. The contributions are as follows:

- We propose Achilles, an efficient TEE-assisted BFT consensus protocol that leverages rollback resilient recovery to break the tolerance-performance tradeoff. Achilles removes the expensive rollback prevention from the critical path of committing transactions.
- We propose *chained commit rules*, making Achilles with linear message complexity and end-to-end latency of four communication steps. Achilles is the first TEE-assisted BFT protocol with the same tolerance threshold, message complexity, and latency as CFT protocols like Raft.
- We also develop proof-of-concept of Achilles and evaluate its performance on the public cloud platform. Our evaluation results show that Achilles achieves a throughput of 75.38K TPS and latency of 5.12ms in LAN with $f = 30$ Byzantine nodes, which is 17×, 6×, and 7× higher throughput than Damysus-R, FlexiBFT, and OneShot-R, respectively.

[1] Achilles, a legendary Greek hero in Homer's epic poems, possess the *ankle* of *Damysus*—the fastest Giants in the Greek mythology [1].

2 Background and Motivations

2.1 Rollback Issues of TEEs

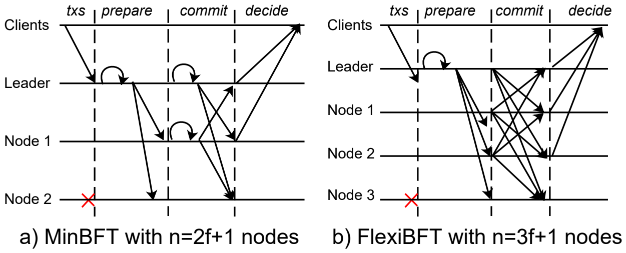
TEEs are CPU extensions that enable applications to run in secure execution environments (known as *enclaves*) with integrity and confidentiality guarantees, by leveraging techniques like hardware-assisted isolation, memory encryption, and remote attestation. Influential TEE platforms include Intel SGX [31] and AMD SNP [34], which have been used in applications such as blockchains [17, 37], trusted storage [49] as well as authentication rate limiting [58]. These applications within TEEs are referred to as *enclave applications*. Specifically, an enclave application has to continuously store its encrypted state data (*e.g.*, invoking seal function in SGX) on untrusted storage to enable state recovery from faults (*e.g.*, power outages or system crashes).

However, existing TEE platforms cannot guarantee the freshness of state data retrieved by enclave applications after rebooting, leading to rollback issues. In particular, an adversary who controls the operating system, can provide an enclave application with stale versions of stored data to roll back its state to a previous state. The state rollback has severe consequences in many applications, especially TEE-assisted BFT consensus [28]. For example, an adversary can reset the virtual message counter [8, 39, 68] implemented within TEEs such that the node sends equivocating messages with the same counter value. Note that, unlike these virtual counters, the below-mentioned persistent counters do not suffer from rollback attacks.

Rollback prevention. Existing rollback prevention solutions rely on trusted *persistent* counters, whose value, once incremented, cannot be reverted to a previous value [54]. Specifically, before an enclave application updates its state, there are two operations: 1) *store* operation, where it binds each state data on the disk with the counter value, and 2) *increase* operation, where it increases the persistent counter by one. After reboots, the enclave application can check whether the state data retrieved from the OS matches the obtained counter value. There are two classes: hardware-based and software-based persistent counters, as below.

1) Hardware-based persistent counters include SGX counter[2] [5], TPM counter [42], and TPM NVRAM [50, 57, 58]. All these counter realizations have poor performance, *i.e.*, long latency for write/read operations and limited write cycles. For example, incrementing a TPM counter for a state update takes about 97ms, and reading a counter for a state check takes about 35ms [58]. Thus, they are impractical for TEE-assisted BFT consensus that requires high performance for continuous state updates.

2) Software-based persistent counters like ROTE [43], Narrator [46, 51] and TIKS [64] are realized by a distributed system of TEEs. Specifically, these TEEs run broadcast protocols (with at least two communication steps) to maintain consistent in-memory counter values. Thus, integrating a software-based counter in TEE-assisted BFT consensus will introduce multiple communication steps. Other prevention methods using trusted server [61] and client-side detection [12] either rely on centralized trust or are not general, making them infeasible for TEE-assisted BFT consensus.



a) MinBFT with n=2f+1 nodes b) FlexiBFT with n=3f+1 nodes

Figure 1. An illustration of the tolerance-performance trade-off. FlexiBFT lowers the tolerance to reduce the expensive access to trusted components denoted by the circle.

2.2 Why Are TEE-Assisted BFT protocols Inefficient?

BFT consensus can leverage the integrity of TEE to prevent authenticated nodes from equivocating messages, resulting in a higher tolerance threshold and better performance (*e.g.*, reducing message complexity, improving system parallelism, or shortening latency) [8, 20, 21, 28, 39, 68]. We now introduce three state-of-the-art protocols called Damysus [20], FlexiBFT [28] and OneShot [21] below, while leaving others in Sec. 7.

Damysus built atop HotStuff [67] leverages two trusted components, *i.e.*, checker and accumulator, to commit transactions in six communication steps in the presence of the minority of Byzantine nodes. The commit latency is end-to-end since it includes two steps for receiving clients' transactions and sending replies. Damysus enjoys the linear message complexity in normal-case operation and view-change phases. Later, OneShot optimizes Damysus by having four communication steps in the normal and piggyback execution, while still having six steps otherwise. Unlike Damysus and OneShot, FlexiBFT lowers the tolerance from $2f + 1$ to $3f + 1$ to achieve higher parallelism, better responsiveness, and less rollback prevention overhead. FlexiBFT commits a block in four communication steps but has $O(n^2)$ message complexity for broadcasting messages. Note that the steps of view-change for rotating leaders are not considered for a fair comparison. Despite these advancements, they are still inefficient in the following two aspects.

Expensive rollback prevention impedes performance. In existing TEE-assisted BFT protocols (*e.g.*, MinBFT [62]), to commit a transaction, the leader (resp., backup nodes) has to access trusted components within TEEs when proposing (resp., voting for) the transaction to prevent equivocation,

[2] Intel SGX does not support persistent counters anymore [42].

as shown in Fig. 1. When the trusted components are invoked each time, they will update the state and increase a trusted persistent counter to prevent rollback issues. Thus, the latency of a transaction in MinBFT [62] includes at least two times the write latency of counters (one for the leader and another for backup nodes). Even worse, to commit a transaction in Damysus-R, both leader and backup nodes use the trusted components twice. (See detailed description of Damysus-R in Sec. 5.) Thus, Damysus-R doubles the overhead of using counters for rollback prevention. Similarly, in OneShot-R, leader and backups use the trusted components once in the normal case, while using trusted components twice otherwise.

However, existing hardware- and software-based persistent counters have significant performance limitations (see Sec. 2.1). Moreover, nodes have to use these counters multiple times during transaction commitment to mitigate rollback attacks. This is why FlexiBFT lowers the tolerance, by which backup nodes can roll back their states to avoid using expensive persistent counters.

None achieves linear message complexity and latency of four communication steps as CFT protocols. CFT protocols like Raft can achieve linear message complexity and latency of four communication steps. By contrast, existing TEE-assisted BFT protocols achieve either linear message complexity (*e.g.*, Damysus) or the optimal four communication steps (*e.g.*, FlexiBFT), but not both simultaneously.

Summary. Existing TEE-assisted BFT protocols are inefficient in rollback prevention and performance (*i.e.*, message complexity and latency), as summarized in Table 1. This motivates us to propose Achilles, an *efficient* TEE-assisted BFT protocol that does not use expensive persistent counters for rollback prevention. Achilles also achieves linear message complexity, end-to-end latency of four communication steps, fault tolerance for the minority of Byzantine nodes, and reply responsiveness (introduced in Sec. 6.1).

3 System Models and Goals

3.1 System Model

We follow the system model of existing TEE-assisted BFT protocols [21, 24, 39]. We consider a distributed system maintained by $n = 2f + 1$ nodes $\{p_i\}_{i=1}^{n}$, and meanwhile accessed by a set of clients. We assume every node is provisioned with TEEs to run some trusted components (specified in Sec. 4.3). We assume a Public Key Infrastructure (PKI) among nodes to distribute the keys required for authentication and message signing. Specifically, each node p_i has a public/private key pair, denoted by (pk_i, sk_i), in which the private key is only accessible by nodes' trusted components.

Each pair of nodes is connected by a reliable channel. We adopt the partial synchrony model proposed by Dwork et al. [23], commonly used in BFT consensus [15, 25, 32, 67]. There is an established bound Δ and an undefined Global Stabilization Time (GST). After the GST point, the delivery of any message transmitted between two honest nodes within the Δ limit is guaranteed. That is, the system behaves *synchronously* following the GST.

Threat model. We consider an adversary $\mathcal{A}$ that corrupts up to f nodes at any time. Corrupted nodes are *Byzantine*, *i.e.*, exhibit arbitrary behavior controlled by the adversary with the exception that TEE integrity (see below) and cryptographic schemes (*e.g.*, public key signatures and collision-resistant hash functions used in this paper) cannot be breached. We define nodes that strictly follow the protocol and do not crash as correct nodes, while the rest are Byzantine.

Given a corrupted node, the adversary gains full control over its operating system and thus can modify, reorder, and delay network messages from/to TEEs. The adversary can start, stop, and invoke the local TEE enclaves with arbitrary input, but it cannot extract the memory contents or manipulate the running code in enclaves to compromise the integrity. The adversary can also roll back TEEs' states to some previous versions (including resetting states) by providing stale stored data outside TEEs, which is also known as rollback attacks [43, 50, 57, 58, 61]. Besides, forking attacks [43, 46] that enable a node to access more than one TEE enclave running the same trusted components are outside the scope of this paper, as this attack can be mitigated by either using TPM PCR [58] or session key mechanism [43, 46].

3.2 System Goals

Clients create a set of transactions, which are sent to nodes. Nodes can process clients' transactions and further pack them in blocks. Each block also includes a cryptographic hash reference of a previous block, called the parent block. (The block format and chain structure are introduced in Sec. 4.2.) Each node running the protocol commits a sequence of linked blocks such that the following security properties hold:

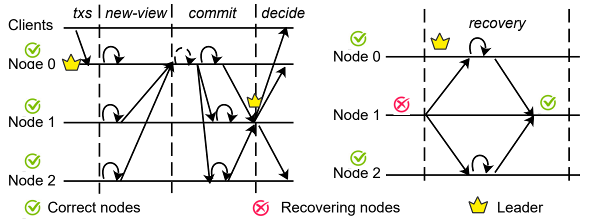
- *Safety:* If two correct nodes commit two blocks b and b' at the same height, then $b = b'$.
- *Liveness:* Clients' transactions will be eventually included in a block committed by correct nodes.

4 Achilles Design

This section introduces Achilles, an efficient TEE-assisted BFT protocol with a rollback resilient recovery mechanism. Achilles customizes the two trusted components, *i.e.*, checker and accumulator, first proposed in Damysus. The customization allows checker to record the latest (un)prepared blocks from leaders, and accumulator to extend the block recorded in checker (Sec. 4.3). With the customized components, Achilles further extends the normal-case operations of Damysus to remove the prepare phase (Sec. 4.4). More importantly, the new rollback resilient recovery of Achilles allows nodes to recover their states with the assistance of other nodes rather than using expensive persistent counters

Table 1. The comparison between Damysus (-R), FlexiBFT, OneShot (-R) and Achilles. Damysus-R and OneShot-R are variants of Damysus and OneShot with rollback resilience, respectively (see Sec. 5). Rollback Res. and Reply Res. are short for Rollback Resistance and Reply Responsiveness, respectively.

Protocols	Threshold	Rollback Res.	# Persistent Counter	Commun. Complixity	Commun. Steps	Reply Res.
Damysus (-R) [20]	$2f+1$	✗(✓)	0 (4)	$O(n)$	6	✗
FlexiBFT [28]	$3f+1$	✓	1	$O(n^2)$	4	✓
OneShot (-R) [21]	$2f+1$	✗(✓)	0 (2 or 4)	$O(n)$	4 or 6	✗
Achilles	$2f+1$	✓	0	$O(n)$	4	✓



a) Normal-case operations b) Rollback Resilient Recovery

Figure 2. An overview of Achilles's normal-case operations and rollback resilient recovery. The circle (resp., dashed) denotes the access to checker (resp., accumulator) components within TEEs.

(Sec. 4.5). For a better understanding of Damysus, we refer readers to Appendix A [47] for more details.

4.1 Overview

Achilles runs in views, in which a delegated node, called the leader, coordinates with other nodes, called backups, to commit transactions. Achilles uses a round-robin policy to change leaders for each view, following the chaining spirit of chained BFT protocols, *e.g.*, HotStuff and Damysus. In Achilles, there are two main components: the one-phase protocol (also called normal-case operations) based on chained commit rules to finalize transactions and rollback resilient recovery, as shown in Fig. 2. Except for them, Achilles also has a pacemaker component. Next, we provide a high-level introduction to these components, with detailed descriptions available in Sec. 4.4 and Sec. 4.5.

Normal-case operations. The normal-case operations in each view contain three phases: 1) new-view phase, in which the leader collects at least $f+1$ view messages containing the latest stored blocks from nodes; 2) commit phase, where the leader creates one block and collects votes from at least $f+1$ nodes; 3) decide phase, where the leader commits the block, execute it and informs the client and other nodes. The new-view phase allows a new leader to synchronize the latest block information from backup nodes. This phase can be skipped for view $v+1$ if the associated leader receives a committed block produced at view v. A similar optimization is adopted in chained BFT protocols like Fast-HotStuff [32], HotStuff-2 [41], and OneShot. Furthermore, if a stable leader policy—where the leader does not change every block—is adopted, this phase can be removed. This is why we do not consider it in transaction latency (in terms of message steps) for a fair comparison with other protocols. See more details of the normal-case operations in Sec. 4.4.

In the above phases, Achilles uses checker and accumulator components to certify messages. The checker component is accessed by nodes (including the leader) in the new-view phase and commit phase, while the accumulator component is only used by the leader in the commit phase, as shown in Fig. 2.

Rollback resilient recovery. After rebooting, a node must recover the state of its trusted components before participating in normal-case operations, pacemaker, or replying to others' recovery requests. The rollback resilient recovery has to ensure *no equivocation* property, *i.e.*, if a node sends two messages *msg* and *msg*′ of the same type in a view, then $msg = msg'$. This property prevents nodes from sending equivocating messages, which play complementary roles for the safety of the consensus.

In Achilles, the checker component maintains important system states, *e.g.*, current view, phase, and the last unprepared block, whereas the accumulator is stateless. Thus, the goal of recovery is to retain the states for the checker component. To achieve the goal, a recovering node can send recovery requests to other nodes to obtain the states in their checker components, as shown in Fig. 2. A nonce is included in a request and associated replies to prevent replay attacks. The detailed description of the recovery procedure is provided in Sec. 4.5.

Pacemaker. Achilles follows the pacemaker design of Damysus and HotStuff [67], which is a mechanism to ensure liveness, *i.e.*, making consensus progress after GST. It has two goals: 1) making all correct nodes and a unique leader enter the same view for a sufficiently long period, and 2) ensuring the leader extends a parent block that all correct nodes will vote for. Specifically, for the first goal, one common solution [14, 15] is increasing the timeouts for a view, until progress is being made. For the second, after a view change in Achilles, the new leader has to collect new-view messages sent by $f+1$ distinct node to extend the latest (un)prepared block stored in nodes' checker components (Sec. 4.4). We refer readers to [67] for more details.

4.2 Data Structures

Signatures and cryptographic hash. Nodes and trusted components use asymmetric signature schemes to authenticate messages. A signature scheme provides two main functions: SIGN function, which generates a signature σ over a message msg using a private key sk; and VERIFY function, which verifies the signature σ over msg using the public key pk. We use $\langle msg \rangle_\sigma$ (resp., $\langle msg \rangle_{\vec{\sigma}}$) to denote a signed message msg that carries a signature σ (resp., a list of signatures $\vec{\sigma}$) using the SIGN function. We use $\sigma.id$ to denote the identity of the signature σ. Note that signatures are generated by trusted components using their private keys (see Sec. 4.3). The cryptographic hash function $\mathsf{H}(\cdot)$ takes a string of arbitrary length as input and outputs a fixed-length string.

Block format. A block b is $\langle txs, op, h_p \rangle$, where txs is a batch of client transactions, op is the execution results, and h_p is the hash value of a previous block (also referred to as the parent block). For convenience, $b.x$ denotes the associated parameter x of the block b. For example, $b.txs$ denotes the included transactions. We assume two functions: $\mathsf{executeTx}(txs, h_p)$ function that outputs the execution results op for txs given chaining blocks ended up with the block with hash h_p; and $\mathsf{createLeaf}(txs, op, h_p)$ function that creates a new block extending a parent block with the hash h_p.

Due to the hash references, blocks can be cryptographically linked to form a chain structure. Thus, $b_1 \succ b_2$ denotes that b_1 extends a block b_2, which anyone can verify through the hash references. We also write $b_1 \succ h$ when b_1 is an extension of a block b_2 with hash value h. We say that a block b_1 conflicts with a different block b_2 if neither $b_1 \succ b_2$ nor $b_2 \succ b_1$. In particular, a genesis block $\mathcal{G}$ is hard-coded. The height of a non-genesis block is the distance from it to the genesis block $\mathcal{G}$, and so the height of the genesis block $\mathcal{G}$ is zero. In Achilles, a valid block b means that $b \succ \mathcal{G}$ and any b' that satisfies $b \succ b'$ have valid execution results $b'.op$ by executing $b'.txs$, and $b.op$ is also valid. Besides, the freshness of blocks is compared according to their views. The block with the highest view is the latest one.

Certificate. In Achilles, a node uses trusted components to certify their messages to prevent message equivocation or obey predefined rules. Other nodes can verify a certificate ϕ signed by these components as proof. We use $\vec{\phi}$ for a list of certificates, and $\vec{\phi}^n$ to indicate that the list has length n. There are five certificates, introduced below.

- **Block certificate.** A block certificate (denoted as ϕ_b) is created by the leader for its block in the commit phase. It has the form $\langle \textsc{prop}, h, v \rangle_\sigma$, where h is the hash of the block, v is the view number at which the block is produced, and σ is the leader's signature. For convenience, $\phi_{b \cdot view}$ and $\phi_{b \cdot hash}$ denote the view v and hash h, respectively. This certificate is produced by TEEprepare function in Algorithm 2, which guarantees that the leader can only make one block per view by setting $flag$ to 1.
- **Store certificate.** A store certificate (denoted as ϕ_s) is created by a node (including the leader) to certify the receipt of a block from the leader in the commit phase (Line 23, Algorithm 1). It has the form $\langle \textsc{commit}, h, v \rangle_\sigma$, where h is the hash of the stored block, v is the view at which the block is produced, and σ is the signature of the node. This certificate is produced by TEEstore function in Algorithm 2. Before calling the function, a node has to verify that the block from the leader at view v is valid.
- **Commitment certificate**: A commitment certificate (denoted as ϕ_c) is the combination of $f + 1$ store certificates produced by the leader in the commit phase. The $f + 1$ store certificates ensure at least a correct node participated in the vote, and therefore holds the corresponding block (correct nodes only vote for blocks they have received). It has the form $\langle \textsc{decide}, h, v \rangle_{\vec{\sigma}^{f+1}}$.
- **Accumulator certificate**: An accumulator (denoted as acc) is generated when the leader calls the TEEaccum function to select the parent block (Line 22-25, Algorithm 2). Specifically, the function inputs $f + 1$ view certificates from nodes. It has the form $\langle \textsc{acc}, h, v, \vec{id} \rangle_\sigma$, where h and v are the hash and view of the parent block, $\vec{id}$ is the vector of the $f + 1$ ids of the nodes that contributed to the accumulator, *i.e.*, that signed the view certificates passed as arguments to TEEaccum, and σ is the signature of the leader. We use $acc.hash$ for h, and $acc.v$ for v. We say that an acc certificate is valid if its signature is correct, and if $\vec{id}$ is a vector of $f + 1$ unique ids (Line 7, Algorithm 2).
- **View certificate**: A view certificate (denoted as ϕ_v) is generated by a node (including the leader) either when a view ends successfully with a committed block, or when time out is triggered. It has the form $\langle \textsc{new-view}, h, v, v' \rangle_\sigma$, where h is the hash of the stored block, v is the view at which the block is produced, v' is the current view, σ is the signature created by the node. The v' prevents a stale certificate from being replayed by Byzantine nodes. This certificate is produced by TEEview function in Algorithm 2.

4.3 Trusted Components

Achilles uses checker and accumulator components running in TEEs. They are inherited from Damysus [20], but are extended to remove the prepare phase in Damysus. The main extension is that in Achilles, checker records information (*i.e.*, hash value) of the latest blocks from leaders, which can be both prepared or unprepared. By contrast, in Damysus, checker only stores information of prepared blocks, *i.e.*, being certified by at least $f + 1$ nodes' votes in the prepare phase. Accordingly, in Achilles, accumulator allows a leader to extend the latest (un)prepared block stored in nodes' checker components. We now introduce these two components.

Checker. The checker mainly provides two services: 1) it binds each consensus message (*i.e.*, block and vote) with a unique identifier in each view to prevent equivocation, and 2) it stores the block from the leader to prevent nodes from hiding the latest one. Thus, the state of node p_i's checker component has three components:

- $\{sk_i, pk_1, ..., pk_n\}$, where sk_i is its confidential private key and $\{pk_1, ..., pk_n\}$ is the public key of nodes;
- $(vi, flag)$, where vi is the current view number, and $flag$ denotes whether the leader has proposed a block at view vi. If the leader produces a block, $flag$ goes from 0 to 1. If a node stores a block in this view, vi increases by one, and $flag$ is initiated as 0;
- $(prev, preh)$, where $prev$ and $preh$ are the view number and hash of the latest stored block, respectively.

The checker component provides two functions:

- TEEprepare (b, h, ϕ): This function inputs the block b, the hash h of the block b, and a commitment or accumulator certificate. It outputs a block certificate ϕ_b with the leader's signature. The certificate ensures that b is the only block extending b' certified by ϕ.
- TEEstore (ϕ_b): This function inputs a block certificate ϕ_b and outputs a store certificate ϕ_s. Anyone can attest to this certificate by the signature.

Accumulator. This component forces the leader to choose the stored block with the highest view among $f + 1$ ones included in nodes' new-view messages. Unlike checker, only the leader interacts with its accumulator in a view. The accumulator component of the node p_i only maintains the private key sk_i and nodes' public keys $\{sk_i, pk_1, ..., pk_n\}$. This component provides the following interface:

- TEEaccum$(\phi_v, \vec{\phi_n})$: This function takes a list of $f + 1$ view certificates and asserts—by generating an accumulator acc—that the first element in that list is for the block with the highest proposed view (i.e., the view at which the block was prepared). This function provides proof for the leader; its block extends the latest block among these in the received $f + 1$ view certificates.

In Achilles, only the checker component maintains states (*e.g.*, vi and $flag$) related to the normal-case operations. The private key and public keys of nodes can be obtained from reconfiguration information, which can be stored on disks. Thus, a recovering node only recovers states of its checker component. See more details in Sec. 4.5.

4.4 Normal-Case Operations

Algorithm 1 presents the pseudocode of the normal-case operations, which has three steps.

Algorithm 1 The pseudocode of normal-case operations

```
1:  $pks$ // public keys
2:  $vi = 0$ // current view
3:  $preb = \langle b, \phi_b, \phi_c \rangle$ // latest stored block from a leader
4:
5:  // commit phase
6:  as a leader
7:      waits for $f + 1$ valid $\phi_v$ with $\phi_v$ in the form $\langle \text{NEW-VIEW}, h, v, v' \rangle_\sigma$ and let $\vec{\phi}_n$ be $f + 1$ $\phi_v$
8:      If each $\langle b, \phi_n \rangle$ satisfies $\phi_v.view == vi \land \mathsf{H}(b) == \phi_n.hash$
9:          Let $\phi_0$ be the one with the highest $v$ among $\vec{\phi}$
10:         $acc = \text{TEEaccum}(\phi_0, \vec{\phi})$
11:         propose($acc._{hash}$, $acc$)
12:
13: //new-view optimizations
14: as a leader
15:     waits for $\phi_c$ of the form $\langle \text{COMMIT}, h, vi - 1 \rangle_{\vec{\sigma} f+1}$ messages
16:     propose($h, \phi_c$)
17:
18: all nodes
19:     wait for $\langle b, \phi_b \rangle$ from the leader
20:     abort if $b$ is not valid
21:     abort if $\neg(\phi_b.view == vi \land \mathsf{H}(b) == \phi_b.hash)$
22:     $preb = \langle b, \phi_b, \perp \rangle$
23:     send $\phi_s$ = TEEstore($\phi_b$) to the leader
24:
25: // decide phase
26: as a leader
27:     wait for $\vec{\phi}_s$, $f + 1$ $\phi_s$ of the form $\langle \text{COMMIT}, h, v \rangle_\sigma$
28:     $\vec{\sigma}$ := the signature of $f + 1$ COMMIT certs. in $\vec{\phi}$
29:     send $\phi_c = \langle \text{DECIDE}, h, vi \rangle_{\vec{\sigma}}$ to all
30:
31: all nodes
32:     wait for $\vec{\phi}_c$ of the form $\langle \text{DECIDE}, h, vi \rangle_{\vec{\sigma} f+1}$ from the leader
33:     abort if $\neg$VERIFY$(\phi)_{pks}$
34:     $preb = \langle b, \phi_b, \phi_c \rangle$; $vi$++
35:     send $b$ along with $\phi_c$ to the clients
36:     send $\phi_c$ to the leader at view $vi$
37:
38: // new-view phase
39: all nodes
40:     when timeout
41:     $vi$++
42:     $\phi_v$ = TEEview()
43:     send $\phi_v$ to $vi$'s leader
44:
45: function propose($h, \phi$)
46:     $op$ = executeTx($txs, h$)
47:     $b$ = createLeaf($txs, op, h$)
48:     $\phi_b$ = TEEprepare($b, \mathsf{H}(b), \phi$)
49:     send $\langle b, \phi_b \rangle$ to all
```

❶ In the new-view phase (Line 39-43, Algorithm 1), a node first increments its view to enter into the new view. Then, it sends a view certificate ϕ_v containing the hash of the latest stored block b and the associated view to the leader of the new view. Note that if a node commits a block in a view

Algorithm 2 TEE code for normal-case operations

```
1:  $(sk, pks)$ // 1 private and public keys
2:  $(vi, flag) = (0, 0)$ // current view and phase
3:  $(prepv, preph) = (0, H(\mathcal{G}))$ // view/hash of latest stored block
4:
5:  function TEEprepare($b, h, \phi$)
6:      abort if $flag == 1$
7:      If $\phi$ is of the form $\langle \textsc{acc}, h', v, \vec{id} \rangle_\sigma$ and is valid
8:          abort if $\neg(\mathsf{H}(b) == h \land b.h_p == h' \land v == vi)$
9:          $flag = 1$
10:         return $\phi_b = \langle \textsc{prop}, h, vi \rangle_\sigma$
11:     else $\phi$ is of the form $\langle \textsc{commit}, h', v \rangle_{\vec{\sigma}^{f+1}}$
12:         abort if $\neg(\mathsf{H}(b) == h \land b.h_p == h' \land v == vi - 1)$
13:         $flag = 1$
14:         return $\phi_b = \langle \textsc{prop}, h, vi \rangle_\sigma$
15:
16: function TEEstore($\phi_b$) where $\phi_b$ is $\langle \textsc{prop}, h, v \rangle_\sigma$
17:     if VERIFY $(\phi_b)_{pks} \land \phi_b$ is from the leader $\land v \geq vi$ then
18:         $prepv = v; preph = h$
19:         $vi = v; flag = 0$
20:         return $\phi_s = \langle \textsc{commit}, h, v \rangle_\sigma$
21:
22: function TEEaccum($\phi_n, \vec{\phi_n}$) where $\phi_v$ is for $\langle h, v, v' \rangle$
23: if $\left( \begin{array}{l} \mathsf{VERIFY}(\vec{\phi_n})_{pks} \land |\vec{\phi_n}| \geq f + 1 \land \phi_n \in \vec{\phi_n} \land \\ (\forall \phi' \in \vec{\phi_n} \text{ where } \phi' \text{ is } \langle \tilde{h}, \tilde{v}, \tilde{v'} \rangle \land \tilde{v'} == vi \\ \land v \geq \tilde{v}) \end{array} \right)$ then
24:         $\vec{id}$ := the ids of the node signed $\vec{\phi_n}$
25:         return $acc = \langle \textsc{acc}, h, v, \vec{id} \rangle_\sigma$
26:
27: function TEEview()
28:     $vi$++; $flag = 0$
29:     return $\phi_v = \langle \textsc{new-view}, preph, prepv, vi \rangle_\sigma$
```

(Line 32-36, Algorithm 1), it can directly increment its view and send the commitment certificate ϕ_c to the leader at the new view. (See the optimizations below.)

❷ In the COMMIT phase, the leader selects the stored block with the highest view from $f + 1$ valid view certificates using the ACCUMULATOR (Line 7-10, Algorithm 1). The leader has to ensure a block corresponding to the hash in the view certificate is correct. (Note that $\phi_v.view = v'$ in Line 8.) The leader then proposes a block to extend the selected block and uses TEEprepare to certify it.

Upon receiving a block from the leader, a node first checks the block is valid. Specifically, the block and all its ancestor blocks are received by the nodes. Besides, all the execution results in them are valid. If passed the check, it stores the block in *preb* (Line 22, Algorithm 1) and sends a store certificate produced by TEEstore function to the leader.

❸ In the DECIDE phase, if the leader collects $f + 1$ store certificates from nodes, it combines them into a commitment certificate ϕ_c and broadcasts ϕ_c.

When receiving a commitment certificate ϕ_c (*i.e.*, assurance that $f + 1$ nodes stored a block) from the leader, a node records the certificate (Line 34, Algorithm 1) as the latest stored block from a leader in *preb*. Besides, the node starts a new view and replies to the clients by forwarding the certificate and blocks (containing the execution results). Clients can confirm the execution results after validating the certificate from one node's reply, thereby ensuring reply responsiveness (see Sec. 6.1).

Optimization of NEW_VIEW phase. When a leader of view v receives a commitment certificate ϕ_c for a block b in view $v - 1$, it can directly propose a block to extend the block b without waiting for $f + 1$ view certificates. The leader must have the block b and all its ancestor blocks before extension. Besides, after receiving a commitment certificate ϕ_c for the current view, a node can directly increase its view and send a view certificate to the leader of the new view.

Block synchronization. When a node receives a block $\langle b, \phi_b \rangle$ from the leader (Line 19, Algorithm 1), it might not receive ancestor blocks of the block b. Thus, it has to pull these blocks from others. Meanwhile, when a node receives a commitment certificate ϕ_c for a block b, it can send replies to clients for all uncommitted ancestor blocks of the block b. This is, due to the chain structure, if a block is committed, all its ancestor blocks are also committed.

4.5 Rollback Resilient Recovery

When a node reboots, it has to carry out the recovery protocol to obtain states for its trusted components within TEEs. Before completing the recovery, its status is recovering, and it cannot participate in the normal-case operations, view-change protocol, and reply to others' recovery requests. Specifically, the node needs to know node configuration, *i.e.*, obtaining its key pairs and other nodes' public keys for communications. The configuration information is stored on local disks in an encrypted and authenticated way, ensuring the adversary can neither forge nor know it. For example, a node can use the seal function in Intel SGX to store the configuration message on the disks and use unseal to obtain them by its MRENCLAVE identity [46]. Note that the configuration information can be set without relying on a trusted third party. This is because nodes' TEEs can utilize mutual remote attestation to build the PKI, as described in [46]. We assume a group of fixed nodes in ACHILLES and discuss the limitation of dynamic reconfiguration in Sec. 6.2.

As introduced in Sec. 4.3, only the CHECKER component in ACHILLES maintains system states (*i.e.*, vi, $flag$, $prev$, and $preh$), and needs to be recovered after rebooting. Algorithm 3 presents the pseudocode of the recovery procedure, which contains three steps as follows:

❶ A recovering node p_j sends a recovery request (in the form $\langle \textsc{req}, non \rangle$) created by TEErequest function to all other nodes. The *non* is a nonce, which prevents Byzantine nodes from replaying recovery replies. In particular, other nodes include the same *non* in their replies (step 2), and the recovering node checks whether the *non* matches (step 3).

Algorithm 3 The pseudocode of recovery

```
upon rebooting systems
    send $\phi_{req}$ = TEErequest() to all

upon receiving $\phi_{req}$ from node $p_j$
    $\phi_{rpy}$ = TEEreply($\phi_{req}, j$) to node $p_j$
    $\langle b, \phi_b, \phi_c \rangle = preb$;
    send $\langle b, \phi_b, \phi_c, \phi_{rpy} \rangle$ to node $p_j$

upon receiving $f+1$ replies of the form $\langle b, \phi_b, \phi_c, \phi_{rpy} \rangle$
    let $\vec{\phi}_{rpy}$ be $f+1$ $\phi_{rpy}$
    Let $\langle b, \phi_b, \phi_c \rangle$ and $\phi_{rpy}$ from the leader with the highest view $v$
    $vi = v + 1$; $preb = \langle b, \phi_b, \phi_c \rangle$
    send $\phi_v$ = TEErecover($\phi_{rpy}, \vec{\phi}_{rpy}$) to $vi$'s leaders

// TEE code for recovery
function TEErequest()
    return $\langle \text{Req}, non \rangle_\sigma$

function TEEreply($\phi_{req}, j$) where $\phi_{req}$ is $\langle \text{Req}, non \rangle_\sigma$
    abort if $\neg$Verify$(\phi_{req})_{pk_j}$
    return $\phi_{rpy} = \langle \text{RPY}, preph, prepv, vi, j, non \rangle_\sigma$

function TEErecover($\phi_{rpy}, \vec{\phi}_{rpy}$), where $\phi_{rpy}$ is for $\langle h, v, v' \rangle$
    Let $id$ be the id $\phi_{rpy}$
    abort if $\forall \phi' \in \vec{\phi}_n$ not have valid $non$ and id
    abort if the node signing $\phi_{req}$ not the leader at view $v'$
    abort if $id$ is not the leader at view $v'$
    if $\left( \begin{array}{l} \text{VERIFY}(\vec{\phi}_{rpy})_{pks} \wedge |\vec{\phi}_{rpy}| \geq f+1 \wedge \phi_{rpy} \in \vec{\phi}_{rpy} \\ \wedge (\forall \phi' \in \vec{\phi}_n \text{ where } \phi' \text{ is } \langle \tilde{h}, \tilde{v}, \tilde{v}' \rangle \wedge v' \geq \tilde{v}') \end{array} \right)$
    then
        $(vi, flag) = (v' + 2, 0)$
        $(prpv, preph) = (v, h)$
        return $\phi_v = \langle \text{NEW-VIEW}, vi, preph, prepv \rangle_\sigma$
```

❷ When receiving the request from node p_j, a node calls TEEreply function to create a recovery reply. First, the function checks the signature is created by node p_j. If passing the check, the function creates a reply $\phi_{rpy} = \langle$RPY, *preph*, *prepv*, *vi*, *j*, *non*$\rangle$,where *vi* is its view number, *non* is the nonce in the request, *preph* and *prepv* is the hash and view of the latest stored block. Then, the node sends the certificate ϕ_{rpy} with the associated block b, its block certificate ϕ_b, and commitment certificate ϕ_c to the node p_j.

❸ The recovering node waits to receive at least $f+1$ recovery replies from different nodes. Next, the node selects the replies from a leader with the highest view among all replies. It sets its view to the view of the leader plus two and updates its state *preb* using the information from the reply. Then, it calls the TEErecover function with the inputs of the $f+1$ reply certificates and the one from the leader. The function will carry out the following checks:

- All reply certificates have the valid *non* and *id*. The *non* matches the one in the request, and *id* is its identity.
- There are $f+1$ reply certificates with valid signatures.
- The certificate from the leader is in the set and has the highest view.

After passing these checks, the function updates the states, *i.e.*, *vi*, *preph*, *prepv* according to the information in the leader certificate. Finally, the node moves to a new view and sends a view certificate to the current leader. The recovery protocol is complete.

When a leader of the current view v enters the recovery procedure, it cannot obtain a recovery reply from the leader of the current view, It has to wait for the next leader to be elected. Besides, a recovering node can send new recovery requests to other nodes if it cannot collect $f+1$ recovery replies with the latest one (*i.e.*, the recovery reply with the highest view number) from the leader in a given period.

There are two key points in the above steps. *First*, among the $f+1$ recovery replies, the one with the highest view must come from the leader of this view. The reason is that only the leader with the highest view has the latest state information. Without this rule, there is a security issue. For example, consider a simple attack case of 5 nodes, *i.e.*, $\{p_i\}_{i=1}^5$. The node p_1 is the leader at view v. First, p_1 sends a block b extending the committed block b_0 to p_2, which sends store certificates to p_1. Then, p_2 is crashed and recovers its states from p_3, p_4, and p_5, which do not have the block b. After repeating the above process over the node p_3 and p_4, the leader p_1 can commit the block b that is only stored by itself. Later, p_1 is partitioned from other nodes, and nodes enter the view $v+1$ with the leader p_2. Finally, the leader p_2 obtains view certificates from p_3 and p_4 and can propose a block b' extending the block b_0. The conflicting block b' will be committed, violating the security.

Second, when a node obtains the highest view v' from the $f+1$ recovery replies, it cannot send any messages for this view. This is because the node does not know whether it has sent messages in this view before rebooting. In particular, the node has to set its view to $v'+2$ to prevent equivocation. The cause is that due to the optimization of NEW-VIEW phase, a node may become a leader of view $v+1$ for receiving a commitment certificate of a block of view v, while most nodes (including the leader of view v) still stay in view v.

4.6 Correctness Analysis

In this section, we provide a high-level view of the proof for *safety* and *liveness* properties (Sec. 3.2) and leave the detailed analysis in Appendix B [47]. Before proving these two properties, we first provide a proof sketch for the *no equivocation* property of the recovery.

Definition 1 (No equivocation)**.** *If a node sends two block (or store) certificates ϕ and ϕ' in the same view, then $\phi = \phi'$.*

Proof sketch of rollback resilient recovery. For no equivocation property, we observe that in the ideal case without recovery, a node can only send one block (or store) certificate for a view due to the CHECKER components. In the case that a node crashes in view v, it will enter a view $v' \geq v$ after completing the recovery procedure in Algorithm 3. Thus, leader (resp., backups) cannot send equivocating block (resp., store) certificates at the same view.

Proof sketch of safety and liveness. For safety, a committed block in view v must be extended by all blocks proposed in view $v' \geq v$. Specifically, in ACHILLES, the block is either directly committed by the leader that collects $f + 1$ store certificates or indirectly committed by a committed child block b'. If the block is directly committed, it has been stored by at least $f + 1$ nodes. The *no equivocation* ensures that no other blocks with view v exist. Besides, the block must be included in at least one view certificate and be selected as the parent block. Thus, subsequent blocks in view $v' \geq v$ will extend it according to the TEEaccum and TEEprepare functions. If the block is directly committed by the block b', all subsequent blocks will extend the block b' and also include itself.

Regarding liveness, if all correct nodes enter the same view with a correct node as the leader for a long period after GST, the leader can coordinate backup nodes to commit its block. Specifically, the leader can collect either $f + 1$ view certificates from the correct nodes or a commitment certificate for the block in the previous view to select the parent block. The selected parent block and all its ancestor blocks must be available for the leader. Then, the leader can create and broadcast a block containing clients' transactions to extend the parent block. Then, the block will be stored and voted on by all correct nodes, resulting in being committed.

5 Evaluation

We implement a prototype of ACHILLES based on Intel SGX. We develop ACHILLES atop the Damysus implementation. We evaluate the performance of ACHILLES in LAN and WAN with different settings (*e.g.*, batch size). We compare ACHILLES with Damysus, FlexiBFT, and OneShot to show the performance improvements. Since Damysus and OneShot do not consider rollback issues, we use a persistent counter by following FlexiBFT to provide associated prevention. We name the two variants as Damysus-R and OneShot-R, respectively. We also evaluate ACHILLES under faults to illustrate the overhead of the rollback resilient recovery. We aim to answer the following questions:

- **Q1:** How does ACHILLES perform with varying nodes in WAN and LAN compared to counterparts? (Sec. 5.2)
- **Q2:** How does ACHILLES perform under faults? (Sec. 5.3)
- **Q3:** How much performance overhead is introduced by SGX-related operations? (Sec. 5.4)

5.1 System Implementation and Setup

Implementation. We use Intel SGX to provide trusted service and develop ACHILLES[3] atop the Damysus implementation[4]. All protocols are implemented in C++. We customize the trusted components, CHECKER and ACCUMULATOR in Damysus. We also realize a rollback resilient recovery for the CHECKER component (Sec. 4). We use OpenSSL library [3] to realize ECDSA signatures with prime256v1 elliptic curves and Salticidae [4] for nodes' connection.

Baselines. We consider three counterparts, Damysus-R, FlexiBFT, and OneShot-R, as introduced below.

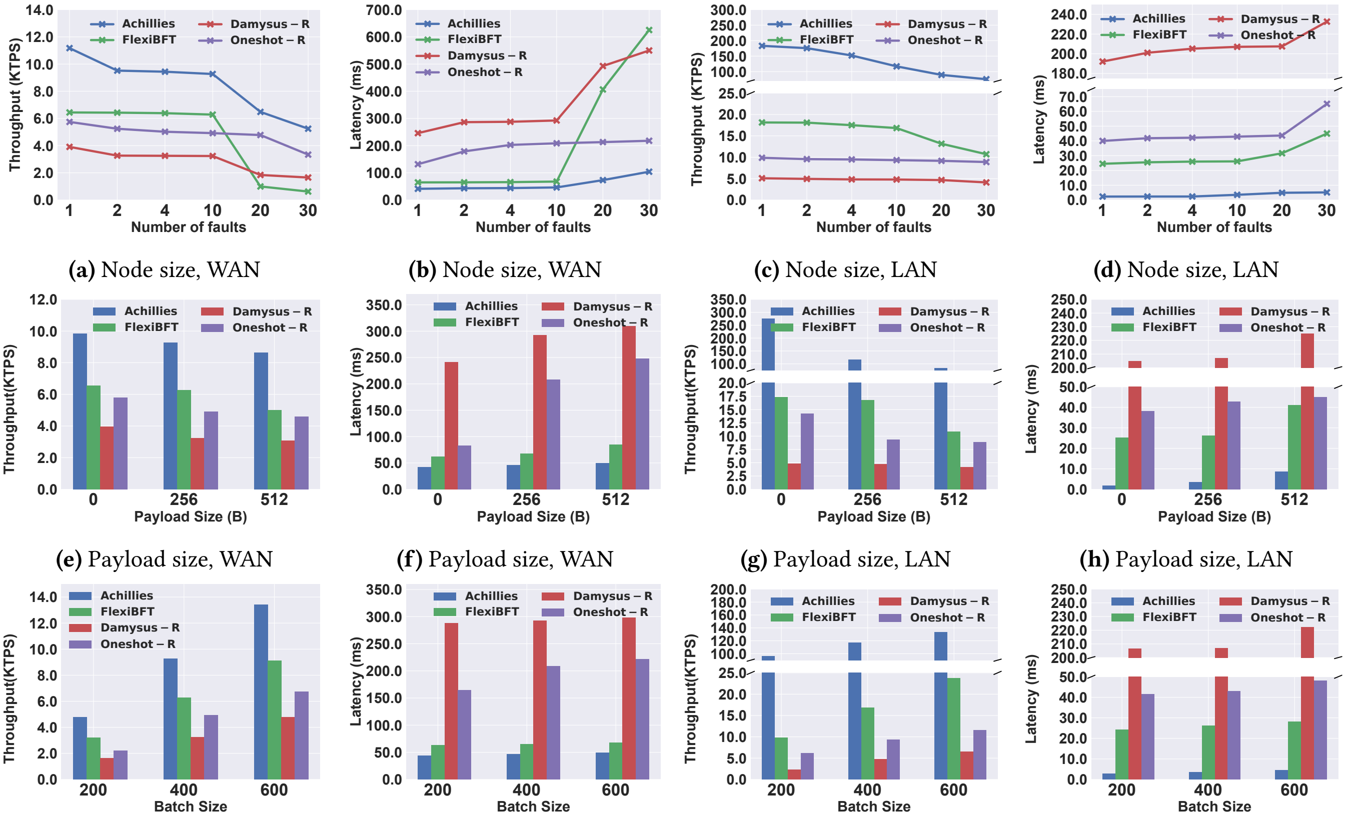
- **Damysus-R.** Damysus-R is a variant of Damysus with rollback prevention. Specifically, only the CHECKER component is required for rollback prevention (Sec. 4.3). Thus, when the CHECKER component is used, it has to store its state with a persistent counter on the disk and then increase the counter by one. In our experiments, we run the chained version of Damysus because it uses the pipelining structure for better performance.
- **FlexiBFT.** For a fair comparison, we implement FlexiBFT on the same platform. Besides, the realization of FlexiBFT also uses chaining structure as Damysus and ACHILLES to serially commit blocks, however, adopts a stable leader pattern, *i.e.*, a leader can continuously propose new blocks without triggering timeouts. We choose FlexiBFT as counterparts to mainly illustrate that even with less usage, the low-performance counters still have a significant impact on its performance, especially in LAN.
- **OneShot-R.** Similar to Damysus-R, OneShot-R is a variant of OneShot with rollback prevention, which uses counters to protect the CHECKER component.

Note that in our implementation, we follow FlexiBFT to emulate the access costs to counters without restricting the counter type (like hardware- or software-based counters in Sec. 2.1). This emulation approach offers flexibility, avoiding the dependency on a specific implementation.

Experimental setup. We conducted all experiments on public cloud service. We rent up to 91 SGX-enabled instances, with one instance per node, and all processes run on dedicated virtual machines that are equipped with 8vCPUs and 32 GB RAM, running Ubuntu Linux 20.04. Each instance is equipped with one private network interface with a bandwidth of 10Gbps. We consider two deployment scenarios: local area network (LAN) and wide area network (WAN). The inter-node RTT in the LAN environment is 0.1 ± 0.02ms. Due to the restricted locality of SGX-enabled instances, we use NetEm [30] to simulate a WAN environment with 40 ± 0.2ms inter-node RTT.

[3] https://github.com/1wenwen1/Achilles.

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



(a) Node size, WAN (b) Node size, WAN (c) Node size, LAN (d) Node size, LAN

(e) Payload size, WAN (f) Payload size, WAN (g) Payload size, LAN (h) Payload size, LAN

(i) Batch size, WAN (j) Batch size, WAN (k) Batch size, LAN (l) Batch size, LAN

Figure 3. Throughput and latency of Achilles with varying parameters in WAN and LAN.

Parameter settings. We vary the fault threshold $f \in \{1, 2, 4, 10, 20, 30\}$ with blocks of 200, 400, and 600 transactions (*i.e.*, batch size), and use transaction payloads of 0 B, 256 B, and 512 B. The total nodes are $2f + 1$ for Achilles, Damysus-R and OneShot-R, and $3f + 1$ for FlexiBFT. Except for transaction payload, each block includes an additional 8 B for metadata (*i.e.*, client and transaction IDs). Payloads of 0 B and transaction numbers of 400 are used to evaluate the protocols' overhead, while other sets of payloads and transaction numbers have been selected to observe the trend when increasing the size of blocks.

Damysus-R, FlexiBFT, and OneShot-R use persistent counters for rollback prevention. To fairly compare with other TEE-assisted BFT protocols using persistent counters for rollback prevention, we have measured their latency for state write/read operations, as shown in Table 4 in Appendix C.1 [47]. By the table, we set the latency of the write operations of counters to 20 ms. We also consider writing latency of $\{0, 10, 20, 40, 80\}$ ms, and evaluate the performance of these protocols in Appendix C.2 [47].

Performance metrics. We consider three performance metrics: 1) throughput: the number of transactions delivered to clients per second, 2) latency (or end-to-end latency): the average delay from when clients create transactions to when these transactions are executed and replied, and 3) commitment latency: the average delay from when transactions are proposed from the leader to when these transactions are executed. In most experiments, we follow Damysus by using commit latency to evaluate Achilles and its counterparts, which can alleviate the impact of clients to provide a more fair comparison. Instead, we use end-to-end latency in Fig. 4 to illustrate the system scalability.

5.2 Fault-Free Performance

5.2.1 Performance in WAN.

We evaluate Achilles and its counterparts in WAN with varying parameters.

1) Varying number of nodes. Fig. 3a and 3b show the throughput and latency of the three protocols with varying fault threshold f. All protocols adopt 400 transactions per block and 256 B payload for each transaction. The throughput of Damysus-R, FlexiBFT and OneShot-R is lower than Achilles, with Damysus-R having the lowest throughput because each node needs to access the expensive persistent counter twice to commit a transaction, whereas only the leader in FlexiBFT needs to access the persistent counter once. When f increases to 20, the throughput of FlexiBFT is lower than that of Damysus-R because the total number of nodes for FlexiBFT is $3f+1$ rather than $2f+1$ in Damysus-R, restricting its scalability with an increase in faults. Similarly, Achilles maintains the lowest latency, whereas Damysus-R has the highest latency when the number of nodes is small. However, as the number of nodes increases, the latency of FlexiBFT increases noticeably because the total number of nodes $3f+1$ makes the increase in faults have a more significant impact on FlexiBFT's latency.

2) Varying payload size. Fig. 3e and 3f show the performance results of three protocols with varying sizes of payloads. The payloads are 0 B, 256 B, and 512 B. The number of faults is 10, and the batch size is fixed at 400. The results show that when the payload increases from 0 B to 512 B, the throughput of the three protocols decreases by approximately 10%, while the latency increases by about 10%. This shows that the payload size has a relatively small impact on the performance of the three protocols in a WAN environment.

3) Varying batch size. Fig. 3i and 3j illustrate the impact of varying batch sizes on the performance of three protocols. The number of faults is 10, the payload is 256 B, and the batch size varies from 200, 400, to 600. As the batch size increases from 200 to 600, the throughput of the three protocols increases significantly by approximately 180%. Latency also increases slightly, with the latency of Achilles increasing by 11.2%, Damysus-R by 3.7%, FlexiBFT by about 6.6%, and OneShot-R by about 3.5%. This shows that the increase in batch size significantly boosts the throughput of the three protocols while also causing a slight increase in latency.

5.2.2 Performance in LAN. To minimize the effect of network communication, we also evaluate Achilles in LAN.

1) Varying number of nodes. Fig. 3c and 3d show the throughput and latency of all protocols in LAN, respectively. As the network communication cost is negligible in a LAN environment, the impact of the cost of accessing the persistent counter becomes more significant. This causes Damysus-R, FlexiBFT, and OneShot-R to maintain a relatively low throughput. As the number of faults increases from 1 to 30, the throughput of Damysus-R, FlexiBFT and OneShot-R decreases by 19.2%, 41.0%, and 10.0%, respectively, while latency increases by 83.6%, 21.1%, and 62.9%, respectively. Because the cost of the persistent counter becomes the dominator, the increase in faults only slightly affects the throughput and latency of Damysus-R, FlexiBFT, and OneShot-R. In contrast, Achilles exhibits significantly higher performance without using the persistent counter. The throughput of Achilles is approximately 18 to 36 times that of Damysus-R, 7 to 10 times that of FlexiBFT, and 8 to 18 times that of OneShot-R.

2) Varying payload size. Fig. 3g and 3h show the performance results of three protocols with varying sizes of payloads with the same setting in WAN. The settings are the same as those in WAN to evaluate the impact of payload size. Similarly, access to the persistent counter is the dominator of performance in Damysus-R, FlexiBFT, and OneShot-R, the increase in payloads has a relatively small effect on their performance. As the payloads increase from 0 B to 512 B, the throughput of Damysus-R decreases by 13.5%, and latency increases by 10.0%. For FlexiBFT, the throughput decreases by 37.2%, and latency increases by about 63.4%. For OneShot-R, the throughput decreases by about 37.6%, and latency increases by about 18.1%. In contrast, the increase of payloads causes more significant changes in throughput and latency of Achilles, with throughput decreasing by approximately 70%, and latency increasing by about 300%.

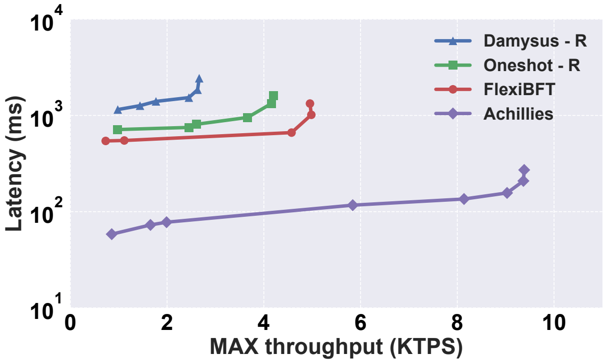


Figure 4. Latency vs. throughput of Achilles in LAN.

3) Varying batch size. Fig. 3k and 3l illustrate the impact of varying batch sizes on the throughput and latency of three protocols. The number of nodes is 1500, the payload is 256 B, and the batch size varies from 200, 400, to 600. Similar to the results in the WAN setting, the increase in batch size significantly boosts the throughput of the three protocols while slightly increasing latency. Additionally, in the LAN setting, due to the significant impact of the persistent counter cost, Achilles's throughput and latency are significantly better than those of the other three protocols when the batch size ranges from 200 to 600.

5.2.3 Throughput vs. latency. Fig. 4 illustrates the latency of the three protocols with increasing throughput until system saturation. The fault threshold is set to 10, the payload is 256 B, and the batch size is 400. The results show that the maximum throughput of Achilles is 9.38K. FlexiBFT experiences a decrease in throughput and an increase in latency because the leader in each view needs to access a persistent counter, and the total number of nodes required is $3f + 1$, resulting in a maximum throughput of 4.95K. OneShot-R performs worse than FlexiBFT, with a maximum throughput of 4.23K, since every node in each view must access a persistent counter. Damysus-R shows the lowest throughput with a maximum of 2.66K and the highest latency. This is due to the additional communication rounds required compared to OneShot-R, as well as the need for each node to access the persistent counter. This demonstrates that Achilles achieves significantly better performance compared to the other two protocols due to its optimal resilience of $2f + 1$ and the lack of need to access a persistent counter.

Table 2. Breakdown of recovery overhead in LAN.

Nodes Latency (ms)	3	5	9	21	41	61
Initialization	8.1	10.7	12.8	15.1	22.0	24.8
Recovery	0.61	1.19	1.49	6.56	10.56	12.31
Total	8.68	11.88	14.28	21.63	32.60	37.09

5.3 Recovery Overhead

We evaluate the recovery overhead of nodes in a LAN environment. We measure the time taken from when a node reboots its trusted components in TEEs to when it completes the recovery and joins in the normal-case operations. The recovery process mainly contains two parts. The first is the initialization process, in which a node establishes connections and restarts Intel SGX. The second is that a node completes the procedure in Algorithm 3. Table 2 lists the time taken for the recovery with varying numbers of nodes. As we can see, with the increase in nodes, the time consumed by initialization and the recovery protocol only shows a slight increase. This demonstrates that the recovery overhead of Achilles is relatively small, allowing a node to recover and rejoin the system quickly.

5.4 Overhead Profiling

To better understand the overhead of using SGX, we implement a new variant of Achilles, called Achilles-C, which operates the trusted components outside the SGX enclave. Achilles-C could be viewed as a chained version of CFT protocols. The comparison with Achilles-C highlights the overhead introduced by using SGX. Furthermore, we compare Achilles with BRaft (version: 1.1.1) [2], an open-source implementation of the Raft protocol. The comparison with BRaft aims to illustrate the cost of BFT guarantees of Achilles given the state-of-the-art CFT protocols.

Table 3 lists the maximum throughput and latency of these protocols with varying f in a LAN network. The batch size is 400 and the payload size for each transaction is 256 B. The evaluation results also demonstrate that Achilles can achieve 76.3% and 97.3% of the throughput of Achilles-C and BRaft in the setting of $f = 10$, respectively, while maintaining the security benefits provided by SGX.

6 Discussion

6.1 Responsiveness and Parallelism Issues

Gupta *et al.* [28] identify restrictive responsiveness and lack of parallelism issues of existing TEE-assisted BFT protocols. To address them, Gupta *et al.* lower tolerance thresholds from $n = 2f + 1$ to $3f + 1$ (See more in Sec. 1.). However, Bessani *et al.* [10] shows that tolerance relaxation is not necessary, and some simple modifications can easily address these two issues. Next, we introduce these modifications, especially for the adopted ones in Achilles.

Table 3. Overhead profiling for Achilles in LAN.

Protocols	Throughput (KTPS)			Latency (ms)		
	$f = 2$	$f = 4$	$f = 10$	$f = 2$	$f = 4$	$f = 10$
Achilles	175.8	152.1	116.9	2.3	2.4	3.5
Achilles-C	216.2	200.6	153.2	2.2	2.3	2.9
BRAFT	298.2	236.6	120.1	2.3	2.4	2.5

Restrictive responsiveness. In TEE-assisted BFT protocols, a client usually receives $f+1$ replies from consensus nodes to ensure a transaction is committed. However, if a transaction is committed by a quorum of $f + 1$ nodes, maybe only one honest node replies to the client. Therefore, the client cannot collect enough replies until all honest nodes synchronize with the commitment by checkpoints, resulting in restrictive reply responsiveness.

Bessani *et al.* [10] points out that this issue is not specific to using trusted components. This issue has been reported in PBFT [15] and BFT-SMaRt [11], in which transaction read can be optimized to avoid running consensus by collecting $n-f$ replies [9]. To address the issue, there are two simple solutions. First, when a node commits and replies to the client, it can broadcast a *Decision* message to other nodes to remind them [7, 44]. Second, the leader can execute transactions and include results in blocks. When the block is committed, any node can send a certificate to clients to certify the results. This method is widely used in blockchains like Ethereum [65]. Achilles adopts the second one and leverages the integrity of TEEs to avoid using certificates.

Lack of parallelism. TEE-assisted BFT protocols that adopt equivocation detection [39, 68] require nodes to serially access trusted components to certify messages, enforcing a gapless sequence of messages [10]. This serialization prohibits the parallelism of protocols like PBFT [15]. Bessani *et al.* [10] shows that techniques like pipelining, concurrent consensus instances, and consensus-oriented parallelization can mitigate the parallelism issue. Achilles follows the chaining spirit of chained BFT protocols to realize linear message complexity and support frequent leader rotation. The chain structure, *i.e.*, linking batched transactions in blocks, can already achieve good scalability and performance. We can further parallel Achilles by concurrent consensus instances [27, 40, 55], which is left for further work.

6.2 Dynamic Reconfiguration

Achilles does not consider dynamic reconfiguration, which allows nodes to be dynamically added or removed over time. Although reconfiguration in BFT consensus has been well explored in prior work [11, 22], integrating dynamic reconfiguration with the recovery procedure in the presence of rollback issues remains challenging for Achilles. This is because a rebooting node relies on configuration information to determine which node the recovery requests should be sent to. Due to rollback issues, the node may use stale configuration information, potentially joining an old group of nodes to process client transactions. This may violate the safety property. Additionally, if reconfiguration occurs during recovery and some nodes from the previous configuration are removed, the rebooting node may fail to gather enough recovery responses, violating the liveness property. Thus, implementing dynamic reconfiguration while avoiding these security flaws requires substantial effort and is left for future work.

6.3 Excessive Faults

In Achilles, we assume no more than f nodes reboot concurrently. Without the assumption, the system may lose liveness since no node can recover from collecting $f + 1$ replies. However, this limitation is not unique to our work. Diskless CFT protocols without stable storage, such as VR [38] and variants of Paxos [16, 35], also share this constraint (no more than f crashed nodes concurrently). Moreover, all BFT protocols have a security threshold f. An adversary compromising more than f nodes would disrupt system liveness/safety. This also holds true for Achilles.

7 Related Work

7.1 Hardware-Assisted BFT Consensus

Hardware-assisted BFT consensus leverages trusted hardware components to enhance the performance and fault tolerance of BFT consensus. Specifically, trusted hardware components can be categorized into: 1) small trusted hardware [66], which provides small trusted abstractions such as append-only log and monotonic counter, and 2) TEEs [31], which support computing arbitrary functions in a trusted manner. Small trusted hardware with a small Trusted Computing Base (TCB) can be realized by Trusted Platform Modules (TPMs) [50, 57, 58] and YubiKey [52].

BFT consensus using small trusted hardware. Chun et al. [18] pioneered the usage of trusted logs to prohibit Byzantine behaviors (*e.g.*, proposal and vote equivocation), which improves the fault tolerance of corrupted nodes from one-third to the minority. Levin *et al.* [36] simplifies the trusted log abstraction to a trusted persistent counter within the same security guarantee. Later, MinBFT [62] and CheapBFT [33] further advance system performance by optimizing the fast path and happy path. Recently, Yandamuri *et al.* [66] improve the resilience of HotStuff from one-third to $1/2 - \epsilon$, while keeping a total of $O(n)$ communication per view in a partially synchronous network.

BFT consensus using TEEs. Hybster [8] explored the potential for parallelizing consensus instances through TrIncX, a TrInc-like trusted counter. FastBFT [39] uses TEEs to realize a secret sharing scheme, by which it can achieve $O(n)$ communication complexity in the normal phase. TBFT [68] replaces the broadcasting communication pattern (as in PBFT) with a leader-based pattern (as in HotStuff), by which it achieves $O(n)$ communication complexity in the normal phase. TBFT also implements a trusted message-sharing mechanism to generate quorum certificates from messages collected from $f + 1$ nodes. Recently, three state-of-the-art protocols, Damysus [20] and FlexiBFT [24], and OneShot [21] significantly advance the TEE-assisted BFT consensus. See detailed description of them in Sec. 2.2. Despite these advancements, existing TEE-assisted BFT protocols are still inefficient in rollback prevention and performance (*i.e.*, message complexity and latency), as discussed in Sec. 2.

7.2 Confidential BFT Consensus.

Except for utilizing TEEs' integrity, BFT consensus can also use TEEs' confidentiality to protect transaction privacy [13, 53, 63]. Specifically, confidential BFT consensus usually runs the whole procedure including transaction ordering and execution within TEEs. For example, CCF [53] is a framework for providing confidential services in permissioned blockchains by maintaining a distributed key-value store inside enclaves. Brandenburger *et al.* [13] introduces an architecture and a prototype for smart-contract execution within Intel SGX for Hyperledger Fabric [6]. Wang *et al.* [63] propose Engraft, which runs Raft within TEEs with additional rollback prevention and liveness enhancement. Confidential BFT protocols also suffer from the performance-tolerance tradeoff. Thus, they can extend the rollback resilient recovery of Achilles. However, designing efficient recovery for them is challenging since they usually maintain the whole system state within TEEs.

8 Conclusion

We propose Achilles, an efficient TEE-assisted BFT protocol that adopts a rollback resilient recovery to break the tradeoff between performance and tolerance. Achilles also leverages chained commit rules to achieve linear message complexity and end-to-end transaction latency of four communication steps, making it the first TEE-assisted BFT to match the efficiency of CFT protocols like Raft. Extensive experimental results demonstrate that Achilles significantly outperforms state-of-the-art TEE-assisted BFT protocols.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Jialin Li, for their insightful comments. We thank Fan Zhang and Tong Liu for their helpful feedback. This work is supported by NSFC under Grants 62302204 and 62361166633; in part by National Key Research and Development Program of China under Grant 2023YFB4503902; in part by the Shenzhen Science and Technology Program under Grants RCBS20221008093248075; and in part by the Australian Research Council (ARC) under projects DE210100019 and DP220101234.

References

[1] Achilles - Wikipedia. https://en.wikipedia.org/wiki/Achilles. Retrieved Feb, 2023.

[2] BRAFT. https://github.com/baidu/braft. Retrieved January, 2025.

[3] Intel SGX OpenSSL. Availableathttps://github.com/intel/intel-sgx-ssl. Retrieved May, 2023.

[4] Salticidae: minimal C++ asynchronous network library. https://github.com/Determinant/salticidae. Retrieved May, 2023.

[5] SGX documentation: SGX create monotonic counter. https://software.intel.com/en-us/node/696638. Retrieved Jun, 2022.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proc. of EuroSys*, 2018.

[7] Roberto Baldoni, Jean-Michel Hélary, Michel Raynal, and Lenaik Tangui. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003.

[8] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-based high performance BFT. In *Proc. of EuroSys*, 2017.

[9] Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *Prof. of SRDS*, pages 1–12, 2021.

[10] Alysson Bessani, Miguel Correia, Tobias Distler, Rüdiger Kapitza, Paulo Esteves-Verissimo, and Jiangshan Yu. Vivisecting the dissection: On the role of trusted components in BFT protocols. *arXiv preprint arXiv:2312.05714*, 2023.

[11] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Proc. of DSN*, 2014.

[12] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for Trusted Execution Environments using lightweight collective memory. In *Prof. of DSN*, 2017.

[13] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Trusted computing meets blockchain: Rollback attacks and a solution for Hyperledger Fabric. In *Proc. of SRDS*, 2019.

[14] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[15] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.

[16] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proc. of PODC*, 2007.

[17] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *Prof. of EuroS&P*, 2019.

[18] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, October 2007.

[19] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proc. of PODC*, 2012.

[20] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined BFT consensus leveraging trusted components. In *Proc. of EuroSys*, 2022.

[21] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Oneshot: View-adapting streamlined BFT protocols with Trusted Execution Environments. In *Proc. of IPDPS*, 2024.

[22] Sisi Duan and Haibin Zhang. Foundations of dynamic BFT. In *Proc. of S&P*, 2022.

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[24] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the performance of chained-BFT. In *Proc. of ICDCS*, 2021.

[25] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Scaling blockchain consensus via a robust shared mempool. In *Prof. of ICDE*, 2023.

[26] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. of SOSP*, 2017.

[27] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient concurrent consensus for high-throughput secure transaction processing. In *Proc. of ICDE*, 2021.

[28] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. Dissecting BFT consensus: In trusted components we trust! In *Proc. of EuroSys*, 2023.

[29] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[30] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, page 2005, 2005.

[31] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of HASP*, 2013.

[32] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-HotStuff: A fast and robust BFT protocol for blockchains. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[33] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proc. of EuroSys*, 2012.

[34] David Kaplan, Jeremy Powell, and Tom Woller. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. Technical report, Advanced Micro Devices Inc., 2020.

[35] Jan Kończak, Nuno Filipe de Sousa Santos, Tomasz Żurkowski, Paweł T Wojciechowski, and André Schiper. Jpaxos: State machine replication based on the paxos protocol. 2011.

[36] Dave Levin, John (JD) Douceur, Jay Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proc. of NSDI*, 2009.

[37] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In *Proc. of SOSP*, page 63–79, 2019.

[38] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[39] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68:139–151, 2019.

[40] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Chen Feng, Yinqian Zhang, and Ivan Beschastnikh. Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering. In *EuroSys*, 2025.

[41] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive BFT. *Cryptology ePrint Archive*, 2023.

[42] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. ADAM-CS: Advanced asynchronous monotonic counter service. In *Prof. of DSN*, pages 426–437, 2021.

[43] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *Proc. of USENIX Security*, 2017.

[44] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $o(n^2)$ messages, and $o(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):1–21, 2015.

[45] Ray Neiheiser, Miguel Matos, and Luís E. T. Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *Proc. of SOSP*, 2021.
[46] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. Narrator: Secure and practical state continuity for trusted execution in the cloud. In *Proc. of CCS*, 2022.
[47] Jianyu Niu, Xiaoqing Wen, Guanlong Wu, Shenqi Liu, Jianshan Yu, and Yinqian Zhang. Achilles: Efficient TEE-Assisted BFT Consensus via Rollback Resilient Recovery. https://github.com/1wenwen1/Achilles/tree/main/doc. Retrieved Feburary, 2025.
[48] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. of ATC*, pages 305–319, June 2014.
[49] Alina Oprea and Michael K Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proc. of USENIX Security*, pages 183–198, 2007.
[50] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proc. of S&P*, 2011.
[51] Wei Peng, Xiang Li, Jianyu Niu, Xiaokuan Zhang, and Yinqian Zhang. Ensuring state continuity for confidential computing: A blockchain-based approach. *IEEE Transactions on Dependable and Secure Computing*, pages 1–14, 2024.
[52] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent Seamons. A tale of two studies: The best and worst of YubiKey usability. In *Proc. of S&P*, 2018.
[53] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. CCF: A framework for building confidential verifiable replicated services. *Technical report, Microsoft Research and Microsoft Azure*, 2019.
[54] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proc. of STC*, 2006.
[55] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-BFT: Scalable and robust BFT for decentralized networks. In *JSys*, 2022.
[56] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proc. of EuroSys*, 2022.
[57] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Proc. of ACSAC*, 2014.
[58] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *Proc. of USENIX Security*, 2016.
[59] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proc. of SOSP*, 2021.
[60] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. Design and evaluation of IPFS: a storage layer for the decentralized web. In *Proc. of SIGCOMM*, 2022.
[61] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proc. of STC*, 2007.
[62] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.
[63] Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *Proc. of CCS*, 2022.
[64] Weili Wang, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Formally verifying a rollback-prevention protocol for tees. 2024.
[65] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger Byzantium version. *Ethereum project yellow paper*, pages 1–32, 2018.
[66] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K. Reiter. Communication-efficient BFT using small trusted hardware to tolerate minority corruption. In *Proc. of OPODIS*, 2023.
[67] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. of PODC*, 2019.
[68] Jiashuo Zhang, Jianbo Gao, Ke Wang, Zhenhao Wu, Yue Li, Zhi Guan, and Zhong Chen. TBFT: Efficient Byzantine fault tolerance using Trusted Execution Environment. In *Proc. of ICC*, 2022.

A Artifact Appendix

This appendix provides a detailed guide on reproducing the results presented in our paper. We build a prototype of Achilles based on the Damysus framework [20] and follow most of its experimental setup. Below, we describe how our system is structured and the steps necessary to reproduce our results.

A.1 Abstract

Achilles is an efficient TEE-assisted BFT protocol that provides a state-machine replication service. The main novelty is the rollback resilient recovery and the optimization of the normal-case operations. We implement Achilles based on the Chained-Damysus, the chained version of Damysus [20]. We add a macro named `ACHILLES` for Achilles in the code, and its corresponding implementation is located under `#if defined(ACHILLES)`. The extension includes modifying the trusted components, *i.e.*, Checker and Accumulator (Sec. 4.3), and reducing the three-phase normal-case operations of Damysus to two-phase. We also add a rollback resilient recovery for rebooting TEEs.

A.2 Description & Requirements

A.2.1 How to access. The code used to produce the results of the experiments is publicly available in Github repository[5] and Zenodo[6], which has 5 branches: Achilles, FlexiBFT, Damysus, Oneshot, and Achilles-Recovery. The last one is to evaluate the performance of the recovery process.

A.2.2 Hardware dependencies. We performed our evaluation on Ali Cloud ECS machines with one ecs.g7t.large SGX-enabled instance per node in the Hong Kong area. All processes run on dedicated virtual machines with 8vCPUs and 32GB RAM running Ubuntu Linux 20.04. Each machine is equipped with one private network interface with a bandwidth of 5 Gbps. Due to the restricted locality of SGX-enabled instances, we use NetEm [30] to simulate a LAN environment with 0.1 ± 0.02ms inter-node RTT and a WAN environment with 40 ± 0.2ms inter-node RTT. See more in Sec. 5.1.

A.2.3 Software dependencies. C++ 14, Python 3.8.10.

A.2.4 Benchmarks. None.

[5] https://github.com/1wenwen1/Achilles
[6] Persistent ID: 10.5281/zenodo.14830621

A.3 Setup

Detailed setup instructions are available in the repository's README file. The deployment process is automated via scripts located in the deployment directory. These scripts allow you to deploy a network of nodes and clients on any SGX-enabled machine, conduct experiments, analyze results, and generate summary data.

To deploy experiments on Ali Cloud, you must set up an Ali Cloud account and register an SSH key. The repository includes scripts to initialize the Ali Cloud ECS instance, streamlining the deployment process. At first, you need to launch an ECS instance as the controller, which is responsible for deploying instances, managing experiments, and handling experimental results. We provide automated scripts and Python files to facilitate instance deployment and experiments. More details on how to use the controller to deploy experiments are introduced in Sec. A.4 and the repository's README file. Although the provided experiment instructions are for Ali Cloud, you can run experiments on any SGX-enabled machines.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Achilles significantly enhances the performance of both Damysus-R, FlexiBFT, and OneShot-R, achieving about 2×, 7× and 0.2× increase in throughput while reducing latency by about 81%, 83%, and 52%,respectively, in WAN experiments with 30 faults. These improvements are demonstrated in the experiment E1 (introduced shortly), with results shown in Fig. 3.
- (C2): Achilles achieves an 17× increase in throughput for Damysus-R, a 6× increase for FlexiBFT, and a 7× increase for OneShot-R, while reducing latency by 98%, 88% and 92%, respectively, in LAN experiments with 30 faults. These improvements are demonstrated in the experiment E2 (introduced shortly), with results shown in Fig. 3.
- (C3): Achilles adopts a rollback resilient recovery: the recovery overhead of Achilles is relatively small since the time consumed by the recovery only shows a slight increase with the increase in nodes. This is demonstrated in the experiment E3 (introduced shortly).

A.4.2 Experiments. We first outline the general steps required to perform Achilles's experiments.
[Preparation] To set up for the experiments, follow these steps:

- **Ali Cloud Account and Configuration**: Log in to Ali Cloud. Enter the Elastic Compute Service interface, choose the region HongKong, and start an instance to serve as the controller.

Table 4. Execution files and corresponding figures

Branches	Execution files	Figures
All branches	scripts/faults_WAN.sh	Fig. 3a, Fig. 3b
All branches	scripts/faults_LAN.sh	Fig. 3c, Fig. 3d,
All branches	scripts/payload_WAN.sh	Fig. 3e, Fig. 3f
All branches	scripts/payload_LAN.sh	Fig. 3g, Fig. 3h,
All branches	scripts/batchsize_WAN.sh	Fig. 3i, Fig. 3j
All branches	scripts/batchsize_LAN.sh	Fig. 3k, Fig. 3l

- **Connect to the controller**: Connect to the controller and all our experimental files are under the Directory `/root/Achilles`.

[Execution] With the preparation complete, navigate to the `/root/Achilles` Directory. If you want to execute the locally with the default config, you can just use the command `python3 run.py −−local −−p1`. If you want to deploy distributed experiments, please proceed with the following steps to execute the experiments:

- **Launch Instances**: Enter the `deployment` Directory. Execute `bash cloud_deploy.sh` to initialize new instances based on the configuration specified in `config.json`. Run `bash cloud_config.sh` to configure the SGX operating environment for these instances. Use command `tmux a` to check the execution of the environment configuration and use command `exit` to exit the tmux terminal.
- **Conduct Experiments**: Return to `/root/damysus_updated` Directory. Run `python3 run.py −−p1 −−faults {faults} −−batchsize {batchsize} −−payload {payload}` to conduct a single experiment. This command performs one experiment using the Achilles protocol. To run a series of experiments, please execute the scripts in the directory `/root/Achilles/scripts/`, as shown in Table 4. For example, running `bash scripts/faults_WAN.sh` will generate the data for the Fig. 3a and Fig. 3b. If an error or issue occurs during operation, use `python3 close.py` to stop the process on each instance.
- **Shutdown Instances**: Use `python3 deployment/delete_instances.py` to terminate all running instances in Ali ECS.

[Results] Upon completion of the experiments, the results will be available in the `stats.txt`. This directory contains a comprehensive set of detailed statistics, including throughput and latency. Review these files to analyze and interpret the outcomes of your experiments. For example, "Achilles_1_256_400_0, 18.1715414, 26.598315" indicates that the throughput and latency for the Achilles protocol, with 1 fault, 400 transactions per block, and a 256 B payload per transaction, are 18.1715414K TPS and 26.598315ms, respectively.

Experiment (E1): [Throughput and latency in fault-free] [1 human-hour + 3 compute-hour]: This experiment is designed to evaluate the system's peak performance in a WAN environment, focusing on throughput and latency. This experiment involves running various scenarios with different configurations, including varying number of faults, payload and batchsize, and comparing the results with other protocols (Fig. 3a, Fig. 3b, Fig. 3e, Fig. 3f, Fig. 3i and Fig. 3j). The corresponding execution files are located in the `scripts/` directory, as shown in Table 4.

Experiment (E2): [Throughput and latency in fault-free] [1 human-hour + 3 compute-hour]: This experiment is similar to Experiment (E1), evaluating the system's peak performance in a LAN environment, focusing on throughput and latency (Fig. 3c, Fig. 3d, Fig. 3g, Fig. 3h, Fig. 3k and Fig. 3l). The execution files are similarly named and located in the `scripts/` directory.

Experiment (E3): [Recovery process] [1 human-hour + 1 compute-hour]: This experiment evaluates the system's robustness in the presence of crash faults (Table2). Switch to the Achilles-Recover branch, and run `python3 runRecover.py ——p4 ——faults faults` to test the initial time and recover time.