Department of Computing
Imperial College London

Auditable and Performant Byzantine Consensus for Permissioned Ledgers

Alexander Shamis

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

May 2023

To Chelsea, Amelia, Henry, Emma, and Batdog

Abstract

Permissioned ledgers allow users to execute transactions against a data store, and retain proof of their execution in a replicated ledger. Each replica verifies the transactions' execution and ensures that, in perpetuity, a committed transaction cannot be removed from the ledger. Unfortunately, this is not guaranteed by today's permissioned ledgers, which can be re-written if an arbitrary number of replicas collude. In addition, the transaction throughput of permissioned ledgers is low, hampering real-world deployments, by not taking advantage of multi-core CPUs and hardware accelerators.

This thesis explores how permissioned ledgers and their consensus protocols can be made auditable in perpetuity; even when all replicas collude and re-write the ledger. It also addresses how Byzantine consensus protocols can be changed to increase the execution throughput of complex transactions. This thesis makes the following contributions:

1. Always auditable Byzantine consensus protocols. We present a permissioned ledger system that can assign blame to individual replicas regardless of how many of them misbehave. This is achieved by signing and storing consensus protocol messages in the ledger and providing clients with signed, universally-verifiable receipts.

2. Performant transaction execution with hardware accelerators. Next, we describe a cloud-based ML inference service that provides strong integrity guarantees, while staying compatible with current inference APIs. We change the Byzantine consensus protocol to execute machine learning (ML) inference computation on GPUs to optimize throughput and latency of ML inference computation.

3. Parallel transactions execution on multi-core CPUs. Finally, we introduce a permissioned ledger that executes transactions, in parallel, on multi-core CPUs. We separate the execution of transactions between the primary and secondary replicas. The primary replica executes transactions on multiple CPU cores and creates a dependency graph of the transactions that the backup replicas utilize to execute transactions in parallel.

Acknowledgements

While there are too many people to thank individually and without whose help, guidance, and assistance this thesis would not be possible. I would like to mention the following in no particular order:

Professor Peter Pietzuch. Peter, thank you for the advice, support, patients, and understanding with me working through my PhD while also working at Microsoft Research. I have learnt so much about being a researcher from you and the ins and out of the academic process. Your mentoring, guidance, and encouragement has been invaluable.

Dr. Miguel Castro. Miguel, thank you for your help and support to enable this PhD journey to begin, and for the advice, guidance, help, and support all the way through. This thesis would not have even started without our conversation at the FaRM offsite in early 2016. However, I am extremely grateful that it did and for all your efforts that made this possible.

Dr. Manuel Costa and Microsoft Research Cambridge. Manuel and the CCF team thank you for giving me the opportunity to do the work that allowed me to write this thesis. I appreciate your support, technical discussions, and feedback. I have learnt so much from all of you and it has been a privilege to be a member of the Confidential Computing group in Microsoft Research Cambridge.

My colleagues. To my colleagues, whether you are at Imperial College or Microsoft, thank you. It has been a real privilege to work with you.

My parents and sister. Igor, Alla, and Greta thank you for your support and encouragement. It has been a long journey from my first memories of Kolomenskoye, Shepetovka, and Kiev to Sydney, Seattle, and then Cambridge. Thank you for the support along this journey.

My family. Amelia, Henry, and Emma while you will not remember most of this journey, for me, you are a vital part of it. Batdog, thank you for the emotional support and spend countless hours waiting with me while I worked, I will miss you.

My wife. Chelsea, without you this thesis and the PhD journey would not have been possible. You were there for me in the highs and the lows and for that, I cannot even begin to thank you. So let me say, thank you for being beside me when we began this journey in Seattle and then in

Cambridge and back to America. Thank you for your love and support every step of the way.

This work was supported by Microsoft Research and EPSRC through Microsoft's PhD Scholarship Programme.

Statement of Originality

This thesis presents my work over the period between October 2018 and May 2023. Parts of the work described were done in collaboration with other researchers and engineers:

- **Chapter 3**: The proposal for adding individual accountability to permissioned ledgers originated from conversations with Peter Pietzuch, Miguel Castro, and myself. Peter Pietzuch and Miguel Castro provided advice and suggestions that led to the ideas and solutions explored in this chapter. The IA-CCF implementation was built on top of Microsoft's CCF project [168] and MIT's PBFT [172]. The correctness proof in Appendix B was primarily written by Burcu Canakci as part of her internship at Microsoft Research Cambridge. I was involved in the design, review, and corrections of the correctness proof.
- **Chapter 4**: The idea to combine Byzantine fault tolerance and deep neural network (DNN) inference was developed with Peter Pietzuch. Antoine Delignat-Lavaud and Andrew Paverd suggested refinements to the distance function in Section 4.3.1.
- **Chapter 5**: The idea and work from this chapter occurred as part of the work to build IA-CCF (Chapter 3). The idea to create a dependency graph of transaction requests came from conversations between Miguel Castro and myself.

I declare that the work presented in this thesis is my own, except where declared above

Alexander Shamis
May 2023

Copyright Statement

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

CHAPTER 1

Introduction

Storing and retaining information within a software system is the cornerstone of modern computing. Users expect that, when they write data to a storage system, it will be retained in a pristine manner and can be read back at any time. While accidental failures have become incredibly rare, we observe an increasing number of situations where data is changed for malicious or malignant purposes. The rise of blockchains has arrived at a time when the general public is beginning to not trust data storage, and blockchains promise to address the issue, re-introducing and re-invigorating the public's trust in systems that rely on storing data securely.

1.1 Overview

Blockchains and permissioned ledgers are a decade old technology that has captured the imagination of industry [178, 249, 97, 87], governments [22, 212, 60, 71], and the public at large [202, 213]. Blockchains and permissioned ledgers include a replicated and immutable ledger that contains data that cannot be rewritten or deleted in an undetectable manner. The ledger is replicated across a large number of machines to ensure resiliency to failures and dishonest actors. Ledgers can be verified by ensuring that the cryptographic primitives that connect the contents of the ledger have not been tampered with. This verification process uses well known cryptographic operations and can be performed by anyone.

Blockchains are commonly separated into two categories: first are public blockchains where anyone can run a replica and send operations that are executed against the current state of the blockchain. Public blockchains are used in a large number of scenarios, but the most popular use cases are crypto-currencies [2, 129]; private blockchains, also known as permissioned ledgers or private ledgers, associate each user to their real world identity. Maintaining real-world identities allows permissioned ledgers to allow only authorized users that are trusted by the operator to perform operations against the permissioned ledger. Permissioned ledgers are used in a myriad of scenarios, including: maintaining records for supply chains [143, 36]; maintaining ownership of luxury items [225, 12]; or retaining government records [184, 46].

Although blockchains have gained popularity and are deployed in an increasing number of scenarios, they *do not* provide the guarantees that users expect. Blockchains are commonly believed to be immutable, and, as such, users expect that once their transactions are executed, the transactions cannot be removed from the blockchain. This, unfortunately, is not true, as any blockchain's consensus protocol can be compromised by malicious actors taking control over a predefined number of replicas, thus allowing the malicious actors to rewrite the ledger. When the ledger is rewritten, evidence of executed transactions can be removed in such a way that clients cannot prove that their transaction had been previously executed and committed by the blockchain.

In addition, the performance of the consensus protocols used by blockchains and permissioned ledgers hampers their usability and utility. Currently, the two most commonly used blockchains obtain a throughput of fewer than 20 transactions per second [35]. The most commonly used permissioned ledger, Hyperledger Fabric, has a peak throughput of 1,207 transactions per second when running a simple benchmark that simulates a bank in which customers transfer funds between accounts [7]. Permissioned ledgers, even with their higher transaction throughput, do not provide a platform that is usable in many real-world or academic scenarios.

1.2 Brief evolution of blockchains

As of the writing of this thesis, there are currently hundreds of different blockchain designs in existence [178, 253, 11, 97, 87, 53, 125, 158, 140]. The proliferation of blockchain systems gives the impression of a mature eco-system. However, the first blockchain was introduced less than 15 years ago. The first modern blockchain was proposed in 2008 by an individual or group of individuals under the pseudonym Satoshi Nakamoto when they published "Bitcoin: A Peer-to-Peer Electronic Cash System" [178]. The Bitcoin paper is the seminal work on blockchains and has led to the popularity of blockchain technologies and the crypto-currencies that many implement [193].

The key technologies that make up Bitcoin existed before Nakamoto's publication: using cryptography to hide the identity of a payer or payee was described by Chaum in 1983 [51],

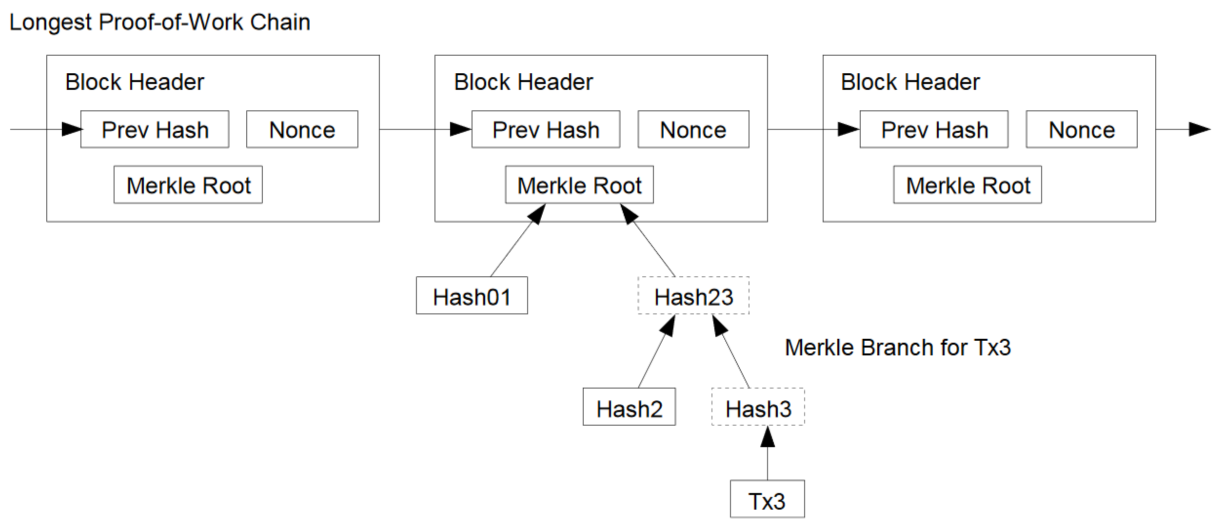


Figure 1.1: Structure of the Bitcoin blockchain. Copied from [178]

and proof-of-work was introduced by Hashcash in 1997 [18]. Nakamoto's key contribution was to bring these disparate ideas together and show their value through a new application, cryptocurrencies. Specifically, the Nakamoto consensus protocol defines the longest chain of blocks as the *valid* version of the blockchain. In Bitcoin, blocks are connected by including a hash of the previous block within the current block creating a chain of blocks. Bitcoin also uses the proof-of-work consensus protocol, which it leverages to include a financial incentive to stop bad actors from attempting to alter the blockchain [178]. Figure 1.1 shows the structure of Bitcoin's blockchain, including how Bitcoin prunes its Merkle tree to reduce the amount of data that needs to be stored within a single block.

After the introduction of Bitcoin, the next major milestone for ledger-based technologies was the introduction of the Ethereum blockchain. The Ethereum project was proposed as a blockchain that can be programmed, i.e., can execute smart contracts, by Vitalik Buterin in 2014 [86]. Buterin's primary contribution was the introduction of today's understanding of smart contracts and their popularization in the Ethereum virtual machine (EVM) [249]. The EVM allows users to submit transaction requests that are written in a higher-level language that compiles into EVM byte code, e.g., Solidity [69]. The design of the EVM guarantees that there are no memory races or undefined behaviour. This is achieved by limiting the API surface area so that transaction execution cannot escape the sandbox of the virtual machine. These safety features have a negative effect on performance, resulting in applications that execute on top of the EVM, having runtime latencies several orders of magnitude higher than if the same application is written in C++.

1.2.1 Consensus protocols

The growth of Bitcoin's popularity has resulted in its low throughput and high latency being brought to the forefront of blockchain research. Bitcoin and its consensus protocol (proof-of-work) does not provide the performance its users and the research community target [68]. In addition, the blockchain community is motivated to show that Bitcoin or an alternative blockchain could replace Visa, Paypal, or another electronic payment system. To quantify this goal, the Visa USA website states: "VisaNet handles an average of 150 million transactions every day and is capable of handling more than 24,000 transactions per second" [241]. This is contrasted by Bitcoin transaction throughput of a "sustained rate of 7 transactions per second" [248].

The throughput of Bitcoin and other proof-of-work based blockchains is governed by the size of the data section in a block and the rate at which blocks are created (mined). In the case of Bitcoin, a block is 1 MB, and a new block is added approximately every 10 minutes. This selection was made to balance the transaction rate and the time it takes for a block to be considered stable while preventing malicious actors from overwhelming the system with new blocks [248]. There are many proposals to tune these parameters, but none increase the throughput to the desired level while maintaining the security properties that the community requires [103, 33, 32].

While tuning proof-of-work was initially done to improve the throughput of Bitcoin and Ethereum, other researchers attempted to address issues inherit to the consensus protocol. The primary issue that was identified is the amount of computational power and electricity required to mine a block — currently, mining consumes more electricity than Finland [233]. As such, the consensus protocols used in blockchain designs include:

- **Proof-of-Work** was initially suggested in 1992 as a method to combat email spam. The proposal requires that a machine, which wants to send an email, solve a cryptographic problem that is unique to every email. The cryptographic problem involves adding additional metadata to an email so that when the email's contents along with the metadata is hashed, it produces the desired hash value [81]. While this technique never gained popularity as it was originally intended, it was popularized by Nakamoto in the Bitcoin paper [178].

- **Proof-of-Elapsed-Time** (commonly referred to as PoET) was introduced by Intel. Its key insight is that proof-of-work is a way of demonstrating that a miner has waited a specific amount of time between creating new blocks. PoET achieves the same effect by using an application that is run within a trusted execution environment (TEE) to ensure that a specific amount of time has passed [128]. There are advantages and disadvantages to using this technique; ultimately, the vulnerabilities [238] with trusted execution environments have meant that PoET is not commonly used.

- **Proof-of-Capacity**. Similar to PoET, proof-of-capacity was designed as a solution to address proof-of-work's electricity consumption. It was introduced in 2015 with the same goals as proof-of-work but with the caveat that it leverages storage rather than computational power [82]. While proof-of-capacity has not gained popularity or widespread adoption in industry or academia, it addresses proof-of-work's electricity usage issue [93]. However, it creates an alternative issue in that nodes need an ever increasing amount of storage to participate in the consensus protocol.
- **Proof-of-Stake** is currently an umbrella term that describes a consensus algorithm that values a miner's contribution when extending a blockchain based on how much of a financial stake they currently have in the blockchain. This technique is used in many scenarios from obtaining consensus when adding a new block to the blockchain [138], to selecting which nodes perform specific roles within the blockchain [97].

Researchers have also explored alternative techniques to address the current design limitations of blockchain consensus protocols, specifically to provide an alternative method of increasing the transaction throughput without replacing proof-of-work. One such research direction has focused on off-Chain payment systems. For example, Teechain describes itself as "the first high-performance micropayment protocol that supports practical, secure, and efficient fund transfers on the current Bitcoin network" and offers off-chain payment system [155]. Teechain starts two nodes that act as representatives between two parties and creates a fast payment transfer channel between the nodes. It guarantees security by running nodes inside Intel's SGX enclaves allowing users to attest the code that is executed. Nevertheless, even off-chain payment systems such as Teechain are unable to achieve the previously mentioned through of Visa. Teechain reports 2480 tx/s as compared to Visa's 24,000 tx/s. In addition, Teechain does not include all transactions executed in the blockchain's ledger, thus reducing transparency.

Alternatively, the authors of Bitcoin-NG [87] assume that mining solves two different purposes: it is used as a way to propose new transactions that are added to the blockchain; and decides which node can propose a new block. Bitcoin-NG proposes a separation of the two concerns such that miners mine blocks as a means to becoming the primary replica during a 10 minute interval. When a miner wins the race to become the primary replica, they are allowed to add new blocks at any rate. While this proposal does not fix all of the issues with Bitcoin, the idea of using proof-of-work to elect leaders has influenced the blockchain consensus protocol research space.

1.2.2 Byzantine fault tolerant consensus protocols

One of the earlier and more influential works in combining blockchains and a Byzantine fault-tolerant consensus protocol is ByzCoin [139]. This system claims to be the progression of taking a blockchain that uses Practical Byzantine Fault Tolerance (PBFT) [49] and "refine it into ByzCoin". ByzCoin (similar to BitCoin-NG) uses proof-of-work to elect a leader and this leader has the same responsibilities as a leader in PBFT. However, unlike BitCoin-NG, ByzCoin uses a predefined number of previous leaders as replicas for the PBFT protocol. This group of replicas is responsible for committing client transactions and ensuring that they appear in ByzCoin's ledger. ByzCoin has a quorum of members that collectively sign transactions and stores these signatures in the ledger allowing the decisions of the PBFT protocol to be verified at any point in the future. The authors of ByzCoin report that they are able to commit up to 1000 transactions per second with a consensus group of 144 machines. Unfortunately, this transaction rate is too slow for many practical scenarios including the previously mentioned Visa scenario.

There have been several systems that follow on from ByzCoin of which Algorand [97] has been one of the more influential. Algorand employs a probabilistic Byzantine agreement protocol where its key novelty is the selection of nodes that participate in the protocol. The system separates the consensus protocol into multiple rounds. Algorand randomly and secretly selects the participants for each round of voting and only allows a selected node to cast a single vote. This secrecy and selection of voting participants allow for a smaller number of nodes to vote in the consensus rounds while maintaining an acceptable level of security. This, in part, leads to better performance than comparable systems. Algorand self reports to be 125 times faster than BitCoin. This, however, is still too slow to meet the performance requirements of the Visa scenario.

1.3 From public blockchains to permissioned ledgers

Permissioned ledgers commonly use a Byzantine consensus protocol in lieu of proof-of-work, allowing them to increase the number of transactions that can execute in a second. Access to the permissioned ledger is controlled and tied to real world identities, thus addressing some of the privacy issues inherent to public blockchains. Tying a real world identity to a user adds a real-world trust component into the system. Users trust a subset of the entities running the blockchain, so that they will not collude with one another and maliciously rewrite the ledger. This trust is derived from the ability to use a user's identity in legal contracts and other means of enforcement within the judicial system.

The first permissioned ledgers were repurposed public blockchains run by organizations within their secured environments [196]. Permissioned ledgers then evolved by changing their consensus protocols and adding additional authentication, tying user accounts to a real world identity. The reason for changing the consensus protocol was to reduce the costs associated with proof-of-work. The consensus protocol used by the first permissioned ledgers was Practical Byzantine Fault Tolerance (PBFT) [49] and then newly proposed consensus protocols that better suited the permissioned ledgers were utilized [175, 209]

The subsequent major development for permissioned ledgers was the introduction of frameworks for building permissioned ledgers [196, 206], with the most popular being IBM's Hyperledger Fabric. Hyperledger Fabric provides a framework for developers to build applications that maintain their blockchains. The framework provides support for multiple programming languages and hides the complexity of creating a blockchain, replacing nodes that host the application, and distributing the ledger between an evolving set of nodes that may experience liveness and availability issues.

1.4 Real world concerns

The introduction of frameworks for building permissioned ledgers has resulted in a large number of organizations and consortiums considering deploying blockchains. However, concerns hinder their ability to utilize a permissioned ledger some that are technical or non-technical. The non-technical concerns include the need to map legislation that governs an industry to the guarantees provided by a permissioned ledger.

1.4.1 Compliance

The need to prove that a group of organizations are in compliance with government regulations has become an emerging area where organizations utilize permissioned ledgers [43, 84]. However, regulations require that organizations secure their systems against all and any malicious behaviour [89, 132]. The regulation then treats malicious behaviour – regardless of who is responsible – as if it was initiated by the organization affected by the malicious actions. Furthermore, the regulation requires that a senior member of the organization is nominated to be personally responsible for any misbehaviour.

These substantial penalties have resulted in concerns and hesitancy when deploying permissioned ledgers. Specifically, individual organizations are concerned that participation in a permissioned ledger would result in them being blamed for malicious actions performed by other organization(s). This fear is due to a fundamental limitation of permissioned ledgers using Byzantine fault tolerant consensus protocol where if more than $2f$ replicas are dishonest the underlying ledger can be rewritten in an undetectable manner.

1.4.2 Performance

The number of transactions a permissioned ledger can execute in a second is an area of concern for organizations that would like to deploy a blockchain or permissioned ledger [173, 211, 241]. Recent research has addressed many of these throughput issues [199, 262]. However, while the research improved the transactional throughput, it focused on increasing the number of transactions simple transactions that can execute in a second – e.g., transactions that increase a counter [49] or the Smallbank benchmark [7]. These benchmarks do not represent real-work workload and executing more complex transactions would result in the permissioned ledger having unacceptable throughput and latency.

1.4.3 Total order and modern workloads

Permissioned ledgers and Blockchains execute transactions that utilize the CPU. However, many organizations have workloads that rely on more than just CPUs to power their business. Specifically, organizations that rely on artificial intelligence and machine learning workloads [96, 54] require GPUs to execute business-critical workloads. We take for granted that the number of business functions that are powered by machine learning and artificial intelligence will increase, further increasing the need for GPUs and other accelerators to power these workloads. Recording ML inference results on the ledger only provides value to the end-user when combined with the CPU-based transactions that caused the ML inference request to be executed along with any subsequent transactions. In addition, the inability to accelerate these workloads on a permissioned ledger further raises organizational friction in deploying a permissioned ledger.

1.5 Research motivation

The introduction of permissioned ledger frameworks has simplified building ledger-based applications. However, there are still limitations that must be addressed. This thesis addresses the real-world concerns that limit the deployment of permissioned ledgers (see section 1.4). Two of these shortcomings are:

- **Accountability.** The research community commonly accepts that it is impossible to create a system where tampering with the ledger can be detected in all situations. The following is commonly accepted as fact: "for all secure BFT protocols designed for $2t+1$ replicas communicating over a synchronous network, forensic support is inherently non-existent; this impossibility result holds for all BFT protocols and even if one has access to the states of all replicas (including Byzantine ones)" [218]. This assumption implies it is impossible to audit a ledger and be guaranteed that all actions that deviate from the correct execution of the protocol are discovered.
- **Performance.** The work of introducing new consensus protocols is partly to improve the throughput and latency of permissioned ledgers. Currently, the consensus protocols of permissioned ledgers rely on transactions being executed sequentially. Sequential execution of transactions is a bottleneck like executing a single transaction takes a substantial

amount of time. In addition, sequential execution can severely under-utilize expensive and scarce computing resources.

1.6 Contributions

This thesis demonstrates that it is possible to build a permissioned ledger that can detect dishonest behaviour and assign blame to the guilty party while efficiently executing computationally expensive workloads, including ML workloads, that utilize the parallelism of both CPUs and GPUs. The combination of these properties shows a permissioned ledger that can meet the needs of an organization that would like to deploy a permissioned ledger. Specifically, the thesis makes the following contributions in the context of three systems:

1.6.1 Accountability

We address accountability in our system, **IA-CCF**. IA-CCF is a permissioned ledger system that assigns blame to misbehaving replicas (and the members who control them), even if all replicas misbehave. It achieves this while permitting changes to the consortium membership and the replica set. IA-CCF's accountability guarantees thus improve the trustworthiness of a permissioned ledger by creating a strong disincentive for misbehaviour.

IA-CCF supports accountability by introducing Ledger PBFT (L-PBFT), a new BFT state machine replication protocol that stores ordered transactions in the ledger together with protocol messages from replicas, using the protocol messages to justify the chosen order. L-PBFT maintains Merkle trees [162] over the ledger and includes the root of the Merkle tree in protocol messages. Since protocol messages are signed by the replicas, this commits them to the entire contents of the ledger. IA-CCF issues receipts to clients that provide succinct, universally-verifiable evidence that a transaction was executed at a given position in the ledger. Receipts include signed protocol messages from multiple replicas that executed the transaction, thus binding them to a prefix of the ledger. If clients obtain a sequence of receipts that violate linearizability, anyone can audit the ledger and receipts to assign blame to at least $N/3$ replicas. Auditing produces an irrefutable universal proof-of-misbehaviour (uPoM) in the form of contradictory statements signed by the same replica. The uPoM can be used by an enforcer, e.g., a court, to punish the members responsible for the misbehaving replicas. Since IA-CCF provides accountability even if all replicas misbehave, the enforcer may have to compel members to produce a ledger, imposing sanctions otherwise. While this introduces a weak synchrony assumption, the enforcer chooses a conservative timeout to make blaming correct members unlikely.

1.6.2 Performance

We present our solution to performant Byzantine consensus based ledger in two systems:

We address performant execution of workloads that utilize hardware accelerators, and record their results on a ledger while utilizing a Byzantine consensus protocol in our system, **DropBear**. DropBear is a cloud-based ML inference service that offers trustworthy inference decisions over DNN models. DropBear allows model owners to upload ensembles of models that consist of one or more DNN models, and clients then submit inference requests over these ensembles. DropBear introduces an execute-agree-attest strategy that separates the expensive execution of inference requests from the agreement and attestation of inference results. Inference requests are first executed in batches by geo-distributed cloud replicas against an ensemble of models. After execution, a primary replica batches the inference requests again orders them with respect to model updates and sends batches to a set of backup replicas. The replicas agree with one another's inference result subject to the bounds associated with the ensemble. After 2/3 of the replicas have agreed, they attest that the result is within the prescribed bounds. Through this separation, DropBear can batch inference requests and consensus operations independently, allowing for a higher degree of parallelism across replicas: batching inference requests for heavily referenced models [131] into execution batches increases GPU utilization, and execution batches can span multiple consensus batches; consensus batches are constructed to better utilize the wide-area network between cloud replicas and reduce the per-batch CPU overhead of cryptographic operations.

We continue our focus on performant execution of permissioned ledger workload but on modern CPUs. We present our system, **Bunyip**, an extension to the IA-CCF auditable permissioned ledger's consensus protocol L-PBFT to allow replicas to execute transactions in parallel. Bunyip gives the primary replica the responsibility to collect transaction execution dependency information and then pass the collected information to the backup replicas. Backup replicas in Bunyip utilize the transaction dependency information to execute transactions in parallel and at the same time ensure that the dependency information passed to them is correct. Through parallel execution, Bunyip can execute a large number of independent transactions in parallel and significantly improve the throughput of transaction execution when executing more complex transactions.

1.7 Publications and patents

The contents of this thesis are influenced by the following publications:

- **IA-CCF: Individual Accountability for Permissioned Ledgers**
  Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward

Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich
19th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2022 Renton, WA, USA [214]

- **Dropbear: Machine Learning Marketplaces made Trustworthy with Byzantine Model Agreement**
  Alex Shamis, Peter Pietzuch, Antoine Delignat-Lavaud, Andrew Paverd, and Manuel Costa
  arXiv preprint, arXiv:2205.15757, May 2022 [215]

- **CCF: A framework for building confidential verifiable replicated services**
  Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, Christoph M Wintersteiger
  Technical Report, MSR-TR-2019-16, April 2019 [206]

The contents of this thesis is influenced by the following patents:

- **Receipts in a Distributed Ledger**
  Eddy Ashton, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Cedric Fournet, Julien Maffre, Peter Pietzuch, and Alex Shamis
  Currently under review

The contents of this thesis have been indirectly influenced by the following publications:

- **AMP: Authentication of Media via Provenance**
  Paul England, Henrique S. Malvar, Eric Horvitz, Jack W. Stokes, Cédric Fournet, Rebecca Burke-Aguero, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, John Deutscher, Shabnam Erfani, Matt Gaylor, Andrew Jenks, Kevin Kane, Elissa M. Redmiles, Alex Shamis, Isha Sharma, Sam Wenker, and Anika Zaman
  12th ACM Multimedia Systems Conference (MMSys), 2021 Istanbul, Turkey [85]

- **Multi-stakeholder media provenance management to counter synthetic media risks in news publishing**
  Jatin Aythora, Rebecca Burke-Agüero, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, John Deutscher, Nigel Earnshaw, Laura Ellis, Paul England, Cedric Fournet, Matt Gaylor, Charlie Halford, Eric Horvitz, Andrew Jenks, Kevin Kane, Marc Lavallee, Scott

Lowenstein, Brian MacCormack, Henrqiue S. Malvar, Sinead O'Brien, J Parnall, Elissa M. Redmiles, Alex Shamis, Isha Sharma, Jack Stokes, Sam Wenker, and Anika Zaman International Broadcasting Convention (IBC), 2020 Amsterdam, Netherlands [16]

1.8 Outline

The remainder of this thesis is structured as follows:

Chapter 2 describes the background material on which the concepts in this thesis are built. We first provide the required description of Byzantine consensus protocols followed by techniques used to audit ledgers. Next, we explain techniques and strategies that have been used to improve the throughput of permissioned ledgers, specifically those that utilize Byzantine consensus protocols.

Chapter 3 explores the role of accountability in permissioned ledgers and their consensus protocols. In this chapter, we argue that setting an arbitrarily limit – after which a permissioned ledger's ledger can be rewritten in a non-detectable manner – results in a user never being able to trust the permissioned ledger. We then describe IA-CCF, a system that co-designs its Byzantine consensus protocol with the ledger and transaction execution to create a permissioned ledger that allows clients to assign fine-grain blame to dishonest replicas. We show how an audit of an IA-CCF permissioned ledger always undercovers malicious actions, even if the ledger is rewritten, thus allowing clients to trust the ledger.

Chapter 4 explores how permissioned ledgers can fully utilize modern hardware through co-design of the hardware resource being utilized and the consensus protocol. In this chapter, we present the permissioned ledger DropBear that, is a cloud-based ML inference service that offers auditable inference decisions using machine learning models. DropBear separates the execution of inference requests on a GPU for ordering the requests in the consensus protocol to obtain high utilization of the GPU.

Chapter 5 explores how a Byzantine consensus protocol in a permissioned ledger can fully utilize modern multi-core CPUs when executing transactions. In this chapter, we present Bunyip, an extension of the IA-CCF permissioned ledger that utilises all of the replica's CPU cores when executing transactions on both the primary and backup replicas. Bunyip captures and then validates the dependency information between transactions. This ensures that a Bunyip replica's CPU cores are fully utilized without compromising the permissioned ledger's linearizability property.

Chapter 6 summarises the thesis and discusses directions for future work.

Chapter 2

Background

This chapter provides background into the concepts and materials explored in this thesis. The chapter is divided into three sections. In the first section, we describe background information to understand Byzantine consensus protocols and how they relate to permissioned ledgers. In the next section, we discuss accountability in distributed systems. We first look at some seminal work and discuss current proposals for adding accountability to permissioned ledgers. In the final section, we discuss techniques and methods used to improve the throughput of Byzantine consensus protocols, focusing on techniques that improve the throughput of transaction request execution.

In addition, we present chapter-specific background material in each of the technical chapters.

2.1 Byzantine consensus protocols

There are several types of consensus protocols used by blockchains and permissioned ledgers. We give a brief overview of different consensus protocols used in blockchains in Chapter 1. In this section, we focus on the consensus protocols used in permissioned ledgers.

2.1.1 Practical byzantine fault tolerance

Lamport, Shostak, and Pease published "The Byzantine Generals Problem" [148], which presented a challenge to the computer science community. The challenge was to design an algorithm for a distributed system where replicas within the system may be actively malicious. The challenge's goal is to design a distributed system which can make progress (correctly commit and serialize transactions) when a subset of the replicas are actively malicious. The system must also define the minimum proportion of replicas which are honest versus those that are malicious.

The first practical solution to the "Byzantine Generals Problem" was proposed by Miguel Castro and Barbara Liskov in their seminal work *Practical Byzantine Fault Tolerance* (PBFT)". They describe an algorithm that solves the "Byzantine Generals Problem" and define the maximum proportion of replicas that can be malicious in their system and prove the correctness of their approach. They implement a system to study the performance impact of building a Byzantine fault tolerant system [49].

The algorithm can be divided into two parts: the first is the failure free execution path where there are $3f + 1$ replicas where f or fewer of these replicas may be performing arbitrary malicious actions. The algorithm works in several stages: in the first stage (*request* stage), a client sends a request to all $3f + 1$ replicas. In the next stage (*pre-prepare* stage) the primary sends a pre-prepare message to all the other replicas, where the message proposes an ordering for a set of requests. Next, all the replicas, except the primary, send a prepare message to all the other replicas in the *prepare* stage. This confirms to all replicas that they agree on which request(s) to order and how to order them. At this point, an honest replica knows that at least $2f + 1$ replicas agree on the request execution order; if not they abort processing the requests. Finally, in the *commit* stage, all honest replicas send a commit message to all other replicas to state that they are aware of at least $2f + 1$ replicas that agree on the ordering information sent in a *pre-prepare* message. After receiving $2f$ commit messages and sending its own a replica considers the requests to be part of the total order, which cannot be reverted, and executes the requests. This protocol is illustrated in Figure 2.1.

The second part of the PBFT algorithm deals with how an incorrect primary is replaced. A primary can be considered incorrect for a variety of reasons such as sending inconsistent requests during the *pre-prepare* stage, not sending messages, or other reasons described by Castro and Liskov [49]. The protocol works by rotating deterministically the primary through all the replicas. It begins when at least a group of $2f + 1$ replicas independently decide that the primary needs to change, and send *view-change* messages signalling this intention. This step is shown in Figure 2.2a.

When the prospective new primary receives evidence (*view-change* messages) that at least $2f + 1$ replicas are requesting a new primary, the prospective primary sends a message indi-

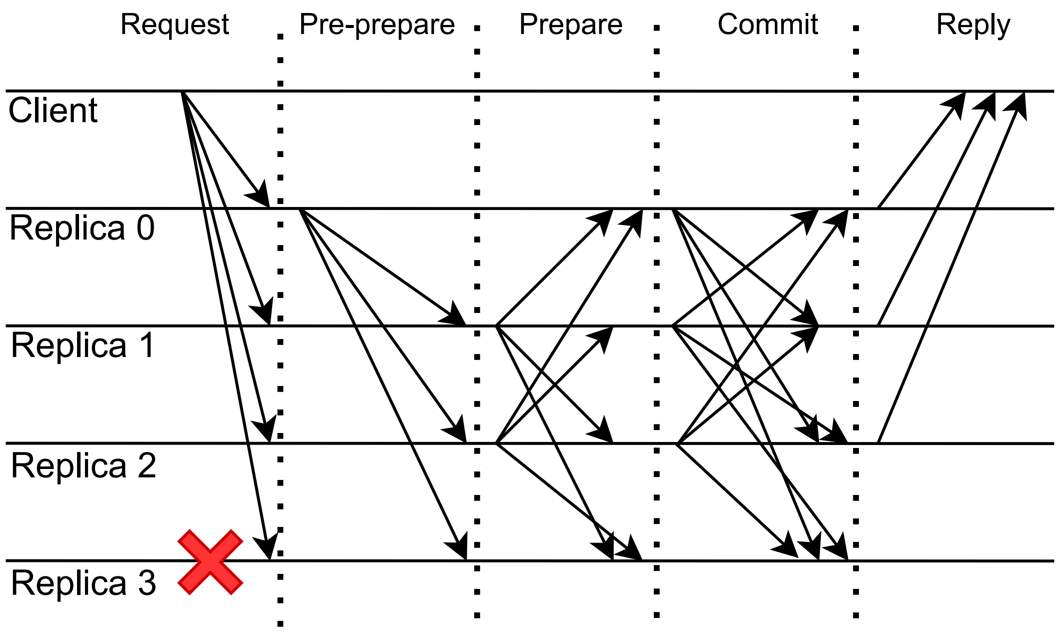
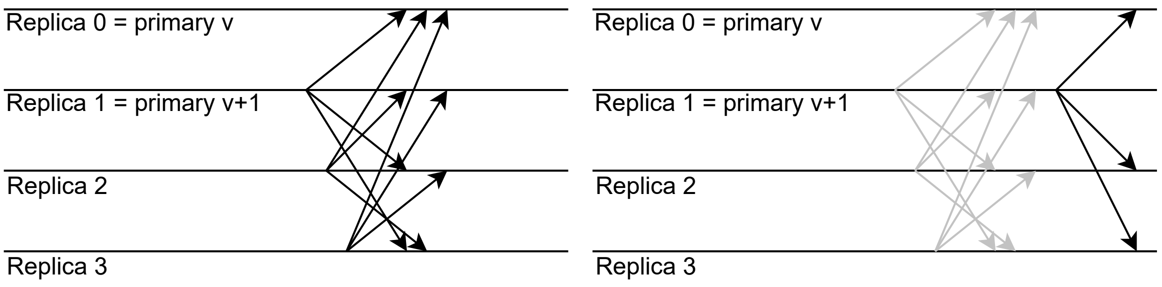


Figure 2.1: Practical Byzantine Fault Tolerance [49].



(a) PBFT: Collect view change evidence. (b) PBFT: Send new view message.

Figure 2.2: Practical Byzantine Fault Tolerance view change protocol [47].

cating that it is the new primary (*new-view* message) with evidence (*view-change* message), that at least $2f + 1$ replicas are requesting a new primary. If the other replicas are able to validate said evidence, they consider the prospective primary to be the actual primary. This is shown in Figure 2.2b.

PBFT and derived approaches are currently the most commonly used consensus protocols for permissioned ledgers.

2.1.2 Byzantine fault tolerance for permissioned ledgers

There have been a large number of Byzantine fault tolerant protocols proposed since PBFT was published in 1999. In this section, we provide an overview of Byzantine fault tolerant protocols, specifically focusing on BFT protocols used by permissioned ledgers.

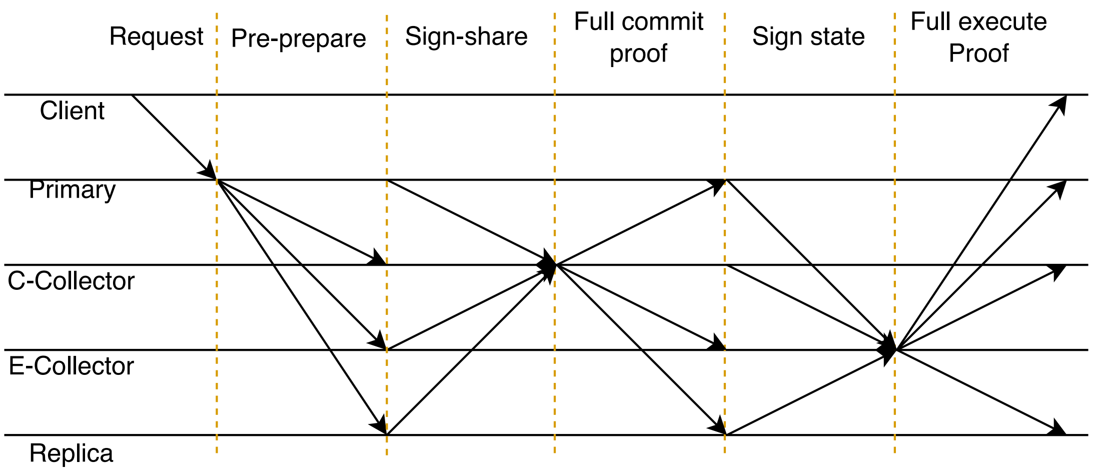


Figure 2.3: SBFT Schematic message flow for n=4, f=1, collectors (c)=0 [106]

.

SBFT is a Byzantine consensus protocol that is designed to work as the consensus protocol for a permissioned ledger that must scale to hundreds of replicas [106]. The protocol uses several techniques to achieve its performance and scalability goals:

- SBFT defines a replica type called a *collector*. When a non-collector replica receives a message as part of the normal progress of the protocol, instead of sending a message to all the replicas, it sends a single message to the collector replica. The collector replica then aggregates the responses, waits until it has $2f + 1$ matching messages, and then forwards them to the non-collector replicas. This is done to remove a large number of network messages from the consensus protocol (removing the all-to-all messaging pattern from PBFT). However, removing the all-to-all messaging pattern results in the requirement that SBFT signs more messages than PBFT using MAC based authenticators.

- SBFT addresses the performance impact of the additional asymmetric cryptography by using threshold signatures [220]. SBFT collectors wait until they gather $2f + 1$ relevant messages and then create a threshold signature which they then send to the other replicas. This results in the non-collector replicas having to verify only a single threshold signature versus $2f + 1$ signatures if the collector replicas did not use threshold cryptography.

Figure 2.3 shows the message flow of SBFT when there are no failures.

The authors report that SBFT achieves more than 2 times higher throughput and 1.5 times lower latency in their WAN experiments than PBFT. However, the creation of threshold signatures is computationally expensive, and it is not obvious if threshold cryptography improves the overall throughput of the system. It is only beneficial if the number of replicas in the system is large.

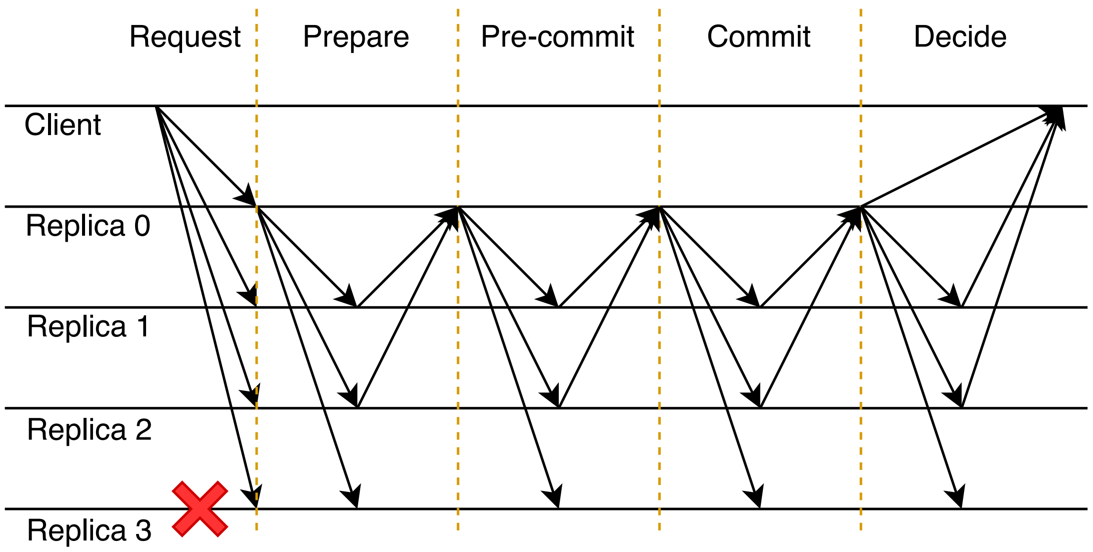


Figure 2.4: HotStuff Schematic message flow for n=4 or f=1

HotStuff is a consensus protocol that is designed to work in permissioned ledger systems and, similar to SBFT, it is designed to scale out to a large number of replicas [254]. HotStuff removes the all-to-all messaging pattern of PBFT and utilises threshold cryptography to reduce the cryptographic cost of signature verification. HotStuff's key contributions are around fast changes to the primary.

In HotStuff, every consensus round starts with a possible view-change "any correct leader, once designated, sends only O(n) authenticators to drive a consensus decision. This includes the case where a leader is replaced. Consequently, communication costs to reach consensus after [global stabilization time] is $O(n^2)$ authenticators in the worst case of cascading leader failures". Figure 2.4 shows the message flow of HotStuff when there are no failures.

HotStuff is shown to have high throughput in a LAN scenario, however, several orders of magnitude lower throughput in a WAN scenario (see Section 3.6.1). HotStuff's WAN throughput issue originates in the blockchains pipelining design. Solutions to this issue have been proposed [262] and experimental results are shown to resolve the WAN throughput issue. The authors of HotStuff have announced a follow-up protocol, DiemBFT, but, at the time of writing, the results for the follow-up work have not been published.

IBFT is the Istanbul Byzantine fault tolerant consensus protocol, a version of which is used by Consensys Quorum and Hyperledger Besu [210, 175]. The protocol is heavily inspired by PBFT and has a nearly identical failure free path and a simplified view-change protocol. The primary change implemented by IBFT is its ability to change the set of replicas used by the consensus protocol. The selection of a new primary is done randomly, unlike PBFT, which selects deterministically in a round-robin fashion.

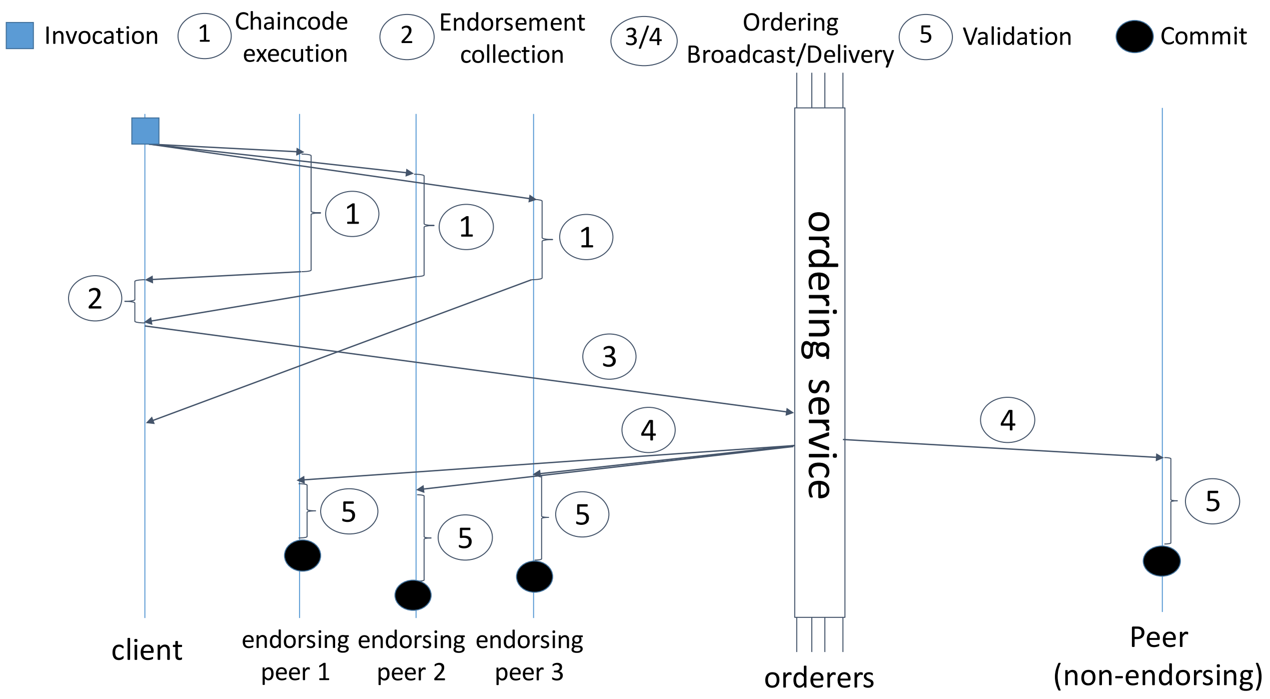


Figure 2.5: Hyperledger Fabric transaction flow. Copied from [11]

The design changes of IBFT have created controversy with claims that the protocol is unsafe [209], resulting in Hyperledger Besu using IBFT version 2, unlike Consensys Quorum that uses the original protocol. However, as neither of the versions of IBFT has been published in a peer reviewed venues, it is difficult to draw a conclusion on the safety or correctness of either version of the IBFT consensus protocol. While, publication does not guarantee correctness, publication in a reputable journal provides assurance that subject matter experts examined the protocol and believe it is complete and correct.

2.1.3 Alternative consensus protocols for permissioned ledgers

There are several permissioned ledgers that do not utilize Byzantine fault tolerant consensus protocols and instead rely on a crash fault tolerant consensus protocol for consensus but harden the permissioned ledger in other ways. Commonly, these crash fault tolerant consensus protocols are hardened through the introduction of trusted execution environments [206, 133, 29].

Hyperledger Fabric [11] introduces the *execute-order-validate* architecture for permissioned ledgers. The architecture is shown in Figure 2.5 and separates the execution and commitment of a transaction into four phases.

1. **Execute.** A client begins its interaction with Hyperledger Fabric by creating a transaction and sending it to a subset of the replicas. The replicas execute the transaction,

produce a read and write set, and send the read/write set, along with a signed endorsement, to the client.

2. **Order.** After the client has received the required number of endorsements, the transaction, along with the read/write set and endorsements, are ready for ordering. The ordering phase creates a total order of transactions that have been executed and endorsed. The total order can be created by replicas running any consensus protocol, however, at the time of writing, the latest release of Hyperledger Fabric only implements a crash-fault replication protocol (Raft) [185, 186]. The use of Raft results in questionable security properties for Hyperledger Fabric, and this is acknowledged by Hyperledger Fabric's official documentation: "Raft is the first step toward Fabric's development of a byzantine fault tolerant (BFT)" [88]. At the time of writing, the creation of a version of Hyperledger that uses PBFT is under development.

3. **Validate.** After a transaction is ordered, it is validated. The validation phase ensures that any transaction inserted into the total order has a valid read and write set produced during the execute phase and that there are no conflicts, which would invalidate the earlier execution of the transaction. In addition, the endorsements are re-checked to ensure that they are produced by replicas that are still active.

4. **Update State.** In the final phase, the state of Hyperledger Fabric's key-value store is updated with transactions that have been successfully validated.

The separation of execution and ordering allows Hyperledger Fabric to scale independently the number of replicas used for any of the previously mentioned functions. However, this independent scaling comes with a performance penalty (see Section 3.6.1).

Confidentual Consortum Framework (CCF) [206] is a permissioned ledger framework that provides similar functionality as Hyperledger Fabric. It utilises the Raft consensus protocol to obtain a total order over executed transaction requests, however, CCF protects its replicas against attackers by moving the execution of each replica into a trusted execution environment (TEE). A TEE is a hardened section of the CPU that provides integrity and privacy over applications it executes. CCF uses Intel SGX [206] TEE. CCF's reliance on general-purpose TEEs means that the service's security is completely undermined if the TEE is compromised.

In this thesis, we assert that TEEs, which allow general purpose computation, are not a practical solution for building permissioned ledgers. In the event that a vulnerability in the TEE is discovered [238, 245], all work that was performed within the time window in which the vulnerability existed cannot be trusted.

2.1.4 Conclusions

In this section, we reviewed the consensus protocols commonly used in permissioned ledgers. The majority of these protocols have evolved from PBFT and thus set a threshold on the number of replicas that can be compromised before the permissioned ledger is controlled by an adversary. While this threshold (f) is scientifically justified within all consensus protocols, for a permissioned ledger, it is an arbitrary threshold with no justification why only f replicas can be compromised and not $f+1$. We assert that any practical permissioned ledger requires a mechanism that provides guarantees when any number, or all, of the replicas have been compromised.

2.2 Accountability

In the previous section, we explored the consensus protocols used in permissioned ledgers and their fundamental inability to ensure a permissioned ledger's safety. We now look at alternative safety guarantees that a permissioned ledger could provide.

We explore accountability as it allows us to build on one of the distinguishing features of permissioned ledgers. All entities that are part of a permissioned ledger are tied to their real world identity. Thus, if malicious behaviour could be detected, even if it cannot be prevented, the malicious entity can be made accountable and face consequences. In this thesis, we conjecture that if a malicious actor is accountable for their actions, they will not act maliciously if the consequences are of an appropriate severity.

2.2.1 PeerReview: practical accountability for distributed systems

The popularization of distributed systems brought with it concerns that replicas in a system could be running on machines that are controlled by a malicious actor. These concerns in part are addressed by the work on Byzantine fault tolerance, however, Byzantine fault tolerance is unable to provide any guarantees if more than f replicas are malicious (see Section 2.1.1). This gap was initially addressed by Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel in their work "PeerReview: Practical Accountability for Distributed Systems" [111].

The goal of PeerReview is to blame replicas that act maliciously either by violating correctness or liveness. PeerReview is a distributed system in which each replica runs a deterministic state machine. PeerReview validates if a replica acted correctly by re-executing the state machine with known good inputs and checking that the expected output is produced. Each replica maintains a tamper-evident ledger containing all the network messages that it received and sent and the commitment (signature using a private key over the ledger) of said replica to the sent and received messages. Each replica includes a hash of the ledger in each message. In addition, the replica signs the message, that ties the replica's ledger to every previously sent message.

PeerReview checks for liveness by requiring that all replicas send a signed response message acknowledging any message that they received. The response message is also added to each replica's ledger. PeerReview then introduces the role of a *witness*. Each witness is associated with a number of replicas and is responsible for validating the correctness and liveness of every replica by occasionally replaying the replica's ledger.

The key drawback of the PeerReview method of accountability is the cryptographic overhead of signing and verifying messages. To alleviate some of the performance impact of the cryptography required to show that a replica behaved correctly or demonstrated dishonest behaviour, Haeberlen et al. introduced a technique for creating a snapshot of a replica's state. This snapshotting technique can be used to verify that the replica acts correctly without having to replay the entire ledger in a limited set of scenarios [110]. However, as we show later in this thesis, applying these techniques to a permissioned ledger results in the transaction throughput and latency of the permissioned ledger being severally degraded (see Section 3.6.1).

2.2.2 Accountability in permissioned ledgers

Designing a distributed ledger that is auditable after its consensus protocol's threshold failure limit has been exceeded is a new area of focus for the research community. The straw-man approach has been for systems to produce transaction execution receipts that are signed by all replicas and send them to clients as proof that a transaction executed [5]. This approach is not practical for the following reasons: It adds additional asymmetric cryptography to the protocol which results in a significant performance degradation; second, it does not address the real world problem of replicas being added and removed from a running system (reconfiguration).

In the remainder of this section, we look at the work that considers accountability in permissioned ledgers. Note that the described work was done in parallel to the work presented in this thesis.

BFT Protocol Forensics is a study of the auditability (forensics) of multiple Byzantine fault tolerant consensus protocols. The purpose of the study is to identify which consensus protocols can be extended so that, if replicas retain their consensus protocol messages, or other trivial changes are made, *some* auditability properties exist. The authors define a ledger as auditable if a third party can review the ledger and correctly determine which replicas performed a dishonest action.

Their results are presented in Table 2.1. Unfortunately, BFT Protocol Forensics does not consider auditability when changing the active replica set nor does it find any BFT protocols that provide forensic properties if $2f + 1$ or more replicas are malicious.

Polygraph proposes a new "accountable Byzantine consensus algorithm" which is designed to detect if there are more than $N/3$ malicious replicas [55]. The Polygraph consensus protocol

Symbol	Interpretation
f	maximum number of faults for obtaining agreement and termination
k	the number of different honest replicas' transcripts needed to guarantee a proof of culpability
d	the number of Byzantine replicas that can be held culpable in case of an agreement violation

Protocols	**Forensic Support**	**Parameters**		
		m	k	d
PBFT-PK HotStuff VABA [3]	Strong	$2f$	1	$f+1$
PBFT-MAC Algorand	None	$f+1$	$2f$	0

Table 2.1: Summary of results: the forensic support values of d are the largest possible and $n = 3f + 1$. Copied from [217]

is a multi-phase protocol where:

1. A replica sends what it believes to be the current state of the system to all other replicas.
2. Replicas then wait to receive at least one message from another replica or a timeout to occur.
3. If a replica receives the state from the current primary replica, it adopts said state.
4. Finally, a replica checks if it received $N-f-1$ messages that agree on a state and, if so this state is adopted, otherwise the replica repeats the above process but rotates its understanding regarding which replica is the primary.

Polygraph, similar to consensus protocols analyzed in BFT Protocol Forensics, does not consider reconfiguration and, as such, is not a practical real world solution nor can it detect if there are more than $2f$ malicious replicas.

ZLB builds on top of Polygraph to define a blockchain and a consensus protocol that can, in a specific set of scenarios, replace malicious replicas [201]. This system, however, only allows for the trading of cryptocurrency, which is done to simplify the detection and rollback of malicious transactions.

The Byzantine consensus protocol is given the responsibility to detect if a replica behaved in a dishonest manner, however, rather than reporting said replica the protocol automatically attempts to replace it. ZLB does support reconfiguration and is therefore the most practical of the previously mentioned solutions, but it can only reconfigure correctly if there are at most $2f$ dishonest replicas. This makes it vulnerable to a patient adversary, which could wait until it has compromised at least $2f + 1$ replicas before performing any malicious actions.

2.2.3 Conclusions

The promise of accountability stems from the expectation that dishonest entities will be deterred from acting dishonestly if they are guaranteed to be caught and face the consequences of their actions. Unfortunately, accountability-based systems that could be used in permissioned ledgers have shortcomings, making them impractical. PeerReview and its derived work degrades the performance of permissioned ledgers to such an extreme that this technique is impractical. Other accountability-based protocols which have been specialized for permissioned ledgers, can be compromised in such a way that a dishonest actor can perform undetectable malicious actions and avoid consequences.

2.3 Scaling Byzantine consensus protocol performance

In this section, we move our focus to another issue that limits the practical usage of permissioned ledgers: their transaction execution throughput and latency. The throughput and latency of transaction request execution of permissioned ledgers is primarily governed by a permissioned ledger's consensus protocol. This concern has been addressed by a large number of systems and by the research that backs them. In this section, we look at how previous work improves transaction throughput and latency of permissioned ledgers.

We explore several families of techniques that have been employed to improve transaction throughput and latency. It is important to note that our focus is on improving transaction *execution* throughput and latency rather than increasing the number of transactions the system can order. In that, we do not look at systems such as Pompē [262], Narwhal and Tusk [67], or leaderless BFT consensus protocols [83, 63], which improve the rate at which transactions are ordered but do not contribute to improving transaction execution.

2.3.1 Byzantine fault tolerant sharding

In parallel to designing blockchains and permissioned ledgers that use Byzantine fault tolerance as their consensus protocol the research community is exploring sharding blockchains and per-

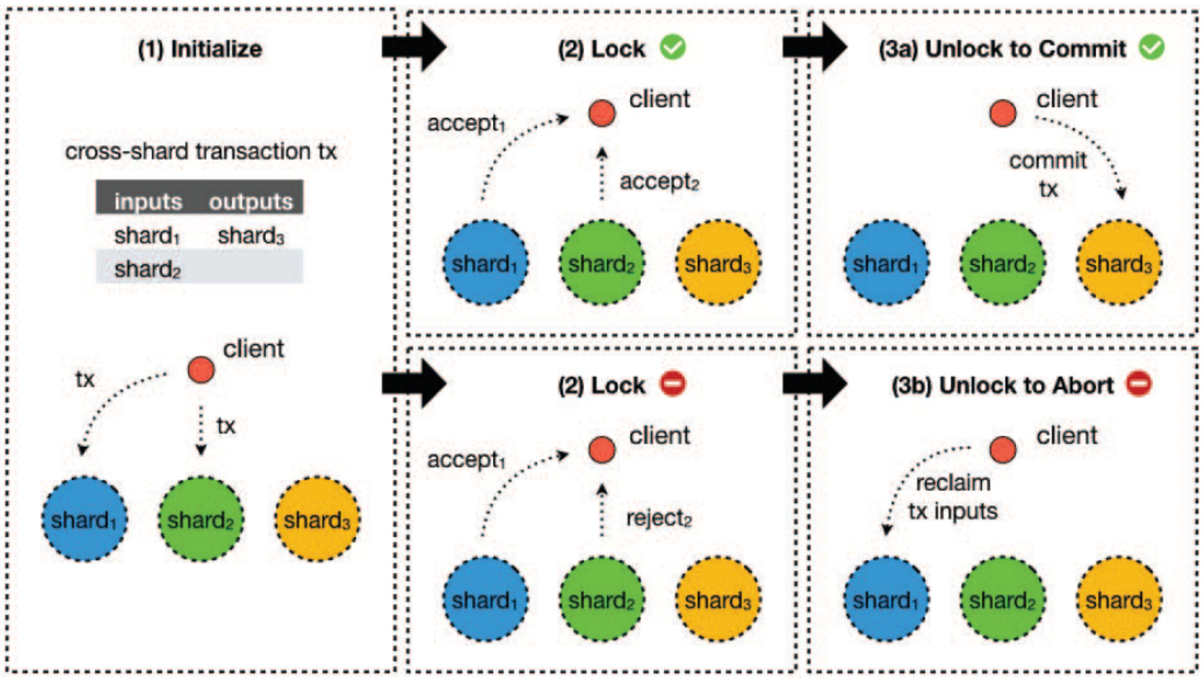


Figure 2.6: OmniLedger Commit Protocol. Copied from [140]

missioned ledgers, using a Byzantine fault tolerant consensus protocol to combine the shards, thus maintaining a consistent view over the entire system.

A Secure Sharding Protocol For Open Blockchains by Saxena et al. introduces ELASTICO, "the first candidate for a secure sharding protocol with presence of Byzantine adversaries" [158]. The authors assign replicas to committees where each committee is responsible for a disjoint part of the blockchain. Each committee executes transactions in their part of the blockchain and, when all the committees finish executing the transactions assigned to them, a final committee combines the results. As the first sharded blockchain, ELASTICO introduces several commonly used techniques including: employing epochs for executing transactions; how transactions are assigned to committees; and requiring potential committee members to complete a proof-of-work puzzle before they are allowed into the committee.

ELASTICO has several flaws in its design that have been recognized by later work including that the identity of the committee members is known before they join, allowing an adversary more time to launch an attack. Follow-up work addresses this issue by using verifiable random functions [165] for committee election, in a similar way to Algorand (see Section 1.2.2).

Chainspace [6] removes the need to have a single committee, which ultimately manages a transaction after it is executed by multiple shards. Figure 2.7 is a diagram representing Chainspace's architecture.

The primary contribution of Chainspace is a "distributed atomic commit protocol" that

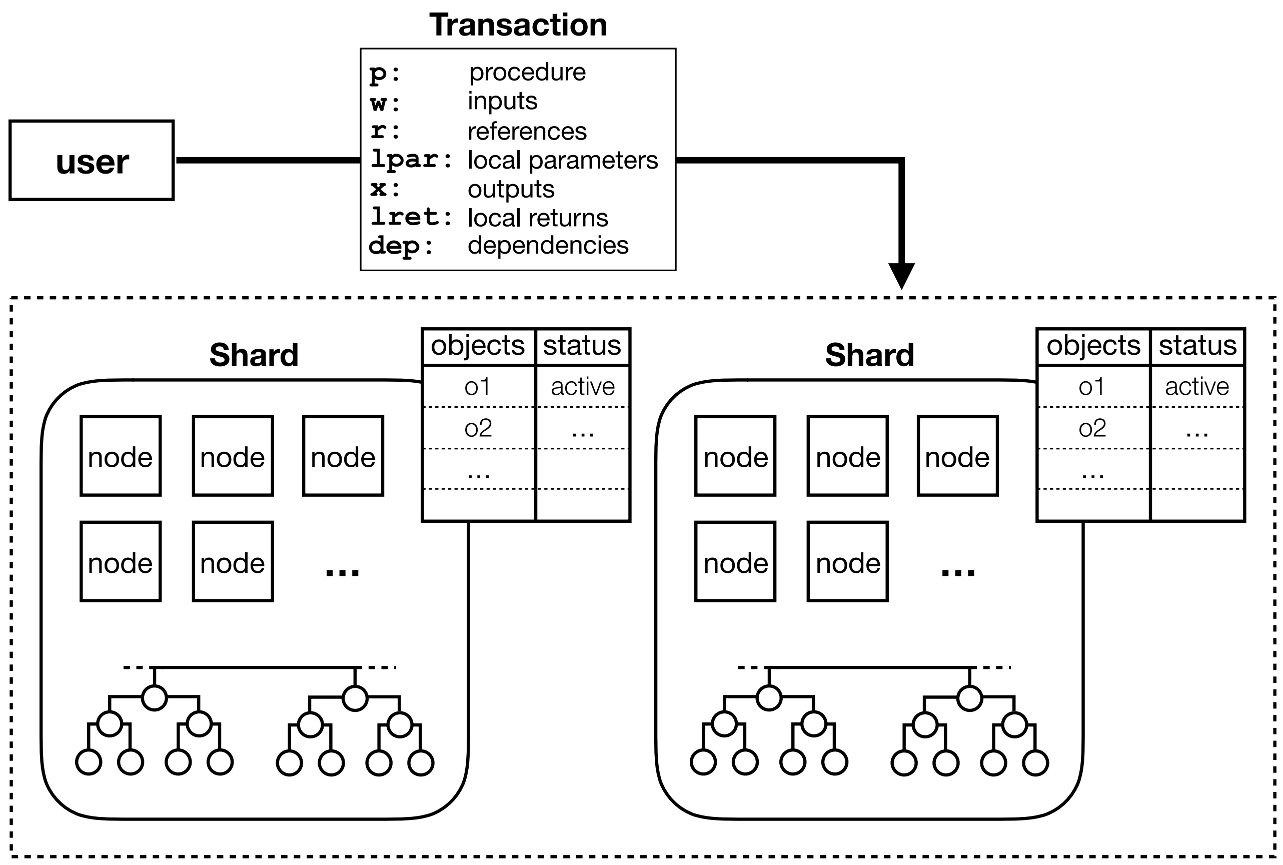


Figure 2.7: Chainspace Architecture Overview. Copied from [6]

"combines two primitive protocols: *Byzantine Agreement* and *atomic commit*". A visual example of the protocol can be seen in Figure 2.8. It involves the following steps:

1. The client produces a transaction and divides it into parts which each shard can execute independently.

2. Each shard executes the sub-transaction that it was assigned. As part of execution, Byzantine agreement is run between the nodes in a shard.

3. Nodes then send cross shard messages propagating their results along with proof that Byzantine agreement ran and agreement was reached.

4. Once a shard receives enough proof that the other shards have successfully or unsuccessfully reached agreement, the shard in question performs another round of Byzantine agreement over these results and either commits or abandons the state reached in Item 2.

5. Shards then send the result of their execution to the client and to other shards that need to create new objects as a result of transaction execution.

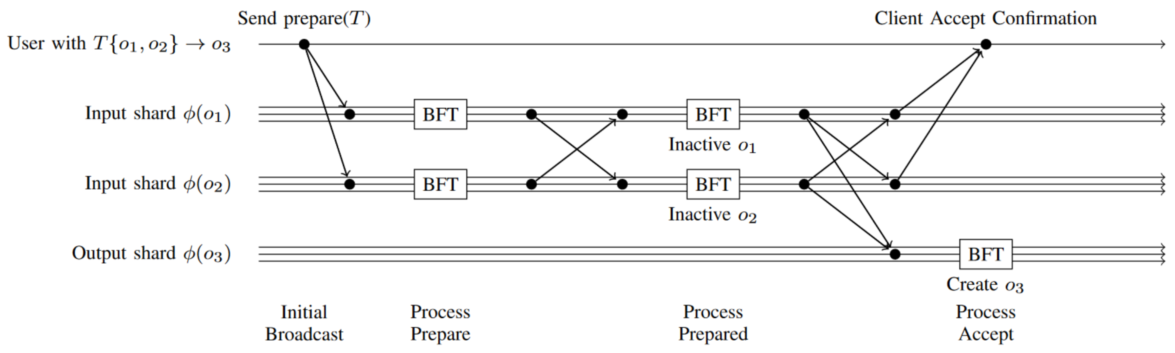


Figure 2.8: Chainspace Commit Protocol. Copied from [6]

6. Finally, another round of Byzantine agreement is run to create the new objects described in Item 5.

The performance of Chainspace is unfortunately lackluster: authors report transaction throughput in the hundreds of transactions per second and the transaction latency in the tens of seconds. However, they show results that exhibit near linear scaling in transaction throughput when adding more shards. The authors conclude that their performance is limited by the performance of their Byzantine fault tolerance consensus protocol implementation.

OmniLedger [140] utilises a commit protocol that requires clients to act as the coordinator for their own transactions. The protocol involves the client sending a sub-transaction to one or more shards and receiving cryptographically verifiable proof of the result of the transactions' execution from each shard. This proof includes evidence of which objects were locked as part of the sub-transaction. After a client obtains said proof, it sends it along with a sub-transaction to other shards, where the sub-transaction depends on the locked objects as input. Finally, after completing the second set of sub-transactions a client receives proof of their execution, which is used as proof to unlock the locked objects and mutate them as necessary. The protocol is illustrated in Figure 2.6.

The authors' argument to justify using an untrusted client as the transaction coordinator is that the client can only lock objects that they own. If they never unlock an object, the client only acts in its own disinterest. This, unfortunately, means that transactions in the system are limited to transferring crypto-currency between accounts.

2.3.2 Byzantine fault tolerance and database sharding

An alternative to designing a new transaction protocol for sharded permissioned ledgers is to reuse techniques that solve similar problems in adjacent fields, relational-database sharding.

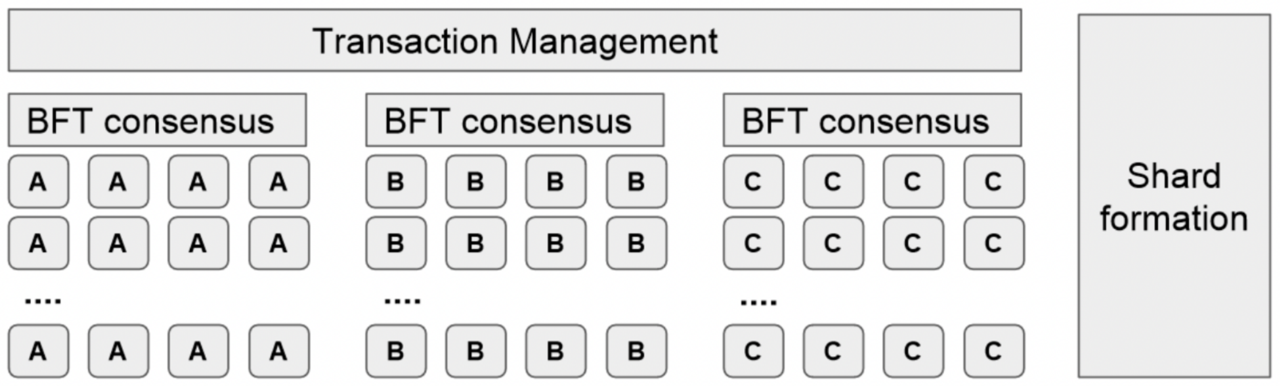


Figure 2.9: Towards Scaling Blockchain Systems via Sharding. Copied from [68]

This approach is explored by Dang et al. in their work “Towards Scaling Blockchain Systems via Sharding” [68]. Figure 2.9 shows an overview of the system. Data is stored in shards, which maintain data consistency and integrity via the PBFT consensus protocol. In addition, a collection of transaction coordinator machines (Reference Committee) that also run a BFT protocol are responsible for coordinating both the two phase commit and the two phase locking protocols, both of which are an integral part of the transaction protocol.

At a high level the transaction protocol is divided into the following parts:

1. A client sends a *Begin Transaction* request to the *Reference Committee.*
2. The *Reference Committee* executes the transactions and sends the sub-transactions to the appropriate shards.
3. Next, the *Reference Committee* waits for the appropriate proofs of execution, created after the shards execute the sub-transaction. The proofs contain either a commit or abort decision.
4. Finally, the *Reference Committee* either performs the commit or abort operation on the global state of the system.

Dang et al. divide their work into three distinct parts: “(i) optimizations that improve the performance of the consensus protocol running within each individual shard, (ii) an efficient shard formation protocol, and (iii) a secure distributed transaction protocol that handles cross-shard, distributed transactions.”. However, there are several areas of concern:

1. The optimizations to the consensus protocol involve reducing the number of messages that PBFT must send to reach agreement. This is implemented by utilizing TEEs and builds on the work by Behl et al. “Hybrids on Steroids: SGX-Based High Performance BFT” [30].

Relying on TEEs for safety is concerning, as even a single TEE being compromised would result in the system losing its safety properties.

2. The system moves nodes between shards to ensure that an attacker is unable to pool its resources and compromise a single shard. The claim that this technique is effective is in itself suspect as a patient attacker can compromise a replica and wait for it to be moved into a target shard. This is a well known attack vector, and multiple blockchains, e.g., Algorand, optimize their design to avoid this issue.

The performance of this and other sharded systems does not meet expectations and the reported throughput is lower than comparable systems that are not sharded, e.g., Hyperledger Fabric [101]. While this result is unexpected, it is not possible to know if this is a problem with the design or implementation.

2.3.3 Transaction execution in Byzantine fault tolerant systems

Practical Byzantine fault tolerance and the majority of related BFT protocols work by agreeing on an order of execution and then sequentially executing the ordered transaction requests. This may result in under-utilization of a replica's resources, because only a single CPU core is used to execute transactions. In addition, if a transaction's execution utilises an accelerator, e.g., a GPU, this resource will be under-utilized if the accelerator is designed to execute highly parallel workloads or multiple workloads in parallel. In this section, we review several BFT systems that attempt to improve the performance of single replica transaction throughput.

High throughput Byzantine fault tolerance [142] introduces the CBASE Byzantine fault tolerant system, which parallelizes transactions within the bounds of state-machine replication. The system introduces an extra stage in the Byzantine fault tolerant consensus protocol pipeline, in which a replica that is ready to execute a batch of transactions runs an application called a *parallelizer*. The parallelizer determines which transactions can be run in parallel. The replica uses the parallelizers' output to execute multiple transactions in parallel while maintaining the same linearizability property as if all transactions were run sequentially.

The parallelizer takes as input the ordered transactions and, by understanding the semantics of the application built on top of the BFT consensus protocol, it can decide which transactions can be executed concurrently. Unfortunately, this level of understanding is not always guaranteed to be correct and if the parallelizer mistakenly believes two transactions can run in parallel, CBASE's linearizability is compromised.

All about Eve: Execute-Verify Replication for Multi-Core Servers [135] introduces the execute-verify model for parallelizing request execution. The execute-verify model requires the developer to build an *oracle* that, when provided with multiple requests, calculates the dependencies between them. Eve's workflow is as follows:

1. As in PBFT, the primary creates a batch of transaction requests and sends a pre-prepare message containing these requests to all the backups.
2. The primary and backups execute the requests in parallel, using the oracle to determine which requests can be executed concurrently.
3. The replicas update their state as part of the request execution and compare the result.
4. If the replicas do not agree on the result, the execution is rolled back and the requests are executed sequentially.

There are several issues that arise from Eve's design. First, the oracle is a best effort technique and for some workloads it may not be possible to write a high quality oracle. Second, users are now required to write a new oracle for every application. Finally, it is possible for a Byzantine primary to force sequential execution without being detected.

The key advantage of Eve over CBASE is that the oracle unlike the parallelizer can incorrectly select which transactions can run in parallel without compromising the system's linearizability.

Rex: Replication at the Speed of Multi-core [109] proposes an automated way of tracking transaction dependencies, however, it only considers crash faults. Rex works by tracking lock acquisition on the primary and then sending the lock acquisition order to backup replicas so that the backups know the dependency between requests before executing them. Rex has the following workflow:

1. Clients send requests to the primary and the primary executes requests across multiple threads.
2. While the primary executes requests, it tracks which locks each request obtains.
3. The primary sends the lock acquisition information when it replicates requests to backups.
4. Backups execute requests in parallel using the lock acquisition information to ensure the system's linearizability properties are maintained.

Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing [94] describes an extension to the Aptos Blockchain [10] that enables the blockchain's consensus protocol to execute some transactions in parallel. The Aptos blockchain, similar to other systems that utilize Byzantine consensus, elects a primary replica that orders

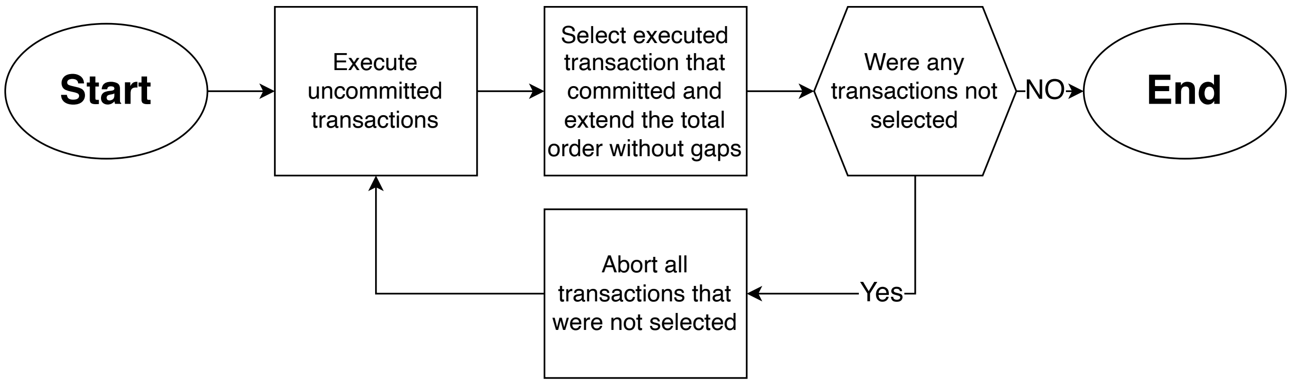

Start
Execute uncommitted transactions
Select executed transaction that committed and extend the total order without gaps
Were any transactions not selected
NO
End
Yes
Abort all transactions that were not selected

Figure 2.10: Transaction execution logic for a single group of ordered transactions.

transactions. The consensus protocol batches the ordered transactions into groups. Replicas in Block-STM sequentially consider a group of transactions and attempt to execute transactions in parallel. The system tracks conflicts that occur when transactions execute in parallel and aborts transactions that experience a conflict with another transaction. In addition, Block-STM ensures that transactions commit in an order that is serializable to the order defined by the primary replica. We show a simplified workflow of how Block-STM executes a group of transactions in fig. 2.10. While, Block-STM introduces and borrows many techniques from transactional memory research to reduce conflicts between transactions many conflicts still occur, requiring transactions to be re-executed and this re-execution wastes compute resources and introduces latency.

Rex, Eve, and CBASE make steps towards executing requests in parallel on backup replicas. However, none of these systems provides a technique to detect automatically and safely transaction dependencies and utilize this information to enable parallel execution of requests in all Byzantine environments. Block-STM is able to automatically and safely ensure transactions in a block will always commit. However, this safety guarantee comes at the cost of wasted resources and increased latency – up to 30% increase in latency – than executing transactions sequentially for workloads that have a large number of conflicting transactions [94].

2.3.4 Conclusions

The transaction throughput and latency of permissioned ledgers are directly connected to their consensus protocols. There have been many attempts to improve the transaction throughput and latency of Byzantine fault tolerant consensus protocols. In this section, we explored techniques that are designed to address these performance concerns. However, they either cannot fully utilize the multiple cores on a replica's CPU or are dependent on approximating which transactions can be run in parallel.

2.4 Summary

In this chapter, we explored the background materials on accountability and the performance of distributed ledgers and their consensus protocols. We began by looking at how consensus protocols are used in distributed ledgers, starting with Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) [48], the seminal work for Byzantine fault tolerant consensus protocols. We then explored more recent consensus protocols used in permissioned ledgers. However, we concluded that the consensus protocols do not provide the safety required for permissioned ledgers, as they only retain their safety properties when fewer than an arbitrary number of replicas are malicious.

We continued by looking for an alternative to ensuring the safety of permissioned ledgers by exploring accountability in Byzantine fault tolerant consensus protocols. We first considered the work for accountability in distributed systems by Haeberlen et al. [111] and then progressed to accountability in permissioned ledgers. We then concluded that while the current work on accountability in permissioned ledgers provides stronger guarantees than only utilizing Byzantine consensus protocols, the techniques that are employed to obtain accountability still fail after an arbitrary number of replicas have been compromised.

Finally, we shifted our focus to transaction throughput and latency, which fully utilises modern hardware. We explored techniques that are used to increase transaction throughput, including several permissioned ledger sharding techniques. Next, we explored techniques used in Byzantine fault tolerant consensus protocols to improve the transaction execution throughput. We observed how these techniques are not practical for permissioned ledgers as either they require oracles that attempt to predict if two transactions would conflict before the transactions are executed, or they do not provide any safety guarantees in an environment with malicious actors.

CHAPTER 3

Individual Accountability for Permissioned Ledgers

In this chapter, we explore accountability in permissioned ledgers. We address the issue that accountability in permissioned ledgers only functions correctly if less than an arbitrary number of replicas are dishonest.

We present IA-CCF, a new permissioned ledger that provides *individual accountability.* It can assign blame to individual members that operate misbehaving replicas regardless of the number of misbehaving replicas or members. IA-CCF achieves this by signing and logging BFT protocol messages in the ledger, and by using Merkle trees to provide clients with succinct, universally-verifiable *receipts* as evidence of successful transaction execution. Anyone can *audit* the ledger against a set of receipts to discover inconsistencies and identify replicas that signed contradictory statements. IA-CCF also supports *changes* to consortium membership and replicas by tracking signing keys using a sub-ledger of governance transactions.

In the context of describing IA-CCF, we show a design for an individually accountable permissioned ledger that retains its accountability property regardless of the number of replicas that become malicious. In addition, this chapter addresses a subset of the previously described concerns – see section 1.4.1 – that are raised by organizations that are evaluating utilizing a permissioned ledger.

3.1 Introduction

As discussed in Section 2.2, BFT protocols ensure safety (linearizability [118]) and liveness, but they can only do this if fewer than 1/3 of N replicas misbehave. With more misbehaving replicas, current permissioned ledger systems can no longer be trusted. When safety violations are detected, blame is shared.

Current systems try to avoid this problem by increasing replication [139, 107, 253]. Adding replicas does not help if they are controlled by the same consortium members and thus do not behave independently. Increasing the number of consortium members, however, is challenging or even infeasible in practice. For example, the Diem Association [77] had 26 members, which prevented it from offering a service with more than 26 independent replicas; other consortia are smaller, which results in fewer independent replicas [191, 17, 127]. Even for large consortia with reputable companies, a persistent attacker may slowly compromise $N/3$ replicas over time, e.g., by exploiting lax security practices, bribing members' employees or exploiting software vulnerabilities. Without accountability after a service compromise, there is also no perceived reputational loss that would incentivize members to prevent or disclose these incidents [120, 99, 42].

Prior work explores accountability for various types of distributed systems [257, 111, 110, 5, 152]. PeerReview [111] makes general message passing systems accountable. As we show in Section 3.6, applying such a general approach to a permissioned ledger system incurs high overhead: all messages must be signed, and auditing is expensive, because it correlates logs across many replicas. More recent work [219, 55, 201, 39] investigates accountability in BFT protocols and blockchains. These proposals, however, offer no guarantees when 2/3 or more replicas misbehave, because misbehaving replicas may rewrite the ledger history without detection.

In this chapter, we describe *Individual Accountability for CCF* (IA-CCF), a BFT permissioned ledger system that identifies misbehaving replicas and assigns blame to the individual members that operate them, *even if all replicas misbehave.* Individual accountability provides strong disincentives for misbehaviour.

IA-CCF is a prototype that extends CCF [206] with support for BFT and individual accountability, while retaining the same user programming model, key-value store, transaction execution engine, and model of governance for changes to the consortium membership and replica set.

IA-CCF supports individual accountability by introducing *Ledger PBFT* (L-PBFT), a new BFT state machine replication protocol that stores ordered transactions in the ledger together with the protocol messages from replicas that justify the execution order. L-PBFT maintains Merkle trees [163] over the ledger, and includes the roots of the trees in protocol messages. Since protocol messages are signed by the replicas, this commits them to the entire contents of the ledger.

IA-CCF then issues *receipts* to clients that provide succinct, universally-verifiable evidence that a transaction executed at a given position in the ledger. Receipts include signed protocol messages from multiple replicas that executed the transaction, thus binding them to a prefix of the ledger.

Given a collection of receipts that violates linearizability, anyone can audit the ledger against the receipts to assign blame to at least $N/3$ replicas. Auditing produces an irrefutable *universal proof-of-misbehaviour* (uPoM) in the form of contradictory statements signed by the same replica. The uPoM can be used by an *enforcer*, e.g., a court, to punish the members responsible for the misbehaving replicas. To provide accountability when all replicas misbehave, the enforcer may have to compel members to produce a ledger, imposing sanctions otherwise. While this formally adds a weak synchrony assumption, the enforcer chooses a conservative timeout to make blaming correct members unlikely in practice.

As an example of auditing, a client Alice may have a receipt for a transaction that executed at index i in the ledger and deposited \$1 million into client Bob's account. If Bob obtains the receipt from Alice and another receipt for a balance query transaction executed at index j ($j > i$) that does not show the balance, he may conduct an audit: he engages an enforcer to obtain the relevant ledger fragment, and replays the transactions between i and j to check for consistency with his receipts. If Bob is right, auditing produces a uPoM for at least $N/3$ replicas, which Bob sends to the enforcer to punish the consortium members responsible for the replicas.

To support changes to the consortium membership, IA-CCF uses *governance transactions* that alter the set of replicas and consortium members [206]. Governance transactions complicate receipt verification and auditing because they change the signing keys that must be considered. IA-CCF, therefore, records governance transactions in the ledger, which allows clients, replicas, and auditors to determine the set of valid signing keys. Clients do not need to keep the full ledger, but only receipts of governance transactions. Since governance transactions are relatively rare, this *governance sub-ledger* is significantly smaller than the full ledger.

Our IA-CCF prototype provides individual accountability without compromising on throughput or latency: it implements a commitment scheme for transaction batches with only a single signature per replica. This enables clients to receive results with receipts after only two network round-trips. Our evaluation shows that IA-CCF can execute over 47,000 tx/s with low latency.

The contributions of IA-CCF and the chapter structure are:

1. L-PBFT, a BFT state machine replication protocol that orders and stores transactions together with the protocol messages justifying the execution order in a ledger (Section 3.3.1, Section 3.3.2);

2. universally-verifiable client receipts that are generated efficiently with the ledger (Sec-

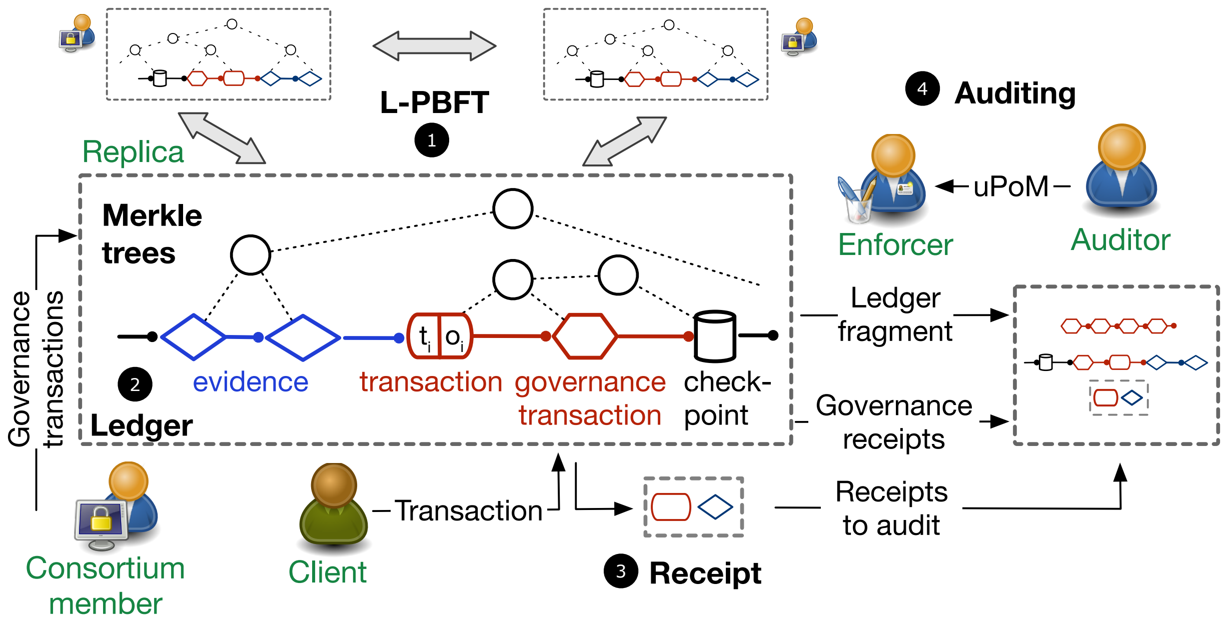


Figure 3.1: IA-CCF permissioned ledger system

tion 3.3.3);

3. an efficient auditing approach using the ledger and associated checkpoints, which produces short proofs-of-misbehaviour (Section 3.4); and

4. a governance mechanism for changing members and replica sets, allowing auditing to assign blame even after members have left (Section 3.5).

3.2 Overview of IA-CCF

Figure 3.1 shows IA-CCF's design. An IA-CCF deployment provides a *service*, with a well known name, to *clients*, which are identified by their signing keys. Clients send requests to execute *transactions* by calling stored procedures that define the service logic. Transactions are executed by *replicas* against a strictly-serializable *key-value store* that supports roll-back at transaction granularity. A transaction *request* t reads and/or writes multiple key-value pairs and produces a transaction *result* o.

Consortium *members*, also identified by their signing keys, own the service. They may be added or removed over the service lifetime. For this, members issue *governance transactions*, which change the consortium membership, add or remove replicas, and update stored procedures. The first governance transaction, the *genesis transaction* gt, defines the initial members and replicas. Its hash is the service name.

❶ **Ledger PBFT (L-PBFT)** is a BFT state machine replication protocol used by replicas to order transactions. L-PBFT is based on PBFT [49]. It provides linearizability and liveness if at most $f = \lceil N/3 \rceil - 1$ out of N replicas fail in a partially-synchronous environment [80].

❷ **Ledger.** L-PBFT maintains an append-only *ledger*, which stores each transaction request t and result o at a ledger index i. Since the consortium membership and the replica set are dynamic, the ledger also records governance transactions. They form a *governance sub-ledger*, which can be used to learn the public signing keys of active replicas and members at any index i.

To assign blame, the ledger also includes *evidence* that a transaction batch was committed by a quorum of replicas. This evidence consists of at least $N-f$ signed L-PBFT protocol messages for a batch. Finally, the ledger stores periodic checkpoints of the key-value store, allowing its state to be reconstructed by replaying the ledger from a checkpoint cp.

All entries in the ledger are bound by Merkle trees. Protocol messages for a transaction batch contain the roots of the Merkle trees. This commits replicas to the whole ledger while allowing succinct existence proofs for entries.

❸ **Receipts** are created by replicas and returned to clients. They bind request execution to members via the replicas' signatures over Merkle tree roots that contains the executed request and the ledger's history. If two or more receipts are inconsistent with any linearizable execution, at least $f+1$ replicas must have signed contradictory statements and can thus be assigned blame.

More precisely, a receipt R for $\langle t, i, o \rangle$ states that request t was executed at index i and produced result o. The receipt consists of $N-f$ protocol messages for t's batch, signed by different replicas, and a path from a Merkle tree root to the leaf that contains an entry for $\langle t, i, o \rangle$.

Clients may obtain receipts from a reply to a request they sent, from replicas, or from other clients. To validate a receipt, clients must check its signatures using the signing keys determined by the governance sub-ledger. A receipt therefore includes the ledger index of the last governance transaction, and clients must obtain the receipt of this governance transaction and all those preceding it. Clients cache governance transaction receipts and fetch missing ones from replicas.

❹ **Auditing** returns a *universal proof-of-misbehavior* (uPoM) if clients obtain receipts that are inconsistent with a linearizable execution. IA-CCF's ledger is universally-verifiable, i.e., anyone can act as an *auditor*: they replay the ledger, check consistency with receipts, and potentially generate a uPoM.

Since all consortium members and replicas may misbehave, an *enforcer*, e.g., a court, must compel members to produce a ledger copy for auditing, sanctioning non-compliance. The enforcer also punishes members based on uPoMs. It is unreasonable to assume that courts could run the service or audit long executions. Therefore, IA-CCF only requires enforcers to re-execute transactions between two consecutive checkpoints to verify a uPoM in the worst case.

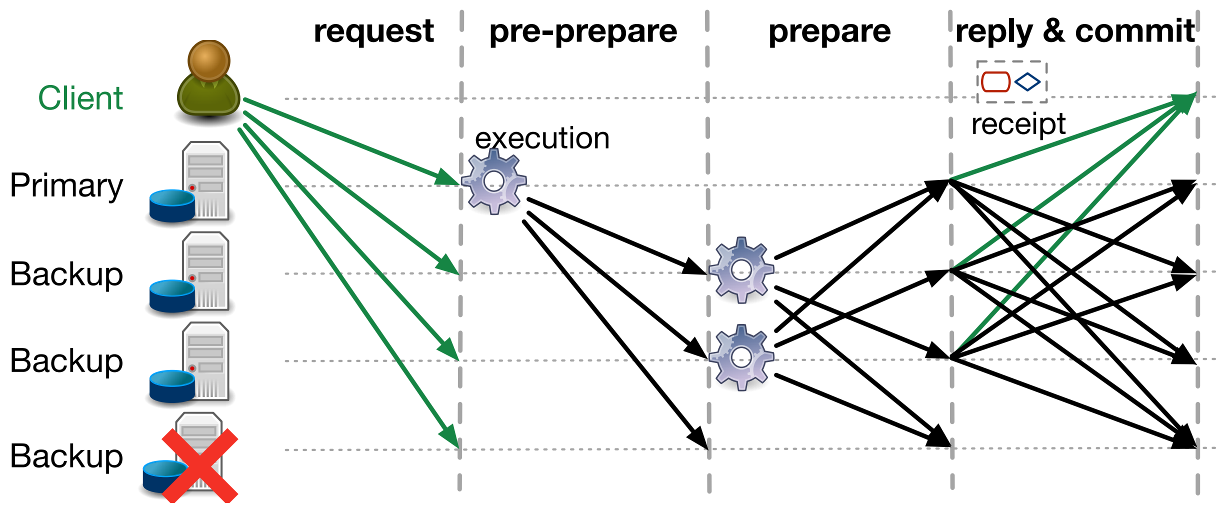


Figure 3.2: L-PBFT protocol with early execution and receipts

After a client passes a sequence of receipts and the governance sub-ledger to the auditor, the auditor confirms the receipts' validity by calculating a Merkle tree root and verifying the replica signatures. It then asks the enforcer to obtain the ledger fragment corresponding to the receipts from the replicas. The auditor checks the validity of the checkpoint cp referenced by the oldest receipt. It then replays the ledger from cp, re-executing transaction requests while checking for consistency with receipts (including governance transaction receipts). If an inconsistency is found at index i, the auditor creates a uPoM $\langle i, \mathcal{F}, cp, R\rangle$ with a ledger fragment $\mathcal{F}$, the checkpoint cp, and the inconsistent receipt R. The uPoM is then forwarded to the enforcer, which imposes penalties on the consortium members blamed.

Threat model. We assume a strong attacker that can compromise replicas, clients, auditors, and members to make them behave arbitrarily, but cannot break the cryptographic primitives. We trust the enforcer to assign blame to replicas and the members that operate them only when it verifies a valid uPoM or fails to obtain data for auditing. IA-CCF provides linearizability and liveness if fewer than 1/3 of the replicas are compromised [49]. With any number of compromised replicas, clients, auditors, and members, IA-CCF never punishes members that operate only correct replicas unless they fail to provide data for auditing. However, when 1/3 or more replicas are dishonest IA-CCF does not provide a liveness guarantee.

3.3 L-PBFT protocol and receipts

Next, we describe how L-PBFT maintains a ledger with transactions and evidence (Section 3.3.1), and how it handles view changes (Section 3.3.2). We then explain how evidence is used to create receipts (Section 3.3.3) and introduce performance optimizations (Section 3.3.4). For ease of presentation, we first assume a fixed replica set; we add dynamic membership

Algorithm 3.1: Ledger Practical Byzantine Fault Tolerance

```
on receiveTransactionRequest($t = \langle \text{request}, a, c, H(gt), m_i \rangle_{\sigma_c}$)
    Pre: verify($t$)
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$
on sendPrePrepare()
    Pre: isPrimary() $\wedge$ ready $\wedge$ $|\mathcal{T}| > 0 \wedge$ hasEvidence($\mathcal{M}, v, s - P$)
    $\mathcal{B} \leftarrow []$ ; $G \leftarrow \{\}$
    foreach $t \in \mathcal{T}$ do
        $\mathcal{B} \leftarrow \mathcal{B} \,||\, H(t)$ ; $\langle i, o \rangle \leftarrow$ execute($kv, t$); $G \leftarrow G||\langle t, i, o \rangle$
    $\langle E_{s-P}, \mathcal{P}_{s-P}, \mathcal{K}_{s-P} \rangle \leftarrow$ getEvidence($\mathcal{M}, v, s - P$)
    $\mathcal{L} \leftarrow \mathcal{L} \,||\, \mathcal{P}_{s-P} \,||\, \mathcal{K}_{s-P}$; $M \leftarrow M||\mathcal{P}_{s-P} \,||\, \mathcal{K}_{s-P}$
    $\mathcal{K}[v, s] \leftarrow$createNonce(); $\bar{M} \leftarrow$ getRoot($M$); $\bar{G} \leftarrow$ getRoot($G$)
    $pp = \langle \text{pre-prepare}, v, s, \bar{M}, \bar{G}, H(\mathcal{K}[v, s]), E_{s-P} \rangle_{\sigma_r}$
    $\mathcal{L} \leftarrow \mathcal{L} \,||\, pp \,||\, G$; $M \leftarrow M \,||\, pp$; $\mathcal{M} \leftarrow \mathcal{M} \cup \{pp\}$; $\mathcal{T} \leftarrow \{\}$; $s \leftarrow s + 1$
    sendToAllReplicas($pp \,||\, \mathcal{B}$)
on receivePrePrepare($pp = \langle \text{pre-prepare}, v, s', \bar{M}, \bar{G}, H(k), E_{s'-P} \rangle_{\sigma_r}, \mathcal{B}$)
    Pre: isBackup() $\wedge$ verify($pp$) $\wedge$ ready $\wedge$ $s' = s$ $\wedge$ $\mathcal{K}[v, s] =$ nil $\wedge$ hasRequests($\mathcal{T}, \mathcal{B}$) $\wedge$
            hasEvidence($\mathcal{M}, s' - P, E_{s'-P}$)
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{pp\}$; $G \leftarrow \{\}$
    foreach $h \in \mathcal{B}$ do
        $t \leftarrow$ removeTx($h, \mathcal{T}$); $\langle i, o \rangle \leftarrow$ execute($kv, t$);$G \leftarrow G||\langle t, i, o \rangle$
    $\langle E_{s-P}, \mathcal{P}_{s-P}, \mathcal{K}_{s-P} \rangle \leftarrow$ getEvidence($\mathcal{M}, v, s - P, E_{s-P}$)
    $\mathcal{L} \leftarrow \mathcal{L} \,||\, \mathcal{P}_{s-P} \,||\, \mathcal{K}_{s-P}$; $M \leftarrow M||\mathcal{P}_{s-P} \,||\, \mathcal{K}_{s-P}$;
    if getRoot($M$) $\neq \bar{M}$ or getRoot($G$) $\neq \bar{G}$ then
        undo($pp, kv, \mathcal{M}, \mathcal{B}, \mathcal{T}, \mathcal{L}$); return
    $\mathcal{L} \leftarrow \mathcal{L} \,||\, pp \,||G$; $M \leftarrow M \,||\, pp$; $\mathcal{K}[v, s] \leftarrow$createNonce()
    $p = \langle \text{prepare}, r, H(\mathcal{K}[v, s]), H(pp) \rangle_{\sigma_r}$
    sendToAllReplicas($p$); $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$; $s \leftarrow s + 1$
on receivePrepare($p = \langle \text{prepare}, r', H(k_{r'}), H(pp) \rangle_{\sigma_{r'}}$)
    Pre: verify($p$)
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
on batchPrepared($pp = \langle \text{pre-prepare}, v, s', \bar{M}, \bar{G}, H(k_p), E_{s'-P} \rangle_{\sigma_p}$)
    Pre: prepared($pp, \mathcal{M}$) $\wedge$ $\exists \langle \text{prepare}, r', H(\mathcal{K}[v, s']), H(pp) \rangle_{\sigma_{r'}} \in \mathcal{M}$
    $c = \langle \text{commit}, v, s', r, \mathcal{K}[v, s'] \rangle$
    sendToAllReplicas($c$); $\mathcal{M} \leftarrow \mathcal{M} \cup \{c\}$
    foreach $\langle t, i, o \rangle \in$getTxForBatch($\mathcal{L}, v, s'$) do
        sendReplyToClient($t$, $\langle \text{reply}, v, s', r, \sigma_r, \mathcal{K}[v, s'] \rangle$)
        if shouldSendReceipt($r, t$) then
            $\mathcal{S} \leftarrow$getMerklePath($G, i$)
            sendReceiptToClient($t$, $\langle \text{replyx}, v, s', \bar{M}, H(k_p), E_{s'-P}, H(t), i, o, \mathcal{S} \rangle$)
on receiveCommit($c = \langle \text{commit}, v, s', r', k_r \rangle$)
    Pre: verify($c$)
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{c\}$
```

in Section 3.5.

3.3.1 Protocol description

To support auditing, a BFT state machine replication protocol, such as PBFT [49], must integrate with a ledger: it must ensure that replicas agree on a ledger with both transactions (requests and results) and protocol messages. It must also handle non-determinism to enable replaying the ledger. L-PBFT addresses this issue by agreeing on non-deterministic inputs [50] and using *early execution*: it requires the primary replica to propose a transaction result, which the backup replicas must agree on for the batch to commit. L-PBFT then maintains Merkle trees over all ledger entries and puts the trees' roots in protocol message, which ensures that all replicas agree on a serial history of the ledger.

Figure 3.2 gives an overview of L-PBFT with early execution: first clients send transaction requests to all replicas. The primary orders the requests, groups them into *batches* and performs early execution. It then sends a pre-prepare message to the backups, which includes the request batch and the execution results. Upon receiving the pre-prepare, the backups execute the requests and confirm that the results match the primary's. If so, they send a prepare message to all other replicas. After a replica receives a pre-prepare and $N-f-1$ matching prepare messages for the same sequence number s and view v, the batch is *prepared* at the replica at v with s if all batches with lower sequence numbers have also prepared. A replica then sends a reply to the clients and commit messages to the other replicas. We say that a batch is *committed* at sequence number s if it has been prepared by $N-f$ replicas in the same view. A client has received a complete response when it has a *receipt* consisting of replies from $N-f$ replicas.

A naive approach would require each replica to sign two protocol messages, i.e., the pre-prepare/prepare and the commit message, for each committed batch. Instead, L-PBFT uses a novel *nonce commitment* scheme, in which replicas only sign the pre-prepare/prepare messages after including a hashed nonce. Instead of signing the commit, a replica includes the unhashed nonce. This effectively halves the signatures that replicas emit to commit batches successfully.

Algorithm 3.1 presents the pseudocode of L-PBFT. The replica state includes: the current view v and batch sequence number s; a set of transaction requests $\mathcal{T}$ waiting to be ordered; a message store $\mathcal{M}$; a nonce store $\mathcal{K}$; a boolean ready indicating if the replica can send/accept pre-prepare messages; a replica identifier r; the key-value store kv; the ledger $\mathcal{L}$; and the Merkle tree M that binds the ledger entries.

In receiveTransactionRequest (line 1), a replica adds a request message to $\mathcal{T}$, where a identifies the invoked stored procedure and its arguments, c is the client identifier, $H(gt)$ is the genesis transaction hash, m_i is the minimum index after which the request can be added to the ledger, and σ_c is the client signature. σ_c and $H(gt)$ ensure that requests cannot be forged or moved to a different ledger, and m_i allows clients to create an ordering dependency between the request and a previously executed transaction.

The function sendPrePrepare (line 4) uses early execution to include the execution result in the batch's Merkle tree root. The primary $p = v \bmod N$ collects a batch of transaction requests, executes them, and appends them to a new Merkle tree G. Then, the primary retrieves the com-

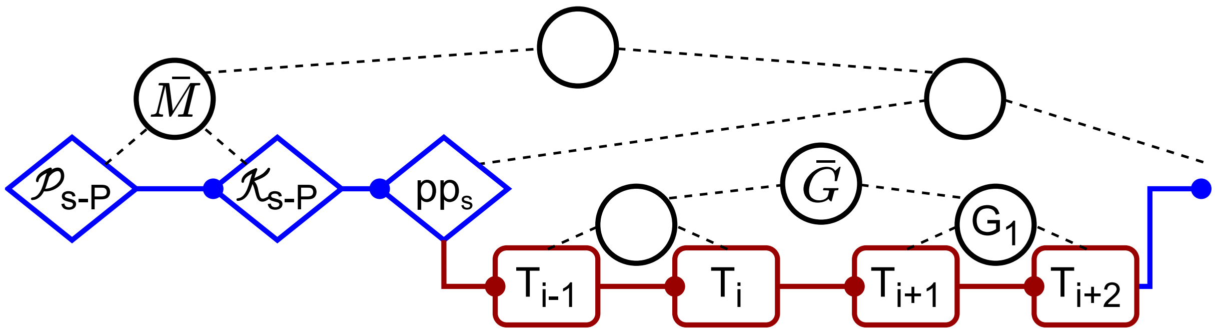


Figure 3.3: Ledger with evidence and Merkle trees

mitment evidence $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ for the batch at $s-P$ from the message store $\mathcal{M}$ and appends it to the ledger. E_{s-P} is a bitmap that records the replicas that supplied commitment evidence.

Next, the primary creates the pre-prepare message with the hash of a fresh nonce $\mathcal{K}[v, s]$, the root of the Merkle trees, $\bar{M}$ and $\bar{G}$, and signs it. G is a Merkle tree that contains all $\langle t, i, o\rangle$ entries in a batch. The complete pre-prepare message has two extra fields: i_g, the index of the last governance transaction, which allows clients to verify receipts with a changing set of replicas (see Section 3.5.2); and d_C, a digest of the key-value store state at the last checkpoint, which enables auditing from a checkpoint without replaying the ledger from the start (see Section 3.4).

By signing $\bar{M}$, the primary commits to the contents of the ledger, including the commitment evidence for $s-P$ that it retrieved and added to the ledger. It is important for the primary to order the evidence to ensure that replicas agree on the ledger: if replicas added their own evidence to the ledger when they received prepare and commit messages, their ledgers could diverge. The commitment evidence $\mathcal{P}_{s-P}$ contains $N-f-1$ prepare messages for sequence number $s-P$ and view v that match the pre-prepare at sequence number $s-P$ in the ledger. $\mathcal{K}_{s-P}$ are the $N-f$ nonces with hashes in the pre-prepare/prepare messages in $\mathcal{P}_{s-P}$. This evidence is sufficient to prove to a third party that the batch at $s-P$ prepared at $N-f$ replicas and therefore committed with s. The pre-prepare message along with the leaves of G are then added to the ledger.

The primary communicates its ordering decision by sending the pre-prepare message to all replicas, together with a list $\mathcal{B}$ of the hashes of transaction requests in execution order. The requests are sent separately by the clients, and the commitment evidence for $s-P$ is not included in the message. The pre-prepare messages are of a constant size except for a bitmap in which each bit represents a replica. Our implementation uses 8 bytes in the E_{s-P} bitmap to support up to 64 replicas, making the pre-prepare messages effectively O(1).

Figure 3.3 gives an example of the ledger state after this step. For each transaction in the batch, the primary adds a ledger entry in the order executed. The entry for T_i has the form $\langle t, i, o\rangle$ where o includes the reply sent to the client and the hash of the transaction's write-set; pp_s is the pre-prepare for s, and $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ are evidence that the batch at sequence number $s-P$ committed. L-PBFT pipelines the ordering of up to $P \geq 1$ concurrent batches to improve performance. Therefore, the commitment evidence lags P behind s, because it is unavailable when the primary sends the pre-prepare for s. Theorem A.1.2 in Appendix A shows that *early execution* maintains linearizability.

When a backup replica receives the pre-prepare (line 15), it rejects the message if it already sent a prepare for the same view and sequence number ($\mathcal{K}[v, s] \neq \textsf{nil}$). Otherwise, it checks if it already has the requests and commitment evidence referenced by the pre-prepare. Replicas store received requests, prepare, and commit messages in non-volatile storage ($\mathcal{M}$) until they receive (or send) a corresponding pre-prepare. To reduce network load, the primary does not resend requests or messages used as commitment evidence. If the backup is missing messages, it requests that the primary retransmit them, because a correct primary is guaranteed to have them.

The backup then executes the requests in the order prescribed by the primary, and adds the resulting transaction entries to a new Merkle tree G (line 19). Then, it adds the same $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ as the primary to the ledger. At this point, the ledger at the backup should be identical to the one at the primary just before the pre-prepare message is added. The backup checks that the roots of its Merkle trees match $\bar{M}$ and $\bar{G}$ in the pre-prepare, respectively. If not, the message is rejected, the entries for batch s are removed from the ledger, and the transactions are rolled back. Otherwise, the backup adds the pre-prepare to the ledger, followed by the leaves of the Merkle tree G, and sends a matching prepare message with the format $\langle \textsf{prepare}, \textsf{r}, H(\mathcal{K}[v, s]), H(pp)\rangle_{\sigma_\textsf{r}}$, where $H(\mathcal{K}[v, s])$ commits a fresh nonce, and $H(pp)$ is the pre-prepare's hash.

If a non-deterministic input is required during the execution of a transaction, L-PBFT nodes first agrees on the non-deterministic input values and then uses the inputs within the transaction execution [50]. Line 22 ensures that a backup's execution of batch $\mathcal{B}$ and its ledger are identical to those of the primary by comparing the Merkle roots $\bar{G}$ and $\bar{M}$. If this check fails, the backup rolls back execution and attempts to view change (Section 3.3.2). This way divergent execution due to bugs, i.e., failing to identify non-deterministic inputs, can affect liveness but not diverge the ledger.

In batchPrepared (line 30), the nonce commitment and early execution allow replicas to return replies to clients in two message round trips without signing reply or commit messages. When the batch prepares at replica r, it sends a commit message with the format $\langle \textsf{commit}, v, s', r, \mathcal{K}[v, s']\rangle$ where $\mathcal{K}[v, s']$ is the nonce the replica committed to in the pre-prepare/prepare messages that it sent for v and s'. Since the nonce $\mathcal{K}[v, s']$ is revealed to clients and replicas only when a replica prepares the batch having a pre-prepare/prepare message and the corresponding nonce can prove to a third party that the replica prepared the batch at v and s' (see Appendix A, Theorem A.1.3).

Algorithm 3.2: View Changes in L-PBFT

```
1  on sendViewChange()
2      Pre: primaryAppearsFaulty($v$)
3      $\mathcal{PP}$ = getPLastPrepared(msgs($\mathcal{L}$) $\cup$ $\mathcal{M}$)
4      $v = v + 1$; ready $\leftarrow$ false; $vc = \langle$view-change$, v, r, \mathcal{PP}\rangle_{\sigma_r}$
5      sendToAllReplicas($vc$);   $\mathcal{M} \leftarrow \mathcal{M} \cup \{vc\}$
6  on receiveViewChange($vc = \langle$view-change$, v', r', \mathcal{PP}\rangle_{\sigma_r}$)
7      Pre: $v' >= v \wedge$ verify($vc$) $\wedge$ hasPrepares(msgs($\mathcal{L}$) $\cup$ $\mathcal{M}$, getLast($\mathcal{PP}$))
8      $\mathcal{M} \leftarrow \mathcal{M} \cup \{vc\}$
9      if |getViewChanges($\mathcal{M}, v'$)| $> f \wedge v' > v$ then
10         $v = v' - 1$; setPrimaryApearsFaulty()
11         sendViewChange()
12 on sendNewView($v$)
13     Pre: isPrimary($v$) $\wedge \neg$ready $\wedge$ |getViewChanges($\mathcal{M}, v$)| $> N - f$
14     $\langle \bar{M}, E_{vc}, h_{vc}, \mathcal{PP}_{ov}\rangle$ = processViewChanges(getViewChanges($\mathcal{M}, v$))
15     $nv = \langle$new-view$, v, \bar{M}, E_{vc}, h_{vc}\rangle_{\sigma_r}$; $\mathcal{L} \leftarrow \mathcal{L} \,||\, nv$; $M \leftarrow M \,||\, nv$
16     sendToAllReplicas($nv$)
17     resendPreparesInNewView($\mathcal{PP}_{ov}$); ready $\leftarrow$ true
18 on receiveNewView($nv = \langle$new-view$, v, \bar{M}, E_{vc}, h_{vc}\rangle_{\sigma_{r'}}, \mathcal{PP}_{nv}$)
19     Pre: isPrimary($r', v$) $\wedge$ hasRequests($\mathcal{T}, \mathcal{PP}_{nv}$) $\wedge$ hasEvidence($\mathcal{M}, \mathcal{PP}_{nv}$)
           $\wedge\, r' \neq r \wedge \neg$ready $\wedge$ |getViewChanges($\mathcal{M}, E_{vc}, h_{vc}$)| $> N - f$
20     $\langle \bar{M}', \mathcal{PP}'_{ov}\rangle$ = processViewChanges(getViewChanges($\mathcal{M}, E_{vc}, h_{vc}$))
21     if $\bar{M}' = \bar{M}$ then
22         $\mathcal{L} \leftarrow \mathcal{L} \,||\, nv$; $M \leftarrow M \,||\, nv$
23         if ready $\leftarrow$ processPreparesInNewView($\mathcal{PP}_{nv}, \mathcal{PP}'_{ov}$) then return
24     undo($nv, s, \mathcal{M}, \mathcal{L}$)
```

Finally, a replica r commits a prepared batch v, s' after it receives $N{-}f$ commit messages, including its own. The nonce hashes in the commit messages must match the ones in the pre-prepare/prepare messages.

We prove that L-PBFT produces a linearizable execution order in Appendix A, Theorem A.1.4.

3.3.2 View changes

During the L-PBFT protocol execution, the primary may misbehave or be slow, which requires a *view change*. The change of the primary must be done in a manner that does not preclude auditing, which is a new requirement that goes beyond PBFT's view change protocol. L-PBFT view changes are auditable and must provide proof that a batch's re-execution produces the same result as the original execution.

L-PBFT addresses this as follows: it sends the evidence that batches prepared during view changes and includes the Merkle tree root $\bar{G}$ of a batch and its execution in the *pre-prepare* message, which ensures that batches are re-executed consistently. During a view change, each replica sends a view-change message with information about prepared requests. The primary for a new view v' sends a new-view message backed by $N{-}f$ view-change messages for v'. For each sequence number with a prepared batch in the view-change messages, the primary picks the batch that prepared with the largest view and proposes it in v'. Since all committed requests have also prepared, this ensures linearizability with batch execution ordered by the sequence numbers at which batches committed.

Algorithm 3.2 formalizes the pseudocode for view changes. If the primary for view v appears faulty or slow, a replica sends a view-change message, $\langle \mathsf{view\text{-}change}, v+1, \mathsf{r}, \mathcal{PP} \rangle_{\sigma_{\mathsf{r}}}$, to all other replicas (line 1), where $\mathcal{PP}$ contains the last P pre-prepare messages that prepared locally (line 3). Only the last message in $\mathcal{PP}$ is required to provide linearizability, because it includes the Merkle tree roots $\bar{M}$ and $\bar{G}$ that determine the ledger contents up to that point. The other pre-prepare messages are used during auditing to verify that replicas reported the batches they prepared in view-change (Section 3.4).

When replicas receive a view-change message (line 6), before processing it, they fetch missing prepare messages from the sender to prove that the last pre-prepare in $\mathcal{PP}$ has prepared. When replicas increment v, they set ready to false (lines 4 and 11), which ensures that they do not send or accept pre-prepare messages until they have completed the new-view.

After accepting $N{-}f$ view-change messages for the new view (line 12), the new primary calls processViewChanges, which picks the view-change message vc_{lp} with the last prepared pre-prepare message pp_{lp} from those with the largest view number. It then updates the ledger to match the Merkle roots in pp_{lp} by fetching missing ledger entries from replicas that sent matching prepare messages. Since at least $f{+}1$ of those are correct, this is always possible. The primary checks that all messages in $\mathcal{PP}$ of vc_{lp} appear at the right ledger positions; if not, it discards vc_{lp} and re-tries (omitted from Algorithm 3.2).

Next the primary resets the ledger to $s_{lp}{-}P$, because the batches up to this point are guaranteed to have committed. It saves all the request batches and commitment evidence for sequence numbers between $s_{lp}{-}P$ and s_{lp} and returns it in $\mathcal{PP}_{ov}$. This is needed to resend pre-prepare messages for the prepared batches in the new view. The function ends by adding an entry with the $N{-}f$ view-change messages that it accepted to the ledger in order of increasing replica identifier; h_{vc} is the hash of that entry and E_{vc} is a bitmap with the replicas that sent the messages. It returns the root of the Merkle tree $\bar{M}$, E_{vc}, h_{vc}, and $\mathcal{PP}_{ov}$ (line 14). The primary appends the new-view to the ledger, sends it to all replicas, resends the prepared batches in pre-prepare messages in the new view, and adds them to the ledger.

When backups receive the new-view (line 18), they obtain missing view-change messages, requests and evidence that it references, and call processViewChanges. If it returns a Merkle tree root equal to the one in new-view, they accept the message, add it to the ledger, and process the pre-prepare messages $\mathcal{PP}_{nv}$. If these match the batches and evidence in $\mathcal{PP}'_{ov}$ for the same sequence numbers, they are added to the ledger; otherwise, all changes are undone.

Algorithm 3.3: Verifying Receipts

```
on verifyReceipt($\langle t,i,o\rangle, \langle v,s,M,H(k_p),E_{s-P},i_g,d_C\rangle, \sigma_p, E_s, \Sigma_s, \mathcal{K}_s, \mathcal{S}\rangle$)
    $\bar{G}' \leftarrow$ pathHash ($\langle t,i,o\rangle$)
    foreach $G_i \in \mathcal{S}$ do
        $\bar{G}' \leftarrow$ pathHash ($\bar{G}'$, $G_i$)
    $pp = \langle$pre-prepare$, v, s, \bar{M}, \bar{G}', H(k_p), E_{s-P}, i_g, d_C\rangle$
    if not checkSignature ($\sigma_p$, $pp$) then  return false
    foreach $r \in E_s$ do
        if $r = p \wedge H(\mathcal{K}_s[p]) \neq H(k_p)$ then  return false
        if $r \neq p \wedge$ not checkSignature ($\Sigma_s[r]$, $\langle$prepare$, r, H(\mathcal{K}_s[r]), H(pp_{\sigma_p})\rangle$) then  return false
    return true
```

3.3.3 Receipts

To allow third parties to audit the ledger against the transaction results returned to clients, L-PBFT returns *receipts*, which are statements signed by $N-f$ replicas that a transaction request t executed at index i and produced a result o. L-PBFT exploits the per batch Merkle tree G together with the nonce commitment scheme (Section 3.3.1) to avoid having replicas sign the reply for each request.

Creating receipts. When a transaction batch described by pre-prepare pp prepares at replica r, view v and sequence number s' (Algorithm 3.1, line 30), it sends $\langle \text{reply}, v, s', r, \sigma_\text{r}, \mathcal{K}[v,s']\rangle$ to every client with a transaction in the batch. (If the client has multiple transactions in the batch, only one reply is sent.) By revealing the nonces, the replicas provide the client with proof that they claimed to have prepared the batch without a signed reply.

Only a designated replica, chosen based on t, sends the result and the rest of the receipt to the client (line 36). The replica computes a list of sibling hashes $\mathcal{S}$ along the path from the leaf to the root of the per-batch Merkle tree G. For the example of T_i in Figure 3.3, $\mathcal{S}$ consists of the digest of T_{i-1} and G_1, which is sufficient to recompute $\bar{G}$ given T_i. It then sends the client $\langle \text{replyx}, v, s', \bar{M}, H(k_p), E_{s'-P}, i_g, d_C, H(t), i, o, \mathcal{S}\rangle$, where i_g and d_C are used for auditing.

Verifying receipts. The client waits for $N-f$ replicas to send reply messages with the same v and s, and for a replyx message with the same v and s. It then recreates the pre-prepare and prepare messages (Algorithm 3.3, line 6), with the information in replyx and the hashes of the nonces, and verifies the signatures. (We describe how to determine N and verify signatures under dynamic membership in Section 3.5.2.) This step is shared across all transaction requests that the client may have sent in the batch.

IA-CCF uses the Merkle tree G to bind signatures in pre-prepare and prepare messages to transactions in the batch, enabling replicas to produce a single signature per batch. In the example in Figure 3.3, the client checks if $\bar{G} = H(H(H(T_{i-1})||H(\langle t,i,o\rangle))||G_1))$ (lines 2–4). If the hashes match, the client has a valid receipt, i.e., a statement signed by $N-f$ replicas that a request t executed at index i and produced a result o; otherwise (or if the client does not receive replies before a timeout), it retransmits the request and selects a different replica to send back replyx. (The application is responsible for ensuring exactly-once semantics if needed.)

Clients store the receipt for $\langle t, i, o\rangle$ as $\langle v, s, \bar{M}, H(k_p), E_{s-P},\ i_g, d_C, \sigma_p, E_s, \Sigma_s, \mathcal{K}_s, \mathcal{S}\rangle$ where Σ_s is a list of the signatures in prepare messages, $\mathcal{K}_s$ is a list of nonces, and E_s is a bitmap indicating the replicas with entries in Σ_s, and $\mathcal{K}_s$, sorted in increasing order of replica identifier. All receipt components, including common hashes in $\mathcal{S}$, are shared across requests in the same batch.

Clients must store the receipts together with the transaction request and the corresponding result to resolve future disputes. This is not a burden because receipts are concise: all components have constant size, except $|\mathcal{S}|$, whose number of entries is logarithmic in the number of requests in a batch; Σ_s and $\mathcal{K}_s$ have up to $N-f$ entries. In addition, most intermediate hashes in $\mathcal{S}$ can be shared across collections of receipts. We explored using signature aggregation [37] to reduce the size of Σ_s, but, for realistic consortia sizes, verifying the signatures becomes more expensive than our current implementation.

3.3.4 Performance optimizations

L-PBFT includes several optimizations to improve transaction and auditing throughput:

Checkpoints in L-PBFT allow new replicas to start processing requests without having to replay the ledger from the start (Section 3.5.1); slow replicas to be brought up-to-date using a recent checkpoint; and auditing to start from a checkpoint instead of the beginning of the ledger (Section 3.4.1).

Checkpoints include the key-value store and the Merkle tree M's newest leaf, root, and the connecting branches. Replicas create a checkpoint cp_s when they execute a batch with sequence number s such that $s \bmod C = 0$. The primary adds a batch to the ledger at sequence number $s{+}C$ with a special *checkpoint transaction*, which records the checkpoint digest. C is chosen to give replicas enough time to complete a checkpoint without delaying L-PBFT execution. Backups only accept the pre-prepare for $s{+}C$ if they compute the same checkpoint digest for sequence number s.

When a replica fetches checkpoint cp_s, it also retrieves the ledger up to s. It does not need to replay the ledger or check all signatures (with the exception of governance transactions; Section 3.5.2). Instead, it checks the signatures in checkpoint receipts and that the ledger contents between consecutive checkpoints are consistent with the Merkle tree roots in the corresponding receipts. This is done from the start of the ledger until $s{+}C$.

Cryptography. L-PBFT reduces the impact of cryptographic operations. Signature verification is parallelized for messages received from replicas and clients [56, 31] to improve throughput and scalability. All messages are sent over encrypted and authenticated connections, even signed messages. This mitigates denial-of-service attacks that consume replica resources verifying signatures [56].

To further improve performance, backups overlap the execution of request batches with the validation of pre-prepare signatures. They only send the prepare after both completed. Since pre-prepare messages are received over authenticated connections, this always succeeds for correct primaries.

3.4 Auditing and enforcement

In this section, we describe how auditing produces *universal proofs-of-misbehaviour* (uPoMs) when linearizability is violated (Section 3.4.1), and the role of the enforcer in obtaining ledgers for auditing and punishing the members responsible for misbehaving replicas (Section 3.4.2). We first focus on the simpler case of auditing without governance transactions; Section 3.5 describes governance transactions and their impact on auditing.

3.4.1 Auditing

An audit is triggered when someone, usually a client, obtains a sequence of transaction receipts that violate linearizability, i.e., when no linearizable execution of the stored procedures that define the transactions can produce the sequence of receipts. The mechanism to detect linearizability violations is application dependent. It involves clients, which interact through a sequence of transactions, exchanging receipts and using the application semantics to reason about the correctness of the receipt sequence.

The goal of auditing is to detect dishonest behaviour regardless of the number of misbehaving replicas, i.e., it must find proof of misbehaviour even if all replicas collude and rewrite the ledger. IA-CCF therefore provides proof of transaction execution in both the ledger and receipts—even if the ledger is rewritten, the misbehaving replicas are unable to alter the receipts.

An audit can be performed by anyone, and begins when an *auditor* receives a collection of receipts. Next, the auditor requests a *checkpoint* and a *ledger fragment* that contains the section of the ledger spanning the receipts. Any honest replica that signed the receipts is guaranteed to have the checkpoint and ledger fragment. When the auditor receives the requested data, it verifies the ledger structure by checking the protocol messages and their order, and validating any signatures in the ledger—but it does not re-execute transactions. Then, the auditor checks that the transactions referenced by the receipts are present at the right positions in the ledger.

If the above steps have not discovered misbehaviour, there remains the possibility that at least $N - f$ of the replicas colluded and agreed on an incorrect execution result. Therefore, the

Algorithm 3.4: Ledger Auditing (simplified)

```
1  on audit($\mathcal{R} = \{\langle\langle t_0, i_0, o_0\rangle, x_0\rangle, \ldots, \langle\langle t_k, i_k, o_k\rangle, x_k\rangle\}$)
2      auditReceipts($\mathcal{R}$)
3      $C_0, s_{C_0}, \mathcal{L}$ ←getCheckpointAndLedger($x_0, x_k$)
4      verifyReceiptsInLedger($\mathcal{R}, \mathcal{L}$)
5      replayLedger($C_0, s_{C_0}, \mathcal{L}$)
6  on auditReceipts($\mathcal{R} = \{\langle\langle t_0, i_0, o_0\rangle, x_0\rangle, \ldots, \langle\langle t_k, i_k, o_k\rangle, x_k\rangle\}$)
7      foreach $\langle\langle t_i, i_i, o_i\rangle, x_i\rangle \in \mathcal{R}$ do
8          if not verifyReceipt($\langle t_i, i_i, o_i\rangle, x_i$) then return $invalidReceipt$
9  on getCheckpointAndLedger($x_0, x_k$)
10     for $C_0, s_{C_0}, \mathcal{L}, r$ ←enforcerGetLedgerPackage($x_o, x_k$) do
11         uPoM ← nil
12         foreach $s \in s_{C_0}$, ..., seqno($x_k + P$) do
13             if not isBatchWellformed($\mathcal{L}, s$) then
14                 $\mathcal{F}$ ←createLedgerFragment(nil, $s, \mathcal{L}$)
15                 uPoM ← $\langle$nil$, \mathcal{F}, r\rangle$; send(uPoM); return
16         if uPoM = nil then return $C_0, s_{C_0}, \mathcal{L}$
17 on verifyReceiptsInLedger($\mathcal{R}, \mathcal{L}$)
18     foreach $\langle\langle t_i, i_i, o_i\rangle, x_i = \langle v, s, H(k_p), \ldots \mathcal{K}_s, \mathcal{S}\rangle\rangle \in \mathcal{R}$ do
19         if not isReceiptInBatch(x_i, $\mathcal{L}$) then
20             $\mathcal{F}$ ←createLedgerFragment(nil, $s, \mathcal{L}$)
21             uPoM ← $\langle \mathcal{F}, \langle\langle t_i, i_i, o_i\rangle, x_i\rangle\rangle$; send(uPoM); return
22 on replayLedger($C_0, s_{C_0}, \mathcal{L}$)
23     $s_{cp} \leftarrow s_{C_0}$; $cp \leftarrow C_0$; kv ← loadCheckpoint($s_{C_0}, C_0$)
24     foreach $s \in s_{C_0}$, ..., seqno($x_k$) do
25         foreach $\langle t_i, i_i, o_i\rangle \in s$ do
26             $\mathcal{L}$, kv ←replayRequest($\mathcal{L}$, kv, $t_i$)
27             if not verifyReplay($\mathcal{L}$, kv, $\langle t_i, i_i, o_i\rangle$) then
28                 $\mathcal{F}$ ←createLedgerFragment($s_{cp}, s, \mathcal{L}$)
29                 uPoM ← $\langle i_i, \mathcal{F}, cp\rangle$; send(uPoM); return
30         if $s \bmod C = 0$ then
31             $s_{cp} \leftarrow s$; $cp \leftarrow$ createCheckpoint(kv)
```

auditor loads the checkpoint and replays the transactions from the ledger fragment to check if execution results are correct. Throughout this process, if dishonest behaviour is uncovered, the auditor can produce a universally-verifiable proof that at least $f+1$ replicas misbehaved.

More formally, Algorithm 3.4 presents the pseudocode for the auditing process. First, the auditor receives an ordered set of receipts $\mathcal{R} = \{\langle\langle t_0, i_0, o_0\rangle, x_0\rangle, \ldots, \langle\langle t_k, i_k, o_k\rangle, x_k\rangle\}$ where $k \geq 1$ and $\forall l \in [0, k) : s_l \leq s_{l+1}$. Here, s_i is the sequence number that is specified in x_i. The auditor invokes auditReceipts (line 2) to check if the receipts are valid and the minimum index requirements have been satisfied. If there is a receipt that violates the requirement in the request, all replicas that have signed the receipt can be blamed.

After that, the auditor must obtain a ledger fragment and checkpoint that are *complete* in relation to $\mathcal{R}$ (line 3). We formally define completeness in Appendix B, but intuitively the ledger fragment must be (i) *well-formed*; (ii) include all batches and evidence between sequence numbers s_{C_0} and s_k where s_{C_0} is the sequence number of the checkpoint transaction that is linked in the first receipt; and (iii) include view-change messages for all views in $\mathcal{R}$. The transaction and checkpoint at s_{C_0} must match the checkpoint linked in the first receipt. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of views where there are at most f Byzantine failures. It is well-formed if it is valid, or if it would be valid if not for the incorrect execution of some transactions and/or checkpoints. A correct replica always maintains a well-formed ledger.

In getCheckpointAndLedger (line 3), the auditor, with the help of an *enforcer*, obtains ledger fragments and checkpoints from replicas that signed the latest receipt with the highest view number in $\mathcal{R}$ (line 10). The auditor checks if responses are complete in relation to the receipts. If a ledger fragment is not well-formed or misses the required view-change messages, the auditor can blame the responding replica. Below, we assume that the responses contain no invalid signatures, we show in Appendix B how the auditor handles that case.

If the batch at s_{C_0} is not a checkpoint or the checkpoint digest does not match the first receipt, the auditor can assign blame to the intersection of replicas that have signed the batch at $s_{C_0} + C$ and the first receipt, as the checkpoint reference in a receipt must always link to the last committed checkpoint. If the fragment is not long enough to include the sequence number in one of the receipts, there must be misbehaviour during a view change. The auditor can then blame at least $f{+}1$ misbehaving replicas: the intersection of the replicas that participated in a view change and that also signed the receipt. A correctness proof and the details of obtaining a complete ledger fragment and checkpoint are described in Appendix B, Lemmas B.1.1 and B.1.3.

After obtaining a well-formed ledger, in verifyReceiptsInLedger (line 4), the auditor compares the receipts with the ledger. If a receipt $\langle\langle t_k, i_k, o_k\rangle, x_k\rangle$ does not match the batch at s_k in the ledger fragment, we show in Theorem B.1.2 that the auditor can assign blame to $f{+}1$ misbehaving replicas. In summary, there are three cases: (i) the pre-prepare with sequence number s_k in $\mathcal{L}$ has a view number $v_l = v_k$; (ii) $v_l > v_k$; or (iii) $v_l < v_k$. In case (i), the ledger fragment contains evidence that the batch with sequence number s_k has prepared at $N{-}f$ replicas. Since at least $f{+}1$ of the replicas that have prepared the batch also signed the receipt, they can be blamed. In case (ii), since $v_l > v_k$, there must be at least $N{-}f$ view-change messages from different replicas that transition to a view greater than v_k in the ledger fragment but claim not to have prepared the batch in the receipt in view v_k. Since there are at least $f{+}1$ of those replicas that also signed the receipt, they can be blamed. In case (iii), since $v_k > v_l$ and the ledger fragment is complete in relation to the receipt, there must be at least $N{-}f$ view-change messages from different replicas that transition to a view greater than v_l in the ledger fragment. Similarly, the intersection of those replicas and the ones that signed the receipt can be blamed.

Since $N{-}f$ or more replicas may have misbehaved, it is necessary to replay transaction execution to check if the results are correct. The auditor does not need to understand the semantics of the service; it can retrieve the code of the stored procedures from C_0. The auditor sets the service state to the checkpoint value and replays transactions. If replaying a transaction fails to match the result in the ledger, the auditor can assign blame to any replica that signed the batch that contains the transaction. This is shown in replayLedger (line 5).

3.4.2 Enforcement

Since IA-CCF provides individual accountability even if all replicas and members misbehave, there must be an *enforcer* outside of the system to obtain checkpoints and ledger fragments for auditing, and to punish members responsible for misbehaving replicas. For example, consortium members may sign a binding contract to establish penalties if a uPoM proves that one of their replicas misbehaved, or if they fail to produce checkpoints and ledgers for auditing by an agreed deadline. These penalties may be imposed by the enforcer via arbitration [23] or a court of law [24].

The enforcer receives a set of receipts $\mathcal{R}$ from the auditor (Algorithm 3.4, line 10). It then verifies that the receipts are valid, and requests all of the replicas that signed the latest receipt with the highest view for a ledger fragment that is complete in relation to $\mathcal{R}$.

Correct replicas respond to the enforcer quickly. If the enforcer does not receive a response from a replica within a reasonable duration, e.g., within minutes, it contacts the controlling consortium member to obtain the checkpoint and ledger. If the member fails to provide this information by an agreed deadline, e.g., within days, it is punished according to the contract. This is important to ensure that misbehaving members cannot escape punishment by failing to produce information for auditing. However, it introduces a weak synchrony assumption that may lead to the punishment of honest but slow members. We expect that the deadline is chosen conservatively to make this unlikely in practice. After the deadline elapses, the enforcer either returns to the auditor $f{+}1$ responses, or it penalizes $f{+}1$ unresponsive replicas.

The enforcer also punishes members if a uPoM proves that one of their replicas misbehaved. When it receives a uPoM, it checks its validity by carrying out an audit, as described in Section 3.4.1, but the ledger fragment size and the number of transactions to replay is bounded by the transactions between two consecutive checkpoints. Furthermore, if there are fewer than $N - f$ misbehaving replicas, the uPoM does not require the enforcer to replay transactions. If the uPoM is incorrect, the enforcer punishes the auditor; otherwise, it punishes the members responsible for at least $f{+}1$ misbehaving replicas.

In practice, we envision the power of the enforcer being derived from the legally binding consortium membership bylaws signed by all members when they join the consortium [149, 190]. Further, we expect the load placed on the enforcer to be small, because auditing is rare—IA-CCF provides linearizability with up to f misbehaving replicas and the enforcer penalizes entities that request information for auditing and fail to produce a valid, minimal uPoM.

3.5 Reconfiguration and auditing

In this section, we describe how IA-CCF can change the consortium membership and the active replica set (Section 3.5.1). We explain how this impacts receipt validation (Section 3.5.2) and auditing (Section 3.5.3).

3.5.1 Reconfiguration

An IA-CCF deployment must handle changes to the active member and replica set while supporting auditing, regardless of how many replicas misbehave. For this, IA-CCF maintains governance data in the form of a *configuration*, which includes the public signing keys for members and replicas and an endorsement of each replica's signing key signed by the member responsible.

Changing the configuration enables members to change the active replica set. This is initiated by a *referendum*: members propose an updated configuration followed by the other members voting on the proposal. The number of votes required to pass the proposal is part of the service's state.

When voting on proposals, members must ensure the integrity of the service, e.g., disallowing an individual member from controlling too many replicas. Members are also limited to adding or removing at most f replicas, which ensures that the configuration change does not affect the service's liveness.

A referendum is carried out through governance transactions: a member proposes a new configuration by sending a *propose* transaction request. This is followed by members sending *vote* requests. Upon executing the final *vote* transaction required for a referendum to pass at sequence number s, the primary ends the current batch, and initiates the reconfiguration process.

A *reconfiguration* first adds evidence for the referendum to the ledger. This is done as part of the old configuration by the primary sending P pre-prepare messages without batched requests, called the *end-of-configuration* batches. The pre-prepare message for the end-of-configuration batch at sequence number $s+P$ contains evidence that the batch at s committed (Section 3.3). In addition, these pre-prepare messages include an extra field: the *committed* Merkle root, which is the root of the Merkle tree at s. This evidence is required for auditing: it commits the replicas that signed the P^{th} end-of-configuration batch to triggering the reconfiguration. Similarly, the signatures of the replicas that prepared the P^{th} end-of-configuration batch must be included in the ledger in the same configuration. Following the first P end-of-configuration batches, the primary pre-prepares another set of P end-of-configuration batches. The configuration change takes effect at $s+2P$.

The replicas in the new configuration create a checkpoint of the key-value store at sequence number $s+2P$. The primary creates a pre-prepare for the checkpoint at $s+2P+1$, followed by

P *start-of-configuration* pre-prepare messages with empty request batches. This ensures that a correct replica commits the checkpoint transaction before other transactions are executed in the new configuration. If any of the end/start-of-configuration batches correspond to a checkpoint sequence number, the checkpoint is skipped. Therefore, the checkpoint digests d_c in the pre-prepare messages always refer to checkpoints in the same configuration.

A newly added replica first obtains the ledger and a recent checkpoint, and replays the ledger from that checkpoint (Section 3.3.4). Replicas that are no longer part of the new configuration retire after sending the pre-prepare for $s+2P$. Removed members and replicas should delete their private signing keys to provide forward security. This prevents them from being blamed for future compromises as once a member or replica has been removed there are no valid uses for the private signing keys. Additionally, the removed members and replicas retain their own and other members and replicas public keys to allow them to authenticate transactions in the ledger.

3.5.2 Governance sub-ledger and receipts

When a client verifies a receipt, it must know which replicas were active when the receipt was created. IA-CCF addresses this with the help of the governance sub-ledger.

Governance transactions are recorded in the ledger and used by auditors to determine the active configuration. Clients, however, do not have a copy of the ledger, but need to verify receipt signatures. To do this, they store receipts for all governance transactions and, for each reconfiguration, they also store the receipts for the P^{th} end-of-configuration batch. We refer to this as the receipts of the *governance sub-ledger*. A client checks that a transaction receipt for index i is valid by considering the governance sub-ledger from the genesis transaction gt up to i. The client verifies the governance receipts, and if successful, the replica signing keys at index i are used to validate the receipt (Section 3.3).

This raises the challenge of how a client determines that it has *all* required governance receipts. IA-CCF includes the ledger index of the last governance transaction in each pre-prepare message and receipt (i_g). A client can request missing receipts from replicas by traversing the sequence of governance receipts. It verifies received receipts incrementally and caches them locally.

With reconfiguration, the definition of a valid receipt is extended: a valid receipt R must include valid governance receipts from gt up to the configuration that produced R.

3.5.3 Auditing

Reconfiguration introduces several new tasks for the auditor: it must consider the governance sub-ledger with receipts; validate that reconfigurations were executed correctly; and ensure that that only one configuration was active for any given index or sequence number. Next, we provide a summary of the required changes to the auditing process; a detailed correctness proof is included in Appendix B.2.

A client initiates an audit by sending inconsistent receipts and the supporting governance receipts to an auditor. The auditor replays these governance transactions to determine the signing keys required to verify each client receipt. After verifying the receipts, the auditor requests a ledger fragment and checkpoint from the enforcer.

The auditor may uncover that multiple configurations were active for a given index or sequence number, this can happen when misbehaving replicas fork or rewrite the ledger. We call this a *fork in governance*. If the auditor finds a fork, there are two P^{th} end-of-configuration batch receipts with the same preceding configuration that are not *equivalent*: they are at different indices or sequence numbers, or their pre-prepare messages do not contain the same committed Merkle root, i.e., they are not preceded by the same governance transactions. In this case, the auditor assigns blame to the replicas that signed both receipts, as a correct replica that prepares a P^{th} end-of-configuration batch commits the final *vote* transaction that triggers reconfiguration.

If the enforcer cannot obtain the required information for a valid receipt R from the sequence of provided receipts, there must be misbehaving replicas. In addition to the misbehaviour described in Section 3.4.1, the misbehaving replicas may have created a fork in governance or incorrectly prepared the P^{th} end-of-configuration batch that succeeds the configuration that produced the receipt R (see Lemmas B.2.2 and B.2.5).

Another possibility is that the configuration that produced a receipt R for a sequence number s may not match the configuration that prepared the batch at s in a well-formed ledger fragment. In this case, blame is again assigned to the replicas that signed R and prepared the P^{th} end-of-configuration batch that succeeds the configuration that produced R (see Lemma B.2.3).

After assigning blame, the auditor sends a uPoM to the enforcer with the supporting governance receipts.

3.6 Evaluation

We evaluate IA-CCF to understand the cost of providing receipts (Section 3.6.1), its scalability (Section 3.6.2), the overheads of receipt validation (Section 3.6.3), and auditing (Section 3.6.5). We finish with a performance breakdown of IA-CCF's design features (Section 3.6.8).

Testbeds. Our experimental setup consists of three environments: (a) a dedicated cluster with 16 machines, each with an 8-core 3.7-Ghz Intel E-2288G CPU with 16 GB of RAM and a 40 Gbps network with full bi-section bandwidth; (b) a LAN environment in the Azure cloud,

Table 3.1: Size of ledger entries (SmallBank)

Ledger entry type	Size (bytes) f = 1	f = 3
Transaction (SmallBank)	216–358	
Pre-prepare	277	
Prepare Evidence	298	995
Nonces	32	64

with Fsv2-series VMs with 16-core 2.7-GHz Intel Xeon 8168 CPUs and 7 Gbps network links; and (c) a WAN environment with the same VMs across 3 Azure regions (US East, US West 2, US South Central). All machines run Ubuntu Linux 18.04.4 LTS.

Implementation. Our IA-CCF prototype is based on CCF v0.13.2 [169] and has approximately 40,000 lines of C++ code. It uses the formally-verified Merkle trees and SHA functions of EverCrypt [197], the MbedTLS library [161] for client connections, and secp256k1 [250] for all secure signatures. Replicas create secure communication channels using a Diffie–Hellman key exchange.

Pipelining batch execution (P in Algorithm 3.1) improves IA-CCF's throughput. We use P=2 for the LAN and P=6 for the WAN, with maximum batch sizes of 300 and 800 requests, respectively. Checkpoints are created every 10K or 4K sequence numbers in the LAN and WAN environments, respectively.

Benchmarks. We use the *SmallBank* benchmark [7], which models a bank with 500,000 customer accounts. Clients randomly execute 5 transaction types: deposit, transfer, and withdraw funds; check account balances; and amalgamate accounts. The size of the ledger entries is shown in Table 3.1 where only the Prepare Evidence and Nonces entries depend on f.

Since IA-CCF's design targets accountability with more than f failures, we omit results from experiments with fewer failures. In such cases, IA-CCF's performance matches that of prior work, because it uses well-established BFT techniques, such as view changes, sending messages via authenticated channels and client-signed requests [31, 56]. Instead, we consider the performance of receipt validation (Section 3.6.3) and auditing (Section 3.6.5), which are new contributions of IA-CCF.

Transaction throughput is measured at the primary replica and latency at the clients. All experiments are compute-bound. Results are averaged over 5 runs, with min/max error bars.

Baselines. We compare against four baselines: IA-CCF-PeerReview, which uses PeerReview for accountability [111], i.e., replicas sign all messages and send signed acknowledgements for all messages; IA-CCF-NoReceipt, an IA-CCF variant that produces a ledger but no receipts; HotStuff [253], a state-of-the-art BFT protocol, which is at the core of the Diem permissioned

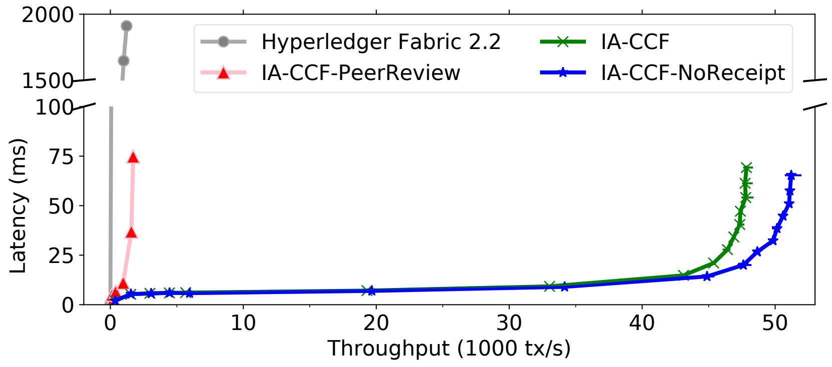


Figure 3.4: Transaction throughput/latency (f=1, dedicated cluster)

Table 3.2: Request latency under low load (WAN)

	average latency	99th percentile latency	network round trips
IA-CCF	183 ms	194 ms	2
HotStuff	340 ms	393 ms	4.5

ledger system [10]; and Hyperledger Fabric (v. 2.2) [11], a popular open-source permissioned ledger system. We compare against Fabric's latest major release that does not include a BFT consensus protocol [88] and only tolerates crash failures using Raft [187].

3.6.1 Transaction throughput and latency

We explore the throughput and latency of transaction execution with 4 replicas (f=1) in the dedicated cluster, comparing IA-CCF, IA-CCF-NoReceipt, IA-CCF-PeerReview, and Fabric.

Figure 3.4 shows a throughput/latency plot as transaction load increases. IA-CCF achieves 47,841 tx/s while maintaining latencies below 70 ms. As the load increases, queueing delays increase latency. IA-CCF-NoReceipt's throughput is 51,209 tx/s, which is only 3% higher than IA-CCF, demonstrating the low cost of receipts.

IA-CCF-PeerReview exhibits an order of magnitude lower throughput because all messages must be signed, e.g., a replica must sign a reply message for each transaction in a batch. This causes IA-CCF-PeerReview to perform two orders of magnitude more asymmetric cryptographic operations than IA-CCF.

Fabric's throughput is only 1,222 tx/s, with a latency of 1.9 s. This is substantially worse than IA-CCF, despite not using a BFT protocol. Our analysis reveals two reasons: Fabric's *execute-order-validate* model requires that replicas issue a signature for each executed transaction, while IA-CCF replicas only require one signature per batch; and Fabric suffers from documented inefficiencies related to its key-value store implementation [177].

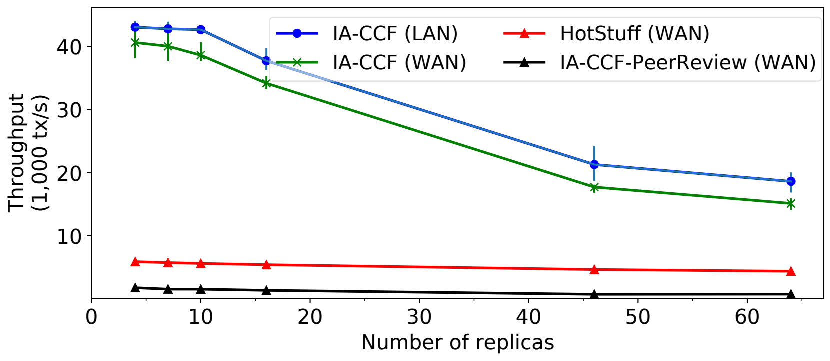


Figure 3.5: Transaction throughput vs. replica count (WAN)

3.6.2 Scalability

Next we consider the effect on transaction throughput when increasing the number of IA-CCF replicas in the Azure WAN environment, spanning multiple regions to reduce correlated failures [25]. We compare against IA-CCF deployed in the Azure LAN environment, IA-CCF-PeerReview, and HotStuff, a BFT consensus protocol without a ledger or key-value store.

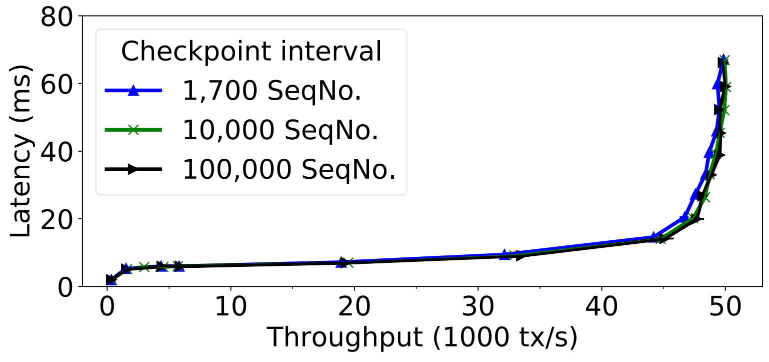
Figure 3.5 shows that, as expected, IA-CCF's throughput decreases with more replicas because more signatures are verified by each replica. Since each replica has a fixed number of threads for checking signed pre-prepare/prepare messages in parallel, throughput decreases when the replica count exceeds the number of hardware threads, which is only 16 in this deployment. IA-CCF is only marginally affected by the higher WAN latencies due to its use of pipelining, as shown by the comparison to the LAN deployment.

HotStuff [255] achieves a throughput of 5,862 tx/s in the WAN environment, which is worse than its reported LAN throughput [261]. While it degrades slowly with more replicas, even with 64 replicas its throughput remains 71% lower than that of IA-CCF. The throughput of IA-CCF-PeerReview is even lower since it performs more cryptographic operations.

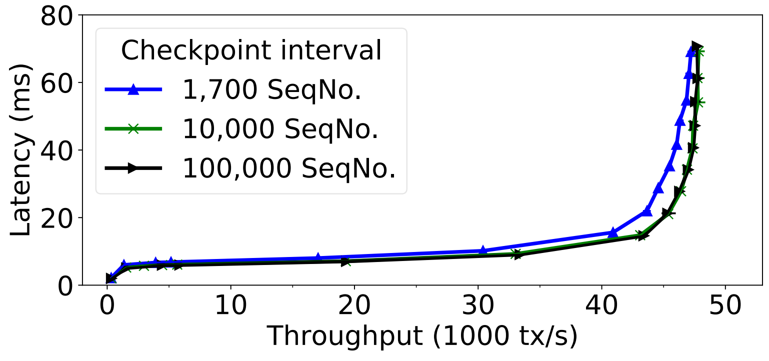
We also measure the request latency of HotStuff and IA-CCF under low load. As reported in Table 3.2, HotStuff's request latency is approximately twice that of IA-CCF's. For both systems, request latency is dominated by the number of network round trips and clients receive transaction results with receipts in only two round trips in IA-CCF.

3.6.3 Receipt validation

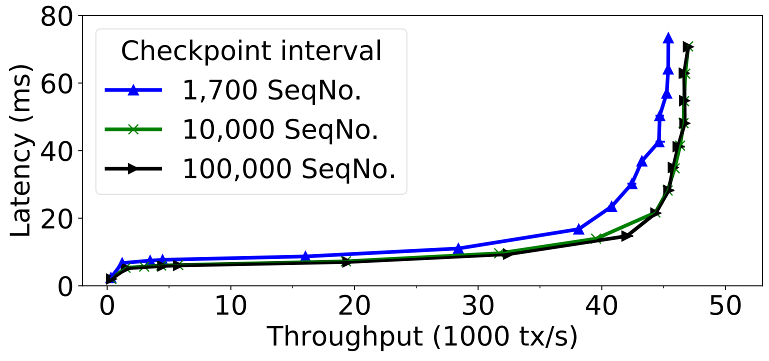
We measure the time required to verify receipts, which depends on (i) the length of the path in the Merkle tree G and (ii) the number of signatures to be checked. Since the number of leaves in G is bounded by the batch size, the path length remains small: verification takes 2.1 µs and 2.3 µs for batches of 300 and 800 requests, respectively. The overall cost is dominated by the signature verification, which takes 18 ms and 52 ms for f=1 and f=3, respectively.



(a) 100,000 accounts



(b) 500,000 accounts



(c) 1,000,000 accounts

Figure 3.6: Transaction throughput/latency when varying the number of accounts and checkpoint interval (f=1, dedicated cluster)

3.6.4 Governance sub-ledger

Next, we consider the size of the governance sub-ledger, which is stored by clients. The sub-ledger is a collection of receipts for every transaction that has updated the governance of an IA-CCF deployment. A receipt's size is 623 bytes or 1,565 bytes for f=1 or f=3, respectively. In addition, the client must store the governance request and the corresponding response, which have variable size. We expect governance operations to be rare. Therefore, storing and verifying governance sub-ledger receipts has low overhead.

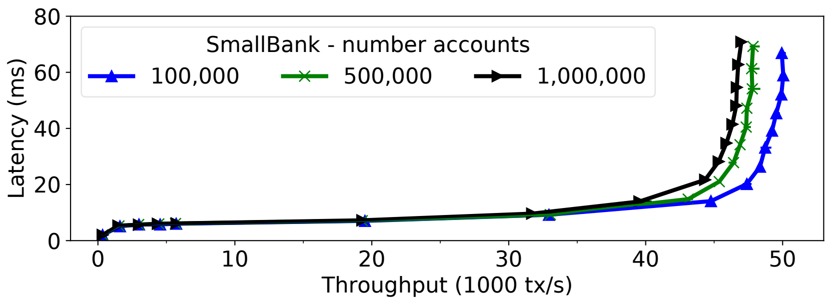


Figure 3.7: Transaction throughput/latency with different account numbers (f=1, dedicated cluster)

3.6.5 Ledger auditing

We want to understand auditing performance. For the SmallBank workload, we compare execution time to auditing time. When measuring throughput at f=1, auditing is 23% faster than execution, because there is no network overhead, message signing, or ledger writes. In each batch, IA-CCF only verifies $2f$+1 rather than up to $3f$+1 signatures. For f=4, the performance gap increases to 67%, as more replicas add communication and cryptographic load during execution. We observe that the bottleneck for auditing is verification of client request signatures, which can be trivially parallelized.

3.6.6 Key-value store

We explore the performance impact of varying the number of entries in the key-value store by varying the number of SmallBank accounts. Figure 3.7 shows a throughput vs. latency plot. As expected, throughput decreases when the number of entries in the key-value store increases. CCF's implementation [206] of the key-value store uses a CHAMP map [224], whose access time grows logarithmically with the number of items.

3.6.7 Checkpointing

We also explore the effect of checkpointing on performance. We vary the size of the key-value store and the checkpoint interval for the SmallBank workload. Figure 3.6 shows the results as throughput vs. latency plots. As expected, the checkpoint overhead increases with the size of the key-value store and the checkpoint frequency, but the overhead is low for checkpoint intervals between 10 and 100,000 (approximately 1 to 10 minutes). The checkpoint interval impacts the overhead to check uPoMs at the enforcer. We expect checkpointing every 10 minutes to be acceptable in practice; it requires the enforcer to replay at most 10 minutes of transactions.

Table 3.3: Breakdown of IA-CCF features (f=1, dedicated cluster)

	Variant	Throughput (tx/s)
(a)	Full IA-CCF	47,841
(b)	IA-CCF-NoReceipt	51,209
(c)	+ without checkpoints	51,288
(d)	+ small key-value store	53,759
(e)	+ without signed client requests	111,926
(f)	+ with MACs only	128,921
(g)	+ without ledger	131,959
(h)	+ with empty requests	299,321
	HotStuff (with empty requests)	307,997
	Pompē (with empty requests)	465,646

3.6.8 Overhead breakdown

To provide a permissioned ledger with individual accountability, IA-CCF implements functionality that goes beyond traditional BFT consensus protocols, e.g., generating receipts. We now explore the impact of implementing this functionality on IA-CCF's throughput in the dedicated cluster.

We compare several variants of IA-CCF, each limiting functionality further: (a) IA-CCF; (b) IA-CCF-NoReceipt, i.e., without creating receipts; (c) without creating checkpoints; (d) with a small key-value store, i.e., the key-value store fits in the CPU cache; (e) without signed client requests; (f) using only MACs for message authentication between replicas; (g) without a ledger; and (h) with empty requests, i.e., without the overhead of executing transactions against the key-value store.

Table 3.3 shows that (a)–(d) have comparable throughput, but not verifying client signatures (e) doubles throughput. Only using MACs instead of signatures (f) or removing the ledger altogether (g) does not increase throughput substantially, but removing the overhead of executing transactions against the key-value store (h) again doubles throughput.

For context, we compare with two Byzantine consensus protocols with similar functionality to (h) above, HotStuff [253] and Pompē [262, 261]. HotStuff's throughput is 307,997 tx/s, but with higher latency (Section 3.6.2). By separating request ordering and consensus, Pompē achieves a throughput of 465,646 tx/s, also with worse latency (IA-CCF's 12 ms to Pompē's 73 ms). IA-CCF could utilize Pompē's techniques for increased throughput by sacrificing its two round-trip latency.

These breakdown results show that IA-CCF's overhead comes primarily from the cryptographic operations required for verifying client requests, followed by the transactional key-value store, rather than the consensus protocol or the mechanisms specific to providing individual accountability.

3.7 Related work

Permissioned ledgers. Many permissioned ledger systems [124, 200, 10, 11] rely on BFT consensus protocols to order transactions. Hyperledger Besu [124] and Quorum [200] use variants of PBFT [175, 210], which do not retain proof of a replica's operations, and therefore cannot assign blame. Diem [10] uses the DiemBFT [27] consensus protocol, which is based on HotStuff [253] and also lacks accountability features.

The IA-CCF prototype is built on top of CCF [206], an open source [168] distributed ledger framework deployed in the Azure cloud [167], which utilises trusted execution environments (TEEs) [61, 134] to harden replicas. Russinovich et al. [206] describe CCF's programming model, receipts, governance, and replication protocols. While CCF enables auditing and can recover a ledger when all replicas crash, it relies on the security of TEEs, and its auditing does not guarantee individual accountability.

Byzantine consensus [49, 56, 141] distributes trust. Recent work on BFT protocols has focused on improving guarantees [63, 171, 15] or performance for particular use cases [223, 262]. SBFT [107] and HotStuff [253] scale to hundreds of replicas using threshold cryptography, which prevents blame assignment. For permissioned ledgers, scaling to many replicas without growing the consortium size does not improve trustworthiness, and consortia typically cannot grow arbitrarily.

Other work has explored misbehaviour and its impact on Byzantine consensus. BFT2F [153] formalizes safety and liveness guarantees after more than f replicas are compromised. It provides PBFT's guarantees with up to f failures and provides *fork** consistency with up to $2f$ failures. For permissioned ledgers, *fork** consistency is not sufficient, because it is susceptible to double-spending attacks.

Depot [159] issues proofs-of-misbehaviour after observing misbehaviour, but it adopts eventual consistency, which is incompatible with permissioned ledgers. Pompē [262] prevents dishonest primaries from controlling the ordering of requests. It does not address scenarios in which there are more than f dishonest replicas though.

Accountability. PeerReview [111] ensures that distributed nodes remain accountable for their actions. As shown in Section 3.6.1, PeerReview incurs a high overhead when applied to a permissioned ledger. In contrast, IA-CCF introduces mechanisms specific to BFT state machine replication, such as a shared ledger with a Merkle tree, to improve both regular transaction execution and auditing.

Accountable virtual machines [110] carries out auditing through *spot checking* of checkpoints, but has the same performance overheads as PeerReview for ledgers. SNP [264] is a networking-specific implementation of accountability, offering provenance for routing decisions.

Such specializations improve performance in particular domains, but are not directly applicable to permissioned ledgers.

BAR [5] and Prosecutor [259] incentivize replicas to act honestly by having honest replicas penalize misbehaviour. This weaker model allows BAR to tolerate more than 1/3 faulty replicas, while Prosecutor uses these incentives to improve performance. If these incentives fail [122], however, replicas share the blame.

Accountability with more than $f+1$ misbehaving replicas has been discussed before [39, 41, 117]. BFT Protocol Forensics [219] and Polygraph [55] propose a ledger auditing mechanism, but assume that fewer than $N-f$ replicas misbehave. They also do not support changing replica sets. ZLB [201] and Tendermint [39] support changes to the replica set but also assume that fewer than $N-f$ replicas misbehave.

3.8 Summary

This chapter presented IA-CCF, the first permissioned ledger that provides individual accountability and is not subject to an arbitrary failure limit after that the permissioned ledger does not provide accountability.

We first presented a definition of individual accountability describing how individual accountability differs from accountability.

Next, we described IA-CCF. First, we presented an overview of IA-CCF and the major components of the permissioned ledger. Next, we described L-PBFT, the Byzantine fault tolerant consensus protocol of IA-CCF, including the protocol messages sent by the consensus protocol when ordering requests and changing the primary replica. This explanation detailed how IA-CCF's ledger is created and how receipts are sent to clients. Finally, we provided a correctness argument for the linearizability of L-PBFT.

Next, we explored auditing the permissioned ledger. We explored how an IA-CCF auditor can find misbehaviour in an IA-CCF ledger, and create a concise uPoM, that is then passed to an enforcer so they can apply any relevant penalties. Next, we introduced IA-CCF's governance protocol that changes the active replica set and showed how during reconfiguration IA-CCF maintains its individual accountability property. Finally, we presented a proof showing that an auditor can always find misbehaviour within an instance of the IA-CCF permissioned ledger.

Finally, we looked at the performance of our IA-CCF prototype. We first examined the throughput and latency of IA-CCF in both a LAN and WAN environments. Next, we focused on understanding how the components of IA-CCF affect the permissioned ledger's overall performance.

CHAPTER 4

GPU Acceleration with Permissioned Ledgers

This chapter presents *DropBear*, the first cloud-based machine learning (ML) inference service that provides clients with strong integrity guarantees, while staying compatible with current inference APIs. DropBear replicates model updates and inference computation among GPU nodes that belong to different cloud providers and provides clients with *inference certificates* that prove agreement for specific model versions.

In the context of DropBear, we look at how hardware accelerators can be used efficiently within a permissioned ledger that utilises a Byzantine fault tolerant consensus protocol. We design the DropBear permissioned ledger such that it reaches agreement even when there are honest but differing inference results due to heterogeneity of contributing models or differences in the hardware and software on each node. To improve performance, DropBear batches inference and consensus operations independently: it first performs the inference computation across an ensemble of models and reaches agreement, before ordering requests and model updates.

This chapter further contributes to addressing concerns of organizations that utilize permissioned ledgers – see section 1.4.3 – by exploring performant execution of ML workloads with accelerators on a permissioned ledger that combine ML and CPU based workloads.

4.1 Introduction

In the previous chapter, we considered the design and evaluation of a permissioned ledger that is auditable regardless of how many replicas are dishonest. We showed that, by co-designing the ledger, Byzantine consensus protocol, and receipts it is possible to make a ledger that provides individual accountability and order a large number of transaction requests. However, a limitation of the previous work is that we only consider the ordering and execution of simple transaction requests, we did not consider the execution of complex transactions.

In this chapter, we explore how permissioned ledgers with a Byzantine consensus protocol can be used to execute complex transaction requests. Specifically, we look at how a permissioned ledger can utilize highly parallel accelerators (GPUs) to build machine learning (ML) inference applications which require the execution of long running inference requests (≥ 1 ms).

ML inference, particularly those obtaining inference decisions from a deep neural networks (DNNs), used by a large number of applications, and this number is only expected to grow [92, 181, 144, 136]. The resource-intensive nature of DNN inference computation has led to cloud providers offering cloud-based ML inference services, such as Azure Machine Learning [166], AWS AI Services [9], and Google Cloud Inference [100]. These services expose simple APIs that allow users to upload a single ML model, or an *ensemble* of models, which combines inference results from multiple models to reach higher accuracy [208], to be hosted on the provider's infrastructure. Clients then submit inference requests over these models, which are executed by cloud nodes with GPUs or other hardware accelerators.

While cloud-based inference services are popular, they require owners of ML models and clients submitting inference requests to trust the cloud provider. In many applications that use ML inference for decision-making, this affects the trustworthiness of the application. For example, ML inference is used by the insurance industry to decide if an insurance application is underwritten [113, 4, 160], or in contentious domains, such as, by courts to aid in sentencing within the criminal justice system [59, 119, 146] where the reasoning for selecting a specific ML model that contributes to a sentencing must be recorded. A malicious cloud provider or a rogue employee may tamper with inference computation to save on computational resources (resulting in lost accuracy) or deliberately change the outcome from a downstream application that depends on inference results.

It is challenging for clients to validate the correctness of inference results. The numeric values of inference results depend on the used models versions and the employed hardware and software, e.g., the type of GPUs [247], their floating-point accuracy and implementation [70], and other optimizations in the inference software stack [52, 66, 204, 207]. Moreover, if model owners use an ensemble of DNN models, correctness also depends on the service correctly aggregating results [246, 79, 26]. In practice, if clients want to establish correctness, they must redo the inference computation and reproduce the result in independent failure domains, defeating the purpose of executing inference requests within a single cloud.

Cloud deployments of ML inference services are often motivated by performance and scalability requirements. Typical applications such as fraud detection routinely handle hundreds of transactions per seconds, and may experience large peaks or periodic usage increases during business hours. Dynamic cloud scaling avoids the ineffective over-provisioning of dedicated resources. The security properties of IA-CCF would provide an ideal solution to the issue encountered by clients want a trustworthy inference decision, however, the system's focus on ordering simple transaction request would result in poor performance (see Section 4.7.3).

In this chapter we describe **DropBear**, a cloud-based permissioned ledger that offers trustworthy inference decisions over DNN models. DropBear allows model owners to upload ensembles of models that consists of one or more DNN models, and clients then submit inference requests over these ensembles. The ledger allows for combining different types of transactions – executed on the CPU and GPU – enabling a ledger to contain the transaction that resulted in a specific model being loaded and used to execute ML workloads. DropBear makes the following contributions:

(1) DropBear realizes a **trustworthy inference service** in which inference results do not depend on the correct execution of any individual cloud node. The key idea is to replicate model updates and the inference computation across multiple nodes in different clouds, which must reach agreement on inference results. To support ensemble training, agreement is defined subject to acceptable bounds on the results when nodes execute different models from the ensemble and combine the results. This limits possible divergence if a dishonest node provides incorrect results. Consensus among the nodes regarding model versions and inference results is reached using a Byzantine agreement protocol (L-PBFT). Allowing model versions to be totally ordered with other transactions that resulted in the specific version of a model being selected.

(2) DropBear follows an **execute-agree-attest** strategy that separates the expensive execution of inference requests from the agreement and attestation of inference results. Inference requests are first *executed* in batches by geo-distributed cloud nodes against an ensemble of models. After execution, a primary node batches the inference requests again, orders them with respect to model updates, and sends batches to a set of backup nodes. The nodes *agree* with the one-another's inference result subject to the bounds associated with the ensemble. After 2/3 of nodes have agreed, they *attest* that the result is within the prescribed bounds. Recording the results in the ledger allows for follow-up transactions – some of which may not be ML based – to be linearized with the inference execution.

Through this separation, DropBear can batch inference requests and consensus operations independently, allowing for a higher degree of parallelism across nodes: batching inference

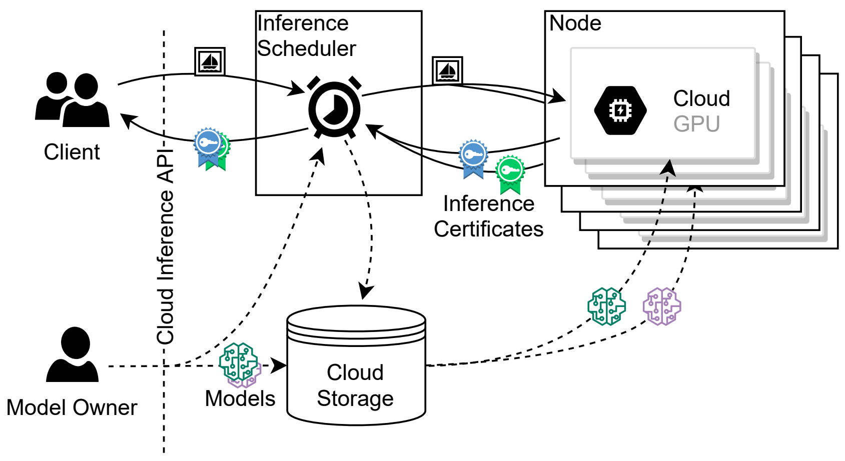


Figure 4.1: Cloud-based ML inference service. A client sends an inference request of a sailboat and receives an inference result, which was executed on the cloud nodes. It also shows a model owner who uploads an ML model to cloud storage and then the inference scheduler directs which nodes load the new model before it is sent to the nodes.

requests for heavily referenced models [131] into *execution batches* increases GPU utilisation, and execution batches can span multiple *consensus batches*; consensus batches are constructed to better utilize the wide-area network between cloud nodes and reduce the per-batch CPU overhead of cryptographic operations.

(3) Despite using multiple cloud nodes to reach consensus about inference results, DropBear maintains an inference API with a single endpoint that remains compatible with current cloud-based APIs [166, 8, 100]. Clients receive inference results together with an **inference certificate**, which is a universally-verifiable proof that sufficient nodes executed the inference request and agreed on the specific result for a given version of an ensemble of models. The inference certificate is created by an *inference proxy*, which combines cryptographically-signed Merkle trees of inference results and consensus messages.

Note that the inference proxy need not be trusted: clients can easily check the inference certificate's correctness by verifying that the content of the inference certificate is signed by the service and the certificate contains the client's inference request.

4.2 Trust in cloud-based machine learning inference

Next we introduce cloud-based ML inference services (Section 4.2.1), describe their security challenges (Section 4.2.2) and introduce our threat model (Section 4.2.3).

4.2.1 Cloud-based machine learning inference services

All major cloud providers offer ML inference services, e.g., Azure Machine Learning [166], AWS AI and ML services [9], and Google Cloud Inference [100]. They host trained ML models and expose APIs through which clients can submit inference requests. Organizations move to cloud-hosted ML inference for a multitude of reasons, including cost savings, access to the latest GPU hardware, and fast scale out for inference workloads [45, 9].

Many of these models are trained using deep learning, which is an unsupervised learning technique that creates deep neural network (DNN) models [98]. Such models have shown great success in diverse areas, image detection and classification, natural language processing, and fraud detection. Inference computation over DNN models is resource-intensive as it requires performing a large number of floating point operations, many of which can run in parallel. Therefore, recent inference systems, such as NVIDIA Triton Inference Server [236], ONNX Runtime [76], and TensorFlow Server [232], execute inference requests on GPUs [1], TPUs [131], or multi-core CPUs, selecting the hardware for which the DNN model was optimized.

In many application domains, the accuracy of DNN inference results can be enhanced by using ensemble learning [208], i.e., combining the results of several trained models to produce a new result with better accuracy than of the input models. Ensemble learning has been shown to avoid overfitting, allows models to be trained on more narrow datasets, and reduces the impact of individual models making incorrect inference decisions [78, 195, 208].

Figure 4.1 gives an overview of a cloud-based ML inference service [137]. The service exposes an API that is used by *model owners* and *clients*: model owners upload one or more models, which may form an ensemble, and are typically expressed in a standardized format, such as ONNX [20] or NNEF [73]. The models are then deployed on *cloud nodes* with hardware accelerators such as GPUs. The nodes execute inference requests submitted by clients using an *inference engine* implementation.

Clients must be authorized to use the service, and they submit inference requests through an application protocol, e.g., HTTP. An inference request specifies the ensemble model to use and the input data for the inference, e.g., an image. The inference request is accepted by an *inference scheduler*, which forwards the request to a cloud node for execution. For an ensemble of models, the scheduler collects the results from multiple nodes, applies an ensemble learning technique to produce an aggregated result, and returns it to the client.

4.2.2 Trust requirements

A client's confidence that an inference request was executed correctly is crucial when using ML inference in applications. This confidence can be gained by recording the reason for using an ML model and the inference results on the same ledger. This applies to many application domains:

Electronic trading. ML methods have been used by banks and hedge funds as part of automated trading strategies [19, 13], and in new types of FinTech scenarios [44, 263, 240]. There are several concerns when using ML inference here: (i) as the trading strategy depends on the inference results, results must not be tampered with by rogue employees e.g., for financial gain; and (ii) government regulators require financial firms to maintain compliance records. When ML inference is used, a trustworthy record of the inference requests is necessary and the reason for selecting a specific model must be selected.

Medical diagnosis. An emerging area for ML inference is the image-based diagnosis of medical conditions, e.g., cancer, using ML models [228, 150, 222]. Given the high accuracy demands in this domain [154, 157, 244], diagnoses from a single model are often insufficient, requiring results from ensembles that combine models with different strengths [216, 192, 14]. In addition, ML-based diagnosis must be accountable: an organization that creates a diagnosis shown to be incorrect would be incentivized to disavow the incorrect results to protect against reputational, financial, or criminal claims. Finally, a patients diagnosis and treatment should be recorded on a ledger to show that a ML created diagnosis was property considered.

Government services increasingly rely on ML models as part of citizen-facing services. For example, the US Internal Revenue Service (IRS) uses ML models to combat tax evasion by discovering discrepancies in tax returns [90, 28]. Using automated AI methods has proven problematic though—court rulings stated that such methods may be "insufficiently clear and verifiable" [182]. While government organizations benefit from the cost savings of cloud-based services, they must demonstrate trustworthy execution. Retaining why the inference result is utilised demonstrates to courts that decisions taken based on ML inference have not been tampered with [58].

4.2.3 Threat model

In our threat model, model owners and clients do not trust all nodes that constitute the ML inference service: a client expects that an inference result sent by any single node may be incorrect. The client, however, trusts the model owner and, by extension, that the originally provided models are correct.

An attacker's goal is to tamper with inference results without clients detecting the malicious behaviour. The attacker may be a rogue employee of a cloud provider, who already has physical or remote access to the nodes, or external to the cloud provider, after having penetrated its defenses. We assume that the adversary can compromise at a fraction of the cloud nodes, despite physical or software protections within a cloud environment. After compromising the nodes, they can direct and coordinate the nodes' behaviour.

Nodes may be distributed across multiple independent cloud environments, e.g., multiple cloud providers or geographically distinct locations that constitute independent failure domains. Each node has its own identity, e.g., a public and private cryptography key pair, and the attacker is unable to obtain an honest node's private key. Nodes can establish secure communication channels, which cannot be tampered if both nodes are honest. We assume that clients and model owners have also verifiable identities [194, 189]. The inference service can authenticate them and create secure channels.

4.3 Trustworthy machine learning inference

We discuss how to establish trustworthiness in an ML inference service (Section 4.3.1) and formalize the definition of a trustworthy service based on agreement (Section 4.3.2).

4.3.1 Establishing trustworthiness

Intuitively, a trustworthy inference result is a result that a client has obtained by correctly executing an inference request against an ML model. Clients can obtain trustworthy inference results in two ways: (1) a result obtained from a trusted node, which executes the inference computation correctly; and (2) a collection of results from multiple nodes, with the client trusting the collective *agreement* across the results.

Trusted node. When a trustworthy node executes the inference request, the node must demonstrate its trustworthiness to the client. This can be achieved when the client controls the node, which is not possible in a cloud environment.

Alternatively, the inference computation can be protected by the hardware through a *trusted execution environment* (TEE) [242, 61]. A TEE leverages a root of trust from the hardware to shield the execution of sensitive computation and its data from the rest of the environment, including higher privileged software layers. While TEEs can be used to make ML computation trustworthy [102, 179, 180, 151], past TEE implementations have been shown to exhibit exploitable security vulnerabilities [40, 176, 239, 183].

Trusted agreement. If a single node cannot be trusted, a trustworthy service can be constructed from a collection of nodes that collectively agree on an inference result. As long as a sufficiently large quorum of trustworthy nodes have agreed, the agreed results can be considered trustworthy.

Byzantine consensus protocols such as PBFT [49] obtain agreement on an execution result from a set of nodes, even if some nodes are malicious. A trustworthy ML service, that maintains a ledger, could thus rely on the agreement of a set of nodes, distributed across multiple cloud providers, using Byzantine consensus. No individual untrustworthy cloud provider would then be able to convince the client to accept an incorrect inference result.

4.3.2 Trustworthy inference through agreement

We formalize the problem of providing a trustworthy ML inference result as follows. We assume that a client submits an inference request against a set of models $m_1 \dots m_k$ that form an ensemble $m \in \mathcal{E}$. The inference request r is served by a distributed ML inference service with nodes $\mathcal{N}$. For a given model $m \in \mathcal{E}$ and a node $i \in \mathcal{N}$, let $q_i^m : \mathbb{R}^{u_m} \mapsto \mathbb{R}^{v_m}$ denote the implementation of the inference computation at node i for model m, where input and output tensors are encoded into flattened arrays. (We require the input and output dimensions u_m and v_m to be the same at every node.)

The client obtains a set of inference results from $N = |\mathcal{N}|$ nodes, of which at most f nodes may return an untrustworthy result. Therefore the client can construct a trustworthy inference result as long as it obtains at least $f+1$ results that the $f+1$ nodes have agreed on, when this quorum of nodes includes at least one trustworthy node. If there are less trustworthy nodes, ledger auditing ensures that only untrustworthy nodes are blamed for their misbehaviour.

To define which inference results are in agreement with each other, we introduce two parameters: (i) a model-specific metric $\delta_m : \mathbb{R}^{v_m} \mapsto \mathbb{R}$ that represents a meaningful distance between two results; and (ii) a similarity threshold ϵ_m that determines which results are sufficiently close to be considered in agreement. Both are provided by the trusted model owner when submitting $\mathcal{E}$; a client may decide to select a different similarity threshold ϵ_m to limit the maximum influence of incorrect or malicious nodes according to their own preferences.

As a default, the model-specific metric δ_m can be defined to be Euclidean distance $\delta_m(x, y) = \sqrt{\sum_{i=1}^{v_m}(x_i - y_i)^2}$. For simple models, such as classifiers that return a vector of probabilities for all possible classes, this measure suffices to exclude misclassifications; for more complex models, such as ones that return higher dimensional results, it is more appropriate to use a different distance measure such as Hausdorff [123] or complex wavelet similarity [243], which are more robust to semantically irrelevant differences.

The distance metric δ_m, together with the threshold ϵ_m, can now be used to define the inference results that are in agreement. We denote $2^{\mathcal{N}}$ the powerset of $\mathcal{N}$, i.e., the set of all subsets of $\mathcal{N}$, and $[2^{\mathcal{N}}]_f = \{C \in 2^{\mathcal{N}} \mid |C| \geq N - f\}$ its restriction to subsets with at most f nodes removed. Given a request $r \in \mathbb{R}^{u_m}$ and subset $\mathcal{X} \in 2^{\mathcal{N}}$, we also define the diameter $\Delta_m(r, \mathcal{X})$ of the inference result set of X as:

$$\Delta_m(r, \mathcal{X}) = \max_{i,j \in \mathcal{X}} \delta_m \left(q_i^m(r), q_j^m(r) \right)$$

The set of results from trustworthy nodes within the threshold ϵ_m is:

$$Q_m(r) = \arg\max_{\mathcal{X} \in [2^{\mathcal{N}}]_f} \{|\mathcal{X}| \mid \Delta_m(r, \mathcal{X}) \leq \epsilon_m\}$$

API function	**Description**
`get_API_endpoints``()` $\rightarrow$ `([API_URL], [pub_key])`	Obtains API endpoint and public node keys from trusted discovery service.
`load_ensemble``(ensemble, [model_URL], dist_fn, acl)`	Uploads `models` that define `ensemble` with access control list and distance function.
`activate_ensemble``(ensemble)`	Activates ensemble.
`retire_ensemble``(ensemble)`	Retires ensemble.
`request_inf``(ensemble, input,` ϵ`)` $\rightarrow$ `([inf_res], inf_cert,` dist_fn`)`	Executes request against `ensemble`; return results, certificate, and distance function.
`verify_cert``(inf_req, [inf_res], inf_cert, [pub_key])` $\rightarrow$ `bool`	Verifies validity of inference certificate.

Table 4.1: DropBear API

This definition selects the largest subset of at least $N - f$ nodes whose diameter is within the threshold, rather than the one with minimal diameter overall (which may unnecessarily exclude legitimate results). While $f + 1$ nodes may be enough for agreement, it would make the definition more brittle, as there could be two sets of equal size (greater than $f + 1$) of diameter smaller than ϵ_m if all malicious nodes skew their result towards a honest outlier. $Q_m(r)$ may also fail to yield a trustworthy subset if honest nodes are too inconsistent and the client may consider whether a larger ϵ_m is acceptable.

Note that $Q_m(r)$ only contains the set of nodes that produce a trustworthy result with respect to δ_m and ϵ_m. In addition, clients may also expect the service to return a single aggregate inference result based on $\{q_i^m(r), i \in Q_m(r)\}$, e.g., using an average or computed δ_m-center approximations.

4.4 DropBear API

The API in Table 4.1 shows the interface through which *model owners* and *clients* interact with an instance of DropBear. It supports the management of ensembles of models and the execution of trustworthy inference requests.

Both model owners and clients must obtain an API endpoint to communicate with a DropBear deployment. The `get_API_endpoints` call contacts a trusted discovery service at a well-known location and returns a list of URL endpoints (`API_URL`) together with their public keys. These identify the set of N nodes, of which up to f may not be trustworthy, that are distributed across multiple cloud providers and constitute the DropBear service. API requests can be sent

```
1 API_URLs, pub_keys = get_API_endpoint()
2 dist_fn = lambda results,dist=0.2:max(filter(lambda set:max(set
   )-min(set)<dist,powerset(results)), key=len)
3 model_URLs = ["https://[...]/resnet101.onnx", ...]
4 load_ensemble("ResNet", model_URLs, [c0, c1], dist_fn)
5 activate_ensemble("ResNet")
6 ...
7 [inf_res], inf_cert, _ = request_inf("ResNet", data)
8 verify_cert(inf_req, [inf_res], inf_cert, pub_keys)
9 ...
10 retire_ensemble("ResNet")
```

Listing 4.1: Example use of DropBear API

to any of the returned URL endpoints.

Model owners provide and maintain the ensembles of models, which includes a distance function that removes potentially dishonest inference results from the ensemble results. They add or update ensembles using the `load_ensemble` call, which includes an `ensemble` name, a list of URLs (`[model_URL]`) that store the models in the ensemble in ONNX format [20], an access control list (`acl`) with clients that may issue inference requests, and a distance function (`dist_fn`). The distance function takes a set of inference results and returns those which it deems to be trustworthy (see $Q_m(r)$ in Section 4.3.2).

Before clients can issue inference requests against a new ensemble, the model owner must activate it – to allow time for the model to be loaded from cloud storage into GPU memory – with the `activate_ensemble` call; it is deactivated using the `retire_ensemble` call. Listing 4.1 gives an example how a model owner loads an ensemble called `ResNet`, accessible by two clients (lines 2–4). The ensemble uses a distance function that returns the largest set of results where all results are within a threshold of 0.2 (line 2). The model owner activates the ensemble (line 5) and eventually retires it (line 10).

The distance function (line 2) assumes a classification model that returns a confidence score for each label. A distance function for a top-1 accuracy classifier for the ImageNet models, whose inference results return confidence scores for 1000 labels, would return the group of inference results whose Euclidean distance is 0.2 of each other; an agreement function for a medical image classification request would return inference results whose Hausdorff distance [156] is within 0.2.

Clients issue inference requests against an ensemble using the `request_inf` call (line 7) and can pass a similarity threshold ϵ to the distance function (see ϵ_m in Section 4.3.2); if ϵ is left blank, the default specified by the model owner is used. After executing the inference request, DropBear returns multiple inference results (`[inf_res]`) and an *inference certificate* `inf_cert`. The inference results are the output from nodes that executed the inference request. The inference certificate provides a proof that multiple nodes agreed on the outcome of the inference execution, and the client must verify it together with the results by calling `verify_cert` (line 8). If the client believes that the results it received are dishonest, the client triggers an audit.

The function `verify_cert` checks the signatures of nodes over the inference results and their signatures over the results from other nodes (see Section 4.5.4). A node's signature over another node's result shows their belief that the other node's result is trustworthy, which we refer to as an *attestation*. The function first checks that at least $N-f$ nodes returned inference results. It then validates that each inference result is signed by the node that produced it. For every signed result, the function checks that at least $f+1$ attestations with valid signatures exist. This ensures that every result is attested by at least one trustworthy node, i.e., more than f nodes attested the results.

4.5 DropBear design

Next we describe DropBear's design (Section 4.5.1) and explain how it provides trustworthy inference results using its execute (Section 4.5.2), agree (Section 4.5.3), and attest (Section 4.5.4) strategy. We finish with a discussion (Section 4.5.5).

4.5.1 Overview

As shown in Figure 4.2, a DropBear deployment consists of N nodes, which are distributed across multiple cloud providers. To provide a trustworthy ML inference service (see Section 4.3.2), each node manages models from ensembles and *executes* inference requests against them. The results provided by different nodes must then *agree* with each other. Agreement can only be achieved if inference requests are *ordered* consistently with respect to model updates, otherwise nodes may execute inference requests against different versions of the same model. Finally, the agreement must be *attested* to prove it to the client.

The design of DropBear realizes the above functionality by following an *execute/agree/attest* strategy, which separates (i) the execution of inference requests and model updates from (ii) the agreement and ordering of the results; and (iii) their attestation for the client. By separating these phases, it becomes possible to achieve higher request throughput, because operations can be batched independently per phase. First requests are grouped into *execution batches*, which are executed in parallel on all N nodes; and then results are collected and grouped again into *agreement batches*, which agree on a trustworthy result using a Byzantine consensus protocol. DropBear selects the maximum execution batch size based on the level of GPU parallelism; and the maximum agreement batch size based on the network latencies between nodes. We experimentally demonstrate the performance benefit of decoupling execution from

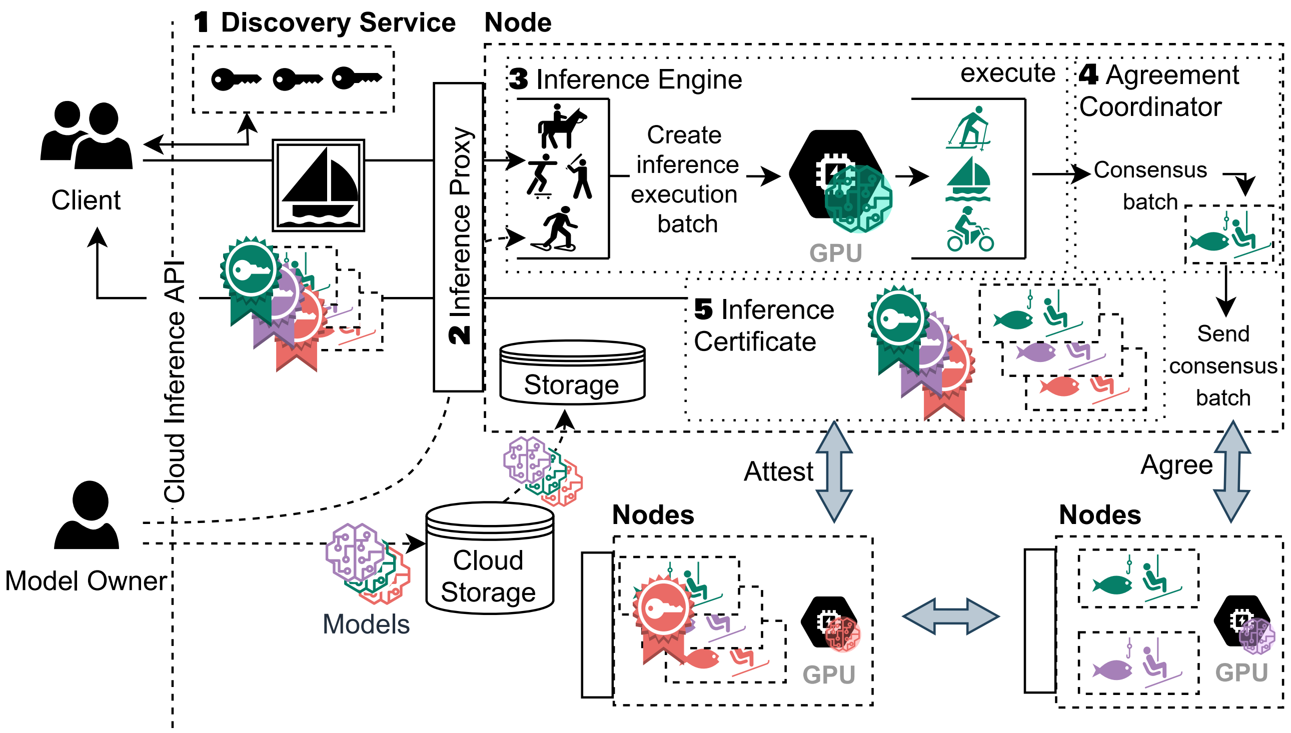


Figure 4.2: DropBear design

agreement in Section 4.7.3.

A DropBear node is identified uniquely by its public/private signing keys. Clients use a trusted **discovery service** ❶ to learn about node identities. They can then send API requests to any node. A node receives requests through its **inference proxy** ❷, which acts as an endpoint for the DropBear API (see Section 4.4) and hides the distributed nature of the service. A node uses its public key to establish a secure TLS communication channel with the client and authenticates it.

The inference proxy forwards requests to the **inference engines** ❸ of other nodes. The inference engine batches requests and uses a GPU inference library to *execute* batches on GPUs. After execution, the **agreement coordinator** ❹ *agrees* on the order of inference requests and ensemble updates, and on their execution results through Byzantine consensus. Finally, the inference proxy *attests* the result by constructing an **inference certificate** ❺ The inference certificate includes appropriate attestations that prove the result's trustworthiness.

A DropBear deployment also provides individual accountability to reduce the likelihood of malicious nodes returning dishonest results. This is realized as DropBear is a permissioned ledger and every node retains a ledger of the protocol messages which are committed by the service.

4.5.2 Execute

After receiving an API call (see Table 4.1), the inference proxy orchestrates the execution of the request and forwards it to *all* other nodes. Requests for loading ensembles (`load_ensemble`) and executing inference requests (`request_inf`) are then executed by the inference engine on each node, as described below; requests for activating/retiring ensembles (`activate/retire_ensemble`) are directly ordered by the agreement coordinator (see Section 4.5.3).

Loading ensembles. After receiving a `load_ensemble` request, an inference engine loads one or more models from the ensemble into GPU memory, which makes them available for inference execution. The models are retrieved from the specified remote URLs and cached locally by the node.

Each node must decide which models from an ensemble to load. If the number of models in the ensemble, $|\mathcal{E}|$, is larger than N, nodes load disjoint partitions of models; otherwise, some models are replicated across nodes. The inference engine selects deterministically which $c = \lceil N/|\mathcal{E}| \rceil$ models to load. The ordered list of models in the ensemble is divided into c-sized groups, and inference engines are assigned groups based on a total order imposed by the nodes' public keys.

Executing inference requests. The inference engine executes inference requests in batches by grouping them into *execution batches* that refer to the same model. There is a trade-off when setting the maximum batch size: while larger batches help exploit GPU parallelism, they increase latency, consume more GPU memory and may become bottlenecked by PCIe bandwidth. The actual used batch size also depends on the number of concurrently submitted inference requests that refer to the same model. We explore the impact of the maximum execution batch size on performance in Section 4.7.3.

DropBear guarantees that inference requests are always executed against the *latest* activated model version. This model, however, is only determined at agreement time after requests have been ordered (see Section 4.5.3)—at execution time, multiple model versions belonging to the same ensemble may exist. To overcome this problem, the inference engine executes such inference requests against *all* versions. This ensures that, even if a new ensemble version is activated before the inference request has been ordered, the request had been executed against that version.

4.5.3 Agree

Parallel batches execute across all nodes, and nodes must agree on the order of inference requests and ensemble activations/retirements to return consistent and trustworthy results to the client. DropBear uses a Byzantine consensus protocol (PBFT [49]) to order requests, which can mitigate f untrustworthy nodes that return incorrect results. The inference results produced by executing the ordered requests are used to establish trusted agreement (see Section 4.3.1), which distributes trust among nodes in multiple cloud environments.

DropBear utilises the PBFT protocol to choose a *primary* node, which is determined based on a monotonically increasing counter called view v. The *agreement coordinator* on the primary node is responsible for ordering inference requests relative to ensemble updates. It creates *agreement batches* and assigns a monotonically increasing sequence number n to each batch. Each agreement batch contains an ordered list of inference requests and ensemble updates.

Ordering requests. The primary node's agreement coordinator proposes an agreement batch, which contains an ordered list of requests $\mathcal{O}$. It then includes the agreement batch in a PRE-PREPARE message with the following format: $\langle \textsf{PRE-PREPARE}, v, n, H(\mathcal{O}), \mathcal{R}_r \rangle_\sigma$. The message specifies which node's agreement coordinator created the ordering by including the view v and records the agreement batch's sequence number n and its hash $H(\mathcal{O})$. It also includes $\mathcal{R}_r$, which is the root of a Merkle tree $\mathcal{R}$ that is constructed by the agreement coordinator over the the inference requests and results. As will be explained in Section 4.5.4, the signed Merkle tree root is used as an optimization to reduce the number of signatures that must be verified. Finally, σ is a signature over the whole message. The PRE-PREPARE message, $\mathcal{R}$, and $\mathcal{O}$ are sent to the agreement coordinators on all other nodes, which act as *backup nodes*.

When a backup node's agreement coordinator receives the primary's agreement batch, it constructs its own agreement batch and sends it in a PREPARE message to all other nodes. This enables the nodes to obtain the required number of results to check that they are trustworthy. The PREPARE message has the following format: $\langle \textsf{PREPARE}, v, n, j, i, \mathcal{R}_r \rangle_{\sigma_i}$. It also includes $\mathcal{R}_r$, which is the root of a Merkle tree whose leaves are the batch's inference requests, results, and hashes of the model used to create the inference results. The PREPARE also includes ordering information, but the agreement coordinator removes the need to include $\mathcal{O}$ by including j, a hash of the PRE-PREPARE message. Finally, the message contains i, the public key of the node that created the message, and is signed by public key σ_i. Again, the agreement coordinator sends the PREPARE message and $\mathcal{R}$ to all other agreement coordinators.

When ordering the agreement batches, the agreement coordinator on the primary node considers each request in the batch and updates its state: for an `activate_ensemble` request, the active version of the ensemble is updated; a `retire_ensemble` request deactivates all versions of the ensemble, reclaiming used GPU memory; for a `req_inference` request, the agreement coordinator considers all inference results that were previously produced for different versions of the ensemble. Since the inference request has now been ordered with respect to previous `activate_ensemble` requests, the agreement coordinator only retains the result for the currently active version of the ensemble.

The agreement batch size that achieves the best performance depends on the properties of the WAN between the nodes. A larger batch size reduces the impact of the WAN latency between nodes, but it increases the request latency that clients experience. The agreement batch size grows linearly with the size of the inference results, as they are included in the batch. Inference results can be large, e.g., a single ResNet model may return a 4000 byte inference result. Our experimental results show that, if the agreement batch size is chosen correctly, the WAN latency does not limited overall request throughput (see Section 4.7.6).

Achieving agreement. When a backup node's agreement coordinator receives the primary's agreement batch, it applies the ordering information by considering each request in the batch and either selecting the appropriate inference result or updating its state. Since it sends its inference results in an agreement batch to the other node's agreement coordinators, every agreement coordinator will receive the $N-f$ inference results required to produce a trustworthy inference result (see Section 4.3.2).

To produce a trustworthy inference result, the agreement coordinator applies the *distance function* after it has obtained $N-f$ inference results. The distance function is provided by the model owner as part of the `load_ensemble` request (see Table 4.1). From the $N-f$ inference results, the function removes ones that are not subject to the agreement bounds specified by the model owner. For each request in the agreement batch, the agreement coordinator applies the distance function of the active version of the ensemble at the time when the request was ordered. Note that the distance function is executed within a sandbox, isolating it from the node's execution and bounding its resources.

4.5.4 Attest

After getting a set of trustworthy inference results, the inference proxy must prove the results' trustworthiness to the client. A client requires that at least one trustworthy node has attested the results. DropBear thus provides the client with $f+1$ attestations of which at least one is trustworthy.

An attestation is created by the agreement coordinator, which signs the inference results that it obtained after applying the distance function. The results are signed by the node's private key and are broadcast to all other agreement coordinators. Based on this, an inference proxy can construct an inference certificate, which shows that at least $f+1$ nodes have agreed and attested at least $N-f$ results.

Attesting results. For an agreement coordinator to create an attestation, it must obtain $N-f$ inference results. To proof to the client that the results were produced on multiple nodes, they must be signed with the private keys of these nodes. However, instead of signing all inference results in the agreement batch separately, the agreement coordinator constructs a Merkle tree $\mathcal{R}$ where each leaf is an inference request and result. This allows the agreement coordinator to sign the root of the Merkle tree $\mathcal{R}_r$ once. By including the Merkle authentication path in the inference certificate, the signature is linked to the inference result.

A node's agreement coordinator waits until it has received $N-f$ agreement batches before attesting their trustworthiness (see Section 4.5.3). It then attests agreement batches by signing a hash of the results within the batch and sending it in a COMMIT message to all other agreement coordinators. The COMMIT message has the format: $\langle \mathsf{COMMIT}, v, n, j, i, \mathcal{A}_r \rangle_{\sigma_i}$. As before, including the root of a Merkle tree $\mathcal{A}$ avoids the need to sign each attestation, because the leaves are the hashes of the inference requests and results being attested. If the agreement coordinator attests all the results in an agreement batch, it includes $\mathcal{R}_r$ from the agreement batch in $\mathcal{A}$ instead of adding individual results. The agreement coordinator sends the COMMIT message and $\mathcal{A}$ to all other agreement coordinators.

If an agreement coordinator receives an attestation for an inference result that it has not obtained, it requests that the sender forward the missing inference result and PREPARE message. The agreement coordinator waits for a response before it considers the attestation as having been received and ready to use. An agreement coordinator considers requests ordered after it has received $N-f-1$ COMMIT messages.

Ledger. DropBear enables auditing and individual accountability by maintaining a ledger like IA-CCF (see Section 3.3.1). Each node writes the PRE-PREPARE, PREPARE, and COMMIT messages to the ledger at the time when a node either sends or receives a valid PRE-PREPARE message. We extend the definition of the PRE-PRPEARE message to include E_{n-P} where, E is a bitmap describing which node's PREPARE and COMMIT messages are stored in the ledger and P is the maximum number of concurrent batches that may be in-progress and uncommitted. In addition, every node will write the req_inference once they have been ordered by a node's consensus protocol during the agree stage (see Section 4.5.3).

Creating inference certificates. After having obtained $N-f$ attestations, an inference proxy can create an *inference certificate*, which has the following format: $\langle v, n, H(\mathcal{O}), \mathcal{S}, \mathcal{P}, \mathcal{D} \rangle$. It contains $\mathcal{S}$, the signatures over the PRE-PREPARE, PREPARE, and COMMIT messages; $\mathcal{P}$, the authentication paths from the Merkle tree R; and $\mathcal{D}$, the authentication paths over the attestations $\mathcal{A}$. After the inference certificate has been created, the inference proxy forwards it, together with the inference results, to the client.

As mentioned in Section 4.4, the client can now verify the inference certificate and results using the `verify_cert` function. Algorithm 4.1 provides the functions's pseudocode. First, the client checks that the nodes signed their own inference results. It confirms that at least $N-f$ results were returned (line 2) and iterates through the inference results (line 4): it obtains

Algorithm 4.1: Verify inference certificate (where public keys $\mathcal{K}$ are obtained from the discovery service.)

```
1  on verify_cert($\langle v, n, \mathsf{H}(\mathcal{O}), \mathcal{S}, \mathcal{P}, \mathcal{D}\rangle, \mathcal{R}, r, \mathcal{K}, f$)
2      if $|\mathcal{R}| < N - f$ then return False
3      $\mathsf{h_{pp}} \leftarrow \mathsf{H}(\langle\mathsf{PRE\text{-}PREPARE}\ v, n, H(\mathcal{O}), \mathcal{R}_{i_r}\rangle_{\mathcal{S}_i})$
4      foreach $i, \mathcal{R}_i \in \mathcal{R}$ do
5          $M_r \leftarrow$get_merkle_root($i, \mathcal{P}, r, \mathcal{R}_i$)
6          if $\neg$ verify_sig($v, n, H(\mathcal{O}), M_r, i, \mathsf{h_{pp}}, \mathcal{S}_i, \mathcal{K}_i$) then return False
7          attestations$_{\mathcal{R}_i} \leftarrow$ get_attestations ($\mathcal{D}, \mathcal{R}_i$)
8          if $|$attestations$_{\mathcal{R}_i}| \leq f$ then return False
9          foreach $\langle i, M_{path}\rangle \in$ attestsations$_{\mathcal{R}_i}$ do
10             $M_r \leftarrow$get_merkle_root($M_{path}, r, \mathcal{R}_i$)
11             if $\neg$ verify_sig($v, n, M_r, i, \mathsf{h_{pp}}, \mathcal{S}_i, \mathcal{K}_i$) then return False
12     return True
```

the root of the Merkle tree $\mathcal{R}$ using the authentication path provided in $\mathcal{P}$ (line 5), and uses this information to verify the node's signature over its result (line 6).

The client then confirms that the results were attested by at least one trustworthy node. It ensures that a minimum number of nodes have attested each result (line 8). For each attestation (line 9), it reconstructs the root of each node's Merkle tree over their attestations (line 10) and verifies the signatures (line 11).

4.5.5 Discussion

Security analysis. We discuss how the trustworthiness of inference results is guaranteed by DropBear's design.

Untrustworthy inference proxy. When malicious inference proxy receives a request from a client, it either: (i) forwards the request to a subset of the nodes, which means that the trustworthy nodes forward the request to one another [48, 57]; or (ii) does not forward the request, thus not returning a valid inference certificate to the client. After a timeout, the client sends the request to another inference proxy. After at most f+1 tries, the client will find a trustworthy inference proxy.

Untrustworthy nodes attempt to tamper with the ordering and execution of requests. Since the PBFT protocol orders requests, the produced results are from models in the same version of an ensemble. If the primary node stops ordering requests, it is re-assigned through PBFT's view change protocol. The current primary is determined based on the monotonically-increasing view v, which increments each time the primary node is changed. To decide on the primary node, the nodes' public keys are ordered lexicographically, and the p^{th} node from the list is chosen as the primary where $p = v \bmod N$. A view change is triggered by backup nodes if they do not receive $N-f$ attestations within a timeout after receiving the request.

Attacking accuracy. Untrustworthy nodes may try to undermine the accuracy of the results returned from an ensemble (see Section 4.7.7). For example, a malicious node may slightly perturb the generated results to reduce overall ensemble accuracy [256]. The distance function, however, mitigates this issue by bounding the maximum tolerated divergence between models. In addition, more important models may be replicated by DropBear, or the model owner may choose to include only models in the ensemble that do not dominate accuracy.

Result aggregation. By design, DropBear does not aggregate the results from the models in the ensemble but leaves this task to the client, because the inference proxy cannot be trusted to aggregate results correctly. Instead, DropBear could be extended to provide trustworthy aggregated results by obtaining an aggregation function Σ_m from the model owner. It would then add an extra consensus round in its agree phase that would ensure that $f + 1$ nodes agree over the value of $\Sigma_m(Q_m(r))$.

Non-determinism. We assume that the execution of an inference request on the same GPU hardware against the same model returns the same result. However, if the same inference request is executed on different hardware, the results may differ. A model owner who is aware that inference computation will execute on heterogenous hardware may write a distance function that accounts for these differences.

State. DropBear maintains state information about a number of parameters including: the actively loaded ensembles; a mapping of the model owner to the ensemble they own; the users that may send inference requests to an ensemble; etc. As DropBear is built upon IA-CCF, all state is maintained in the IA-CCF key-value store (see Section 3.2). In addition, the cloud providers running the DropBear instance may define a number of stored procedures which clients and model owners may invoke. When invoked a stored procedures may update the data in the key-value store.

Active nodes. As DropBear extends the IA-CCF permissioned ledger, DropBear follows the same procedure to change the set of active nodes. The cloud providers running the DropBear service act as the members and submit the required governance requests and votes to add or remove nodes (see Section 3.5).

Note that a DropBear governance proposal request to add a new node must include the model of the proposed node's GPU, inference framework, and any other information that could affect an inference result (see Section 4.1). This information is required to ensure consistent playback during an audit (see Section 4.6.2).

4.6 Individual accountability

In the previous section, we described DropBear's design and explained how it provides trustworthy inference results using the execute-agree-attest strategy. In this section, we build on this design to show how DropBear can be made auditable and provide individual accountability.

4.6.1 Extended protocol messages

In the previous section we presented a simplification of the PRE-PREPARE, PREPARE, and COMMIT messages. We extend the three messages with the parameters of PRE-PREPARE, PREPARE, and COMMIT messages from L-PBFT (see Section 3.2), such that each message contains the superset of the parameters. This provides DropBear with the same individual accountability properties as IA-CCF.

This changes DropBear protocol message to utilize the same PRE-PREPARE and PREPARE messages as L-PBFT except that DropBear's consensus PRE-PREPARE and PREPARE messages include $\mathcal{R}_r$ which is the root of the node's per batch Merkle tree whose leaves are the batch's inference requests and results. In addition, DropBear extends L-PBFT's COMMIT message to include $\mathcal{A}_r$ which is the root of a Merkle tree $\mathcal{A}$ whose leaves are the hashes of the inference requests and results being attested by the node sending the COMMIT message.

DropBear's VIEW-CHANGE and NEW-VIEW messages are copied from L-PBFT and the parameters are not modified.

4.6.2 Auditing

Auditing a DropBear ledger follows a similar pattern to auditing an IA-CCF ledger. Audits are triggered when an entity, usually a client, obtains an inference result with a valid inference certificate that they believe could not have been produced if the inference request was correctly executed against the models in the currently active ensemble. For example, if an insurance policy is approved that should not have been approved (see 4.1). There are two such scenarios when audits are triggered: (i) the client believes a dishonest inference result was produced or; alternatively (ii), the client believes that there is no valid set of `load_ensemble`, `activate_ensemble`, and `retire_ensemble` requests that could result in the inference request being executed against the models specified in the inference certificate. In both scenarios, the audit begins with the client passing the inference result and certificate that it believes to be dishonest to the model owner. The model owner – as the entity that uploaded the ensemble to the DropBear service – decides if it wants to perform an audit. This decision is made based on abnormalities observed in the inference requests, either by inference results observed on the ledger or complaints offered raised by clients. The model owner utilises the receipts that it has collected when it sent `load_ensemble`, `activate_ensemble`, and `retire_ensemble` requests along with the client's inference certificate to make its decision.

Linearizability violation. The first step in auditing an inference result and the accompanying inference certificate is deciding if the correct models were used to produce the inference result. The model owner checks which ensemble should be active by examining the receipts that it obtained when sending `activate_ensemble` requests. The model owner examines the sequence number assigned to the executed `activated_ensemble` requests to determine which models should be active. If the model owner determines that incorrect models were used to produce an inference result, the model owner performs an audit akin to that of IA-CCF.

To assign blame for violating linearizability, an audit of a DropBear ledger follows the auditing steps described in Section 3.4.1. In brief, an overview of auditing a DropBear ledger is as follows:

1. The auditor obtains the appropriate checkpoint and ledger from one of the replicas along with a receipt for the `activate_ensemble` with the highest sequence number that is lower than that of the suspected malicious inference certificate, i.e., the `activate_ensemble` request that activated the ensemble against which the inference request that is suspected to be malicious was executed.

2. The auditor loads the checkpoint and re-executes the `load_ensemble`, `activate_ensemble`, and `retire_ensemble` requests in the ledger. The auditor does not need to replay the `request_inf` entries as any correct execution of the requests cannot affect the state of the permissioned ledger (key-value store).

3. While re-executing the model update requests the auditor ensures that they were sent by model owners that have the appropriate permission.

If the above detects a linearizability violation, a uPoM is created. The uPoM is sent to the enforcer who punishes the dishonest replicas.

Invalid inference result. If a linearizability violation is not detected, it is possible that the malicious behaviour is the malicious execution of the inference request. To verify the inference execution the auditor re-executes the inference request. To ensure the correct re-execution of the inference results the auditor is required to obtain a computer that has the same GPU as the node which executed the request.

If the result from the re-execution does not match that of the inference certificate, a uPoM is created and sent to the enforcer to provide individual accountability. Individual accountability acts as a disincentive for cloud providers that attempt to gain value from returning incorrect results or as an incentive to secure the service provided to the model owner.

4.7 Evaluation

We evaluate DropBear to explore the cost of providing trustworthy ML inference (Section 4.7.2), the impact of separating execution and agreement batching (Section 4.7.3), the impact of model agreement (Section 4.7.4), the impact of ensemble updates (Section 4.7.5), and DropBear's scalability (Section 4.7.6). We finish by considering the impact of untrustworthy nodes on ensemble accuracy (Section 4.7.7) and of the sizes of inference requests and results (Section 4.7.8).

4.7.1 Experimental setup

We implement DropBear in approximately 5,000 lines of C++ code. It uses the ONNX Runtime (v1.8.1) [76] as part of its inference engine to execute inference requests on GPUs, EverCrypt's SHA functions [197], the MerkleCPP library [164], the MbedTLS library for client connections [161], and secp256k1 for all signatures [250].

Testbeds. Our experimental setup consists of three environments: (a) cluster—a dedicated 5-machine cluster, each with an 8-core 3.7-Ghz Intel E-2288G CPU with 16 GB of RAM and a 40 Gbps network with full bi-section bandwidth; (b) cloud/single-site—a single datacenter cluster in the Azure cloud (East US) with NC12s_v3 VMs, each with a 12-core 2.60-Ghz Intel E5-2690 CPU with 224 GB of RAM and 2 NVIDIA v100 GPUs with 16 GB of RAM; and (c) cloud/multi-site—a WAN configuration using the same VMs across 3 Azure regions (East US, East US 2, US South Central). All machines run Ubuntu Linux 18.04.6 LTS.

We set the execution batch size to 4, as larger batch sizes do not improve performance with our GPUs. In the single-site environments, we use an agreement batch size of 25 with a batch pipeline of 2; in cloud/multi-site, we use a batch size of 50 with a pipeline of 4.

Workloads. To emulate a realistic distribution of diverse DNN models in a cloud-based inference service, we use 42 ImageNet models taken from the ONNX and PyTorch model zoos [188, 198], as listed in Table 4.2, which categories the models into model families. These models vary in size from several to hundreds of MBs, covering various complexities and depths. We update all models to accept batches of inference requests. We employ a set of 1000 images taken from the ImageNet validation dataset [205] as input to inference requests.

4.7.2 Inference throughput and latency

We begin by evaluating the impact on throughput and latency when providing trustworthy inference requests using DropBear. We compare against two state-of-the-art distributed inference systems, which do not support trustworthy inference: (i) Clockwork [108] focuses on predictable inference latencies; and (ii) INFaaS [203] selects models based on the SLOs of inference requests.

Model family	**#Vars**	**Model family**	**#Vars**
ResNet [115, 116]	17	SqueezeNet [126]	3
VGG [221]	10	AlexNet [145]	1
DenseNet [121]	5	GoogleNet [229]	1
Inception [230]	1	MnasNet [231]	2
ResNext [251]	2		

Table 4.2: DNN models and variants used in evaluation

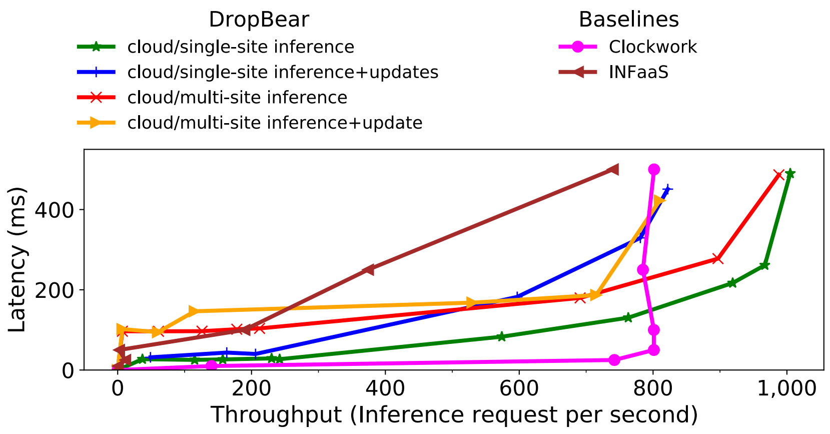


Figure 4.3: Request throughput versus latency (All systems use 15 ResNet50 instances. Clockwork and INFaaS use 1 worker node and the reported latency as their target SLO. DropBear is deployed with f=1 with all ensembles on all nodes.)

Similar to the authors of Clockwork, we were unable to obtain the resources to deploy prior systems. Clockwork requires more GPU memory (32 GB) than is available to us; INFaaS only supports AWS. Since the hardware configurations from the respective papers (Clockwork: Dell PowerEdge R740 servers with 32 CPU cores, 32 GB RAM, and 2 NVIDIA V100 GPUs; InFaaS: AWS m5.24xlarge and p3.2xlarge VMs) are similar to ours, we include their results for comparison.

We consider two request workloads for DropBear: inference only consists of inference requests; inference+updates also includes around 8% of model updates. Clockwork and INFaaS do not perform model updates.

Figure 4.3 shows a throughput vs. latency plot. DropBear's peak throughput with pure inference requests is 1005 request/s at a latency of 489 ms and 988 request/s at latency 487 ms in the cloud/multi-site configuration. When it also updates models, ordering updates with inference requests, the throughput drops to 822 request/s with an average latency of 451 ms and 810 request/s at latency 422 ms in the cloud/multi-site configuration. We observe that DropBear's throughput is bounded by the request execution on the GPU, based on the number of requests included in an execution batch and the consumed PCIe bandwidth when models

	Execution batch size				
# Ensembles	1	2	4	8	16
10	321	592	877	888	-
20	350	583	817	785	-
30	280	533	721	-	-
42	289	471	633	-	-

(a) inference+updates

	Execution batch size				
# Ensembles	1	2	4	8	16
10	302	521	844	924	904
20	333	564	818	900	-
30	278	495	735	722	-
42	291	489	681	696	-

(b) inference

Table 4.3: Batch sizes and ensemble counts ("-" denotes a GPU out-of-memory error.)

are loaded (see Section 4.7.3). The increase in latency arises from the cryptography operations to verify inference requests and produce inference certificates.

In comparison, Clockwork achieves a peak throughput of 801 requests/s with a latency of 50 ms without guaranteeing the trustworthiness of results. Clockwork is unable to increase throughput beyond this point, even when its SLO budget is increased 10 times to 500 ms. INFaaS obtains a comparable throughput of 739 requests/s with 500 ms latency.

Despite the additional communication and cryptographic overhead that DropBear's execute/agree/attest strategy introduces, it manages to achieve higher throughput. This is due to the fact that the bottleneck becomes the expensive GPU inference computation, which effectively hides the additional cost of trustworthy agreement. DropBear's slightly higher throughput can be explained by the systems' different design goals: Clockwork makes a throughput trade-off to increase latency predictability and does not use multiple threads to send inference requests to the same GPU; INFaaS trades off throughput to obtain the highest inference accuracy within a time budget and uses the Triton Inference server to execute inference requests.

To put DropBear's peak throughput into perspective, we use a micro-benchmark to measure the maximum throughput of inference requests that the ONNX Runtime inference library [76] can achieve on a single Azure NC12s_v3 node. We find that, with pure inference requests, DropBear's throughput reported above is within 4% of this maximum. This demonstrates that the DropBear's performance is limited by the GPU computation performed by a state-of-the-art inference library such as ONNX Runtime.

4.7.3 Impact of batching

We explore the impact of the execution phase in DropBear's *execute/agree/attest* strategy under different workloads.

First, we measure throughput and latency when varying the number of distinct ensembles (10, 20, 30, 42) over which inference requests are served. Figure 4.4 shows that DropBear achieves a throughput of 831 and 698 requests/s with 10 ensembles and 42 ensembles, respec-

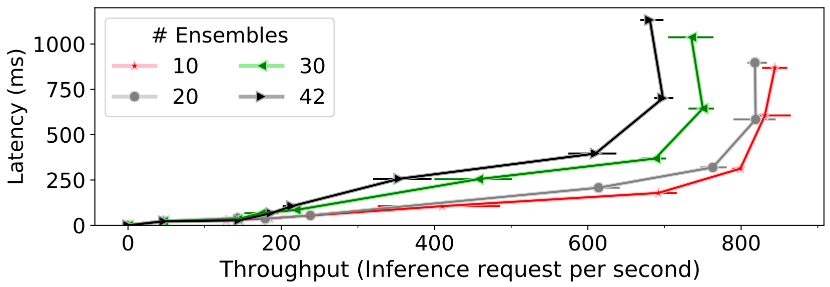


Figure 4.4: Varying ensemble count (f=1 on cloud/single-site.)

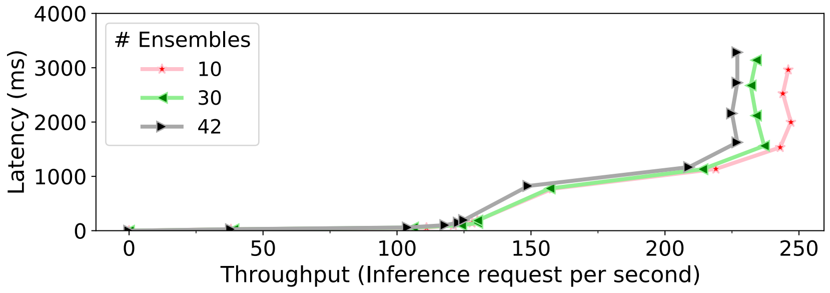


Figure 4.5: Comparison with agree/execute strategy (f=1 on cloud/single-site.)

tively, while the latency stays below 700 ms. The throughput with more ensembles decreases because a higher number of ensembles means that there are fewer opportunities to create large execution batches that refer to the same model.

We confirm this explanation by investigating how the maximum execution batch size and the ensemble count affects throughout. We considers two request workloads, pure inference requests (inference) and with 8% updates (inference+updates). Table 4.3 shows that, with inference requests and model updates, batch sizes of 4 and 8 do not change throughput much, but a batch size of 4 allows DropBear to support more loaded ensembles due to GPU memory constraints. For that reason, we use a batch size of 4 in our cloud/single-site experiments.

With pure inference requests, DropBear's throughput improves when increasing the execution batch size. This result is consistent with findings for other inference services [108, 203, 236]. The impact of loading models means that the PCIe bus becomes saturated, delaying the transfer of requests to the GPU, thus mitigating the benefit of a larger execution batch size.

Second, we examine the impact of separating execution and agreement batches in DropBear's design. For this, we implement a DropBear variant that executes inference requests during the agreement phase of the Byzantine consensus protocol (agree/execute). In this approach, the consensus protocol first orders requests into agreement batches and then executes inference requests.

Figure 4.5 shows a throughput vs. latency plot of DropBear with agree/execute for different ensemble counts. Here, the peak throughput is 246 request/s, compared to DropBear's

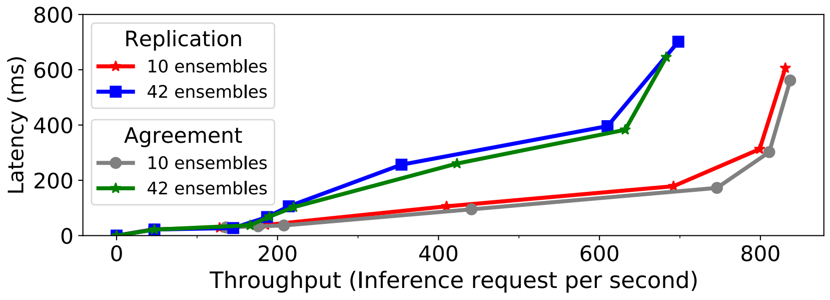


Figure 4.6: Model replication vs. agreement (f=1 on cloud/single-site.)

854 requests/s. This 3× performance reduction stems from two issues. The first issue is that the agree/execute strategy results in lower GPU utilization. While it can exploit larger agreement batches (3 versus 2) to mitigate this, this still fails to remove all idle periods of the GPU. Even when we turn off execution batches, DropBear still achieves 316 requests/s.

The second issue is that the lack of batching during inference execution with agree/execute. When inference requests are executed out-of-order within an agreement batch, thus allowing batching, there are fewer opportunities to batch inference requests, because agreement batches contain fewer executable requests (25 in cloud/single-site; 50 in our cloud/multi-site configuration). Instead, with the *execute/agree/attest* strategy, the execution batch size is bounded only by the number of concurrent inference requests submitted by clients.

4.7.4 Model replication vs. agreement

Next, we consider the overhead that DropBear's agreement approach introduces using a distance function. We compare against a variant of DropBear that only replicates the same sample model on multiple nodes without agreement.

Figure 4.6 shows a throughput/latency plot when the models in the ensemble are either replicated or use the agreement approach. With replication, we see a maximum throughput of 837 requests/s with 10 ensembles and 683 requests/s with 42 ensembles and latency of 561 ms and 646 ms, respectively. While using the agreement approach, we observe a peak throughput of 831 and 698 requests/s with 10 ensembles and 42 ensembles, and latency of 606 ms and 700 ms, respectively. We observe that the agreement approach increases latency by 8% under heavy client load, while under low load latency only increases 2%. This disparity is due to the computational requirements of the distance function.

4.7.5 Updating ensembles

We explore how model updates affect concurrently executing inference requests. In this experiment, we slowly increase the percentage of clients that issue `load_ensemble`

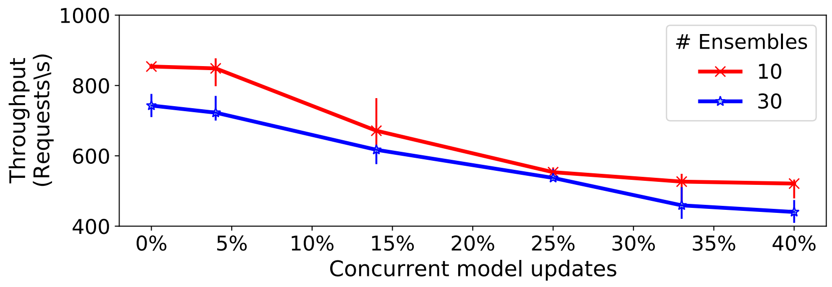


Figure 4.7: Concurrent model updates (f=1 on cloud/single-site.)

and `activate_ensemble` requests. We consider two configurations with 10 and 30 deployed ensembles, respectively.

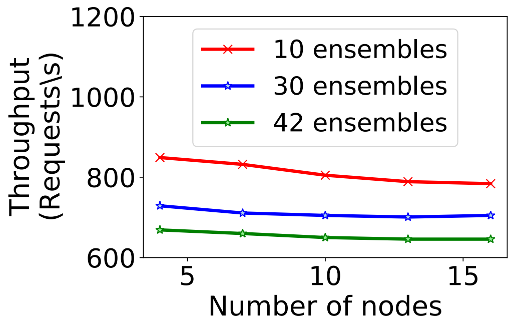
Figure 4.7 shows that, with 30 ensembles, there is a gentle decline in the number of executed inference requests as the percentage of ensemble updates increases. This is due to three factors, which grow with the number of concurrent ensemble updates: (i) the network bandwidth used to copy models from remote storage; (ii) the PCIe utilization when models are copied to the GPU; and (iii) the overhead of executing inference requests against multiple models when a new model update has been loaded but not yet activated. These three factors reduce the throughput from 743 requests/s, when no ensembles are updated, to 440 requests/s with 40% of ensemble updates. When more than 34% of requests are model updates, the model updates begin queueing at the GPUs, which reduces the impact on inference requests, as can be seen from the change in slope.

With fewer loaded ensembles (10), we observe a faster degradation in the rate of inference requests until 25% of requests are model updates. This is due to the greater likelihood of inference requests being executed against an updating ensemble, thus requiring to execute multiple inference requests against the current and new model. This decline abates when 34% of requests are ensemble updates due to queuing, as DropBear only allows one concurrent update per ensemble.

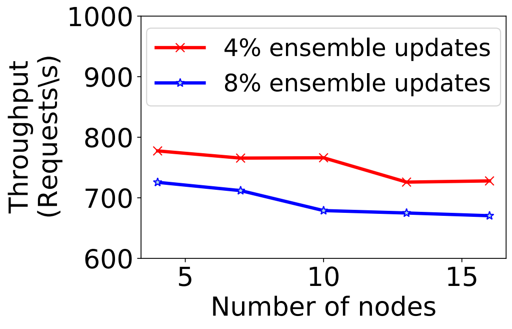
4.7.6 Scalability

It is possible to scale DropBear linearly by load-balancing requests between multiple independent deployments of DropBear. This has no impact on clients, because the execution of inference requests is stateless on the nodes. Each ensemble must be assigned to one deployment, otherwise the serializability of ensemble updates cannot be guaranteed. An individual deployment can also be scaled up by using larger VMs sizes for the nodes, e.g., with more GPUs.

In a single DropBear deployment, if we increase the number of nodes, the deployment can mitigate against more untrustworthy nodes f. We conduct an experiment that explores how scalable DropBear's design is with more nodes. We deploy nodes in our cloud/multi-site configuration across 3 Azure regions and change the node count from 4 to 16, thus increasing



(a) inference



(b) inference+update

Figure 4.8: Increasing node count (f=1 to 5 on cloud/multi-site.)

f from 1 to 5. We locate the primary node in US South Central, forcing worst-case round-trip network latencies of 32 ms.

First, we explore how the throughput with different ensemble counts is affected when increasing the node count. Figure 4.8a shows that, with 4 nodes distributed across the 3 Azure regions, the throughput of 849 requests/s is similar to the previous cloud/single-site experiment (see Figure 4.4). With 16 nodes, we see a throughput degradation to 784 requests/s when 10 ensembles are loaded. Adding more nodes increases the message and cryptographic overheads due to the extra signatures and Merkle tree roots that must be verified for every agreement batch. The throughput degradation with 30 and 42 loaded ensembles is similar. The reduced throughput with more ensembles is also caused by the less effective batching—the same as in the cloud/single-site experiments (see Section 4.7.3).

Next, we investigate how the workload mix of inference and update requests affects throughput with more nodes. Figure 4.8b shows DropBear throughput in the cloud/multi-site configuration with clients that issue `load_ensemble` and `activate_ensemble` requests. When 4% of clients update ensembles, the inference request decreases from 777 to 728 requests/s as f increases from 1 to 5. While the throughput decreases from 726 to 670 requests/s when 8% of clients update ensembles. Increasing the number of ensemble updates results in higher PCIe utilization forcing inference requests to queue when being transferred to the GPU.

4.7.7 Untrustworthy nodes

We examine how untrustworthy nodes affect the accuracy of an inference result in an ensemble. For this, we use the ImageNet1000 (mini) [74] dataset as input for 2 ensembles, each with 4 models: ensemble A contains ResNet152, VGG19, DenseNet-201, and ResNeXt101; and ensemble B contains GoogleNet, ResNet18, MnasNet0.5, and VGG11. We combine the results returned when executing an inference request: for each result, we take the label with the highest confidence and check if more than f models made the same prediction. When all nodes are

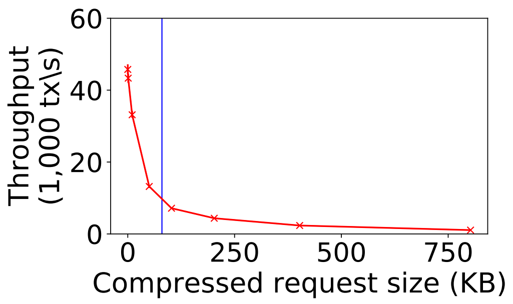
Dishonest result	**Accuracy**
VGG19	76.9%
DenseNet-201	75.9%
ResNet152	75.8%
ResNeXt101	75.3%

(a) ensemble A

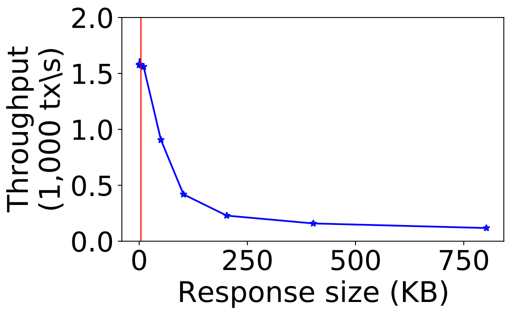
Dishonest result	**Accuracy**
VGG11	66.3%
ResNet18	65.4%
GoogleNet	65.1%
MnasNet	63.0%

(b) ensemble B

Table 4.4: Ensemble inference accuracy with untrustworthy nodes (The top-1 accuracy when f=1 and 1 untrustworthy node returns a dishonest model result.)



(a) Increasing request size



(b) Increasing response size

Figure 4.9: Inference request/result size (f=1 on cluster.)

trustworthy, ensemble A has an accuracy of 79.9% and ensemble B has an accuracy 70.0%. Here the ensemble training technique improves accuracy up to 7.5pp compared to a single model.

We now consider when an untrustworthy node returns a dishonest result. Table 4.4 shows that, with one untrustworthy node, ensemble A achieves a maximum accuracy of 76.9% and minimum accuracy of 75.3%—a reduction in accuracy by up to 4.6pp; ensemble B has an accuracy range of 66.3% to 63.0% (reduced by up to 7pp). With the evaluated models, an untrustworthy node thus reduces accuracy up to 2pp when compared to replicating a single model. We observe that DropBear provides a significant improvement to accuracy over replication when returning all honest inference results and only a small accuracy reduction when a dishonest result is returned. Further, DropBear can utilize techniques such as proactive recovery [50] to bound the time a node returns dishonest results.

4.7.8 Size of inference requests and results

To understand how DropBear is impacted by sizes of requests and responses, we measure throughput after replacing GPU inference execution with a randomly sampled response. Generally, models that operate on larger inputs/outputs are more complex and thus would be GPU bottlenecked (see Section 4.7.2).

First, we vary the compressed request size from 16 bytes to 800 KB, with a fixed response size of 4000 bytes. Note that DropBear clients compress images using the lowest level of zlib [75] compression. We find that the average compressed image size from ImageNet1000 (mini) [74] is 80 KB.

Figure 4.9a shows that a peak throughput of 45,765 requests/s is achieved with 16-byte requests. With 1 KB and 10 KB requests, throughput reduces to 43,335 requests/s and 33,149 requests/s, respectively, with larger requests halving throughput as the request size doubles. The typical ImageNet size of 80 KB (blue line) achieves 1581 requests/s.

On small inputs, the overhead of verifying inference request signatures becomes the bottleneck. When compressed requests are larger than 100 KB, the bottleneck shifts to creating and verifying the Merkle trees over the requests/responses (see Section 4.5.3). This cost could be reduced by splitting the inputs into smaller chunks in the Merkle tree and computing their hashes in parallel across a larger number of threads.

In Figure 4.9b, we vary the inference result from 16 bytes to 800 KB, with the representative compressed request size of 80 KB. We observe that peak throughput is obtained when results are between 16 bytes and 10 KB, with typical ResNet output of 4 KB marked as a red line. For results over 50 KB, throughput halves as the size doubles. Similar to the previous experiment, DropBear remains CPU bound on signature verification when the results are small, and the bottleneck shifts to Merkle tree computation for result sizes larger than 10 KB.

4.8 Related work

Distributed ML inference. Clipper [64] provided one of the first abstraction layers on top of multiple inference serving frameworks. INFaaS [203] allows users to specify a minimal confidence and time budget and selects an appropriate model against which the inference requests are executed. Clockwork [108] make the latency of inference execution predictable. None of these systems assume untrustworthy nodes that may interfere with the returned result. ModelSwitch [260] selects models that reduce inference confidence to maintain a request latency SLO. Tolerance Tiers [114] provides an API that allows users to decide if a simpler model should be used when an SLO is violated. DropBear focuses on trustworthy inference, but it could use SLO-aware policies for model selection from an ensemble.

TensorFlow serving [232], Triton inference serving [236], and TorchServe [234] provide developers a mechanism to host models built in popular ML frameworks. These systems simplify inference serving in real-world deployments. The concept of a cloud-hosted ML inference service was introduced by Kim et al. [137]. Inference-as-a-service products, e.g., AWS SageMaker [8], AI Platform [100], and Azure ML [166], have been deployed by all major cloud providers.

Secure ML inference. Securely obtaining inference results has been studied before. Bost et al. [38] and SecureML [174] use privacy-preserving techniques to execute particular ML inference workloads, but they cannot provide a general trustworthy ML inference in the cloud. CrypTFlow [147] and CrypTFlow2 [64] obtain cryptographically secure inference results through multi-party and two-party computation. While they manage to secure larger models, such as ResNet, their performance precludes practical cloud deployments: they require over a minute to compute a single ResNet50 inference result.

Slalom [235] and Privado [102] exploit trusted execution environment (TEEs) [61] to produce trustworthy inference. Therefore, they rely on the security of TEEs, which have suffered from successful attacks [40, 176, 239, 183]. In addition, these solutions are restricted to CPU-based inference, which increases latencies by several orders of magnitude compared to GPUs. Graviton [242] proposes a novel TEE abstraction that encompasses a GPU, but it requires hardware changes, which are unavailable today.

Byzantine ML. To our knowledge, there are no ML inference systems that assume Byzantine node behaviour. Byzantine failures have been shown to be catastrophic for ML training [105], and researchers have investigated the impact of Byzantine adversaries on distributed optimization [72]. Krum [34] and trimmed mean [252] create Byzantine-resilient algorithms for distributed stochastic gradient descent (SGD), which provides a mechanism for protecting against some Byzantine failures in large-scale machine learning. It can be paired with DropBear to mitigate malicious nodes during training and inference serving.

4.9 Summary

This chapter presented DropBear, the first cloud-based ML inference service that provides clients with strong integrity guarantees, while staying compatible with current inference APIs.

We first surveyed the field of cloud-based ML inference services. We then looked at the evolution of ML inference from providing ML models to end-users progressing to hosted services that receive inference requests via request messages that are executed against a ML model. We then explored the security implications that exist when a single cloud-hosted service is placed in a position of trust, i.e., the cloud hosted infrastructure is used to execute inference requests. This security concern led us to explore scenarios in which the end-user cannot trust the inference results produced on a virtual machine hosted by a cloud provider.

Next, we looked at use-cases for trustworthy ML inference and discussed the merit of using a trusted node versus a Byzantine consensus protocol to obtain agreement. Upon showing the drawbacks of utilizing trusted hardware, we defined trustworthy inference via a Byzantine agreement for a trustworthy inference service. Next, we showed the API for a trustworthy inference service that utilised Byzantine agreement and presented an overview of the system model for such a service.

We then described DropBear, our solution to the trustworthy inference problem. We explained that, to obtain high performance, DropBear's Byzantine consensus protocol implements an execute-agree-attest strategy that separates the expensive execution of inference requests from the agreement and attestation of inference results. This allows DropBear to implement a separate batching and execution strategy for hardware accelerators (e.g., GPUs) that are required during execution and the batching strategy for the Byzantine consensus protocol that optimizes for WANs. We then showed that a permissioned ledger that uses the execute-agree-attest execution strategy can provide individual accountability.

Finally, we evaluated the execute-agree-attest strategy and the performance claims made throughout the chapter. We showed how the execute-agree-attest strategy compares to a naive execution strategy and that a permissioned ledger can fully utilize a GPU when a proper execution strategy is employed by its consensus protocol.

Chapter 5

Parallel Acceleration of Permissioned Ledgers

This chapter describes Bunyip, the first permissioned ledger that allows transaction requests to be executed in parallel on both the primary and backup replicas. Bunyip's Byzantine consensus protocol separates the execution of transaction requests into two phases: requests are first executed on the primary, where the dependencies between transaction request execution are calculated; backup replicas use this dependency information to execute transaction requests in parallel. This ensures that Bunyip provides the same linearizability guarantees as a Byzantine fault tolerant consensus protocol that executes transaction requests sequentially e.g., IA-CCF (see Chapter 3).

In the context of Bunyip, we continue our exploration of performant transaction execution. We focus on parallelizing the execution of transaction requests, which have a high CPU overhead but where requests are not always dependent on the result of the previously executed transaction. We show that Bunyip, which provides the abstraction of a single large machine that executes transactions sequentially, can address the complexity of writing permissioned ledger applications while fully utilizing modern multi-threaded CPUs to execute transactions requests concurrently and address the practical concerns of executing complex transactions – see section 1.4.2.

5.1 Introduction

In this chapter, we continue our exploration into improving the performance of permissioned ledgers. We retain our focus on improving the performance of transaction request execution. In the previous chapter, we explored how permissioned ledgers that utilize Byzantine fault tolerance can improve request transaction execution with hardware accelerators. We now move our focus to improve the performance of transaction execution when the execution solely uses the CPU.

CPU centric transaction execution is the most common type employed by Byzantine consensus protocols. Previous work, has not addressed how non-conflicting transactions can be discovered, such that, they can be executed in parallel on all replicas (see Section 2.3). Existing work considers sharded execution, but each shard still executes requests sequentially. Alternatively, some attempts were made to decide which transactions a replica can execute in parallel, however, they depend on the developer building oracle services that guess which transactions can be parallelized (see Section 2.3.3). The limitations of either sharded execution or oracles are a barrier that makes increasing transaction execution throughput in permissioned ledgers unpractical.

In this chapter, we describe Bunyip, a permissioned ledger built on top of IA-CCF that addresses the problem of slow transaction request execution. Bunyip is an individually accountable permissioned ledger. Bunyip implements L-PBFT, receipts, ledger, and a governance protocol thus retaining the safety and liveness properties of IA-CCF. However, Bunyip changes the interface through which users and members interact with the service so that users need only to communicate with a single replica. Bunyip makes the following research contributions:

(1) We propose and describe a simple API that provides the abstraction of single threaded transaction execution. The service, however, executes transaction requests in parallel. Bunyip is able to execute transaction requests in parallel on all replicas. This transaction API does not require the developer to understand the parallelism employed by the replicas.

(2) We describe a method of automatically detecting conflicts between the execution of transaction requests. Unlike previous solutions, Bunyip does not require the developer to write code beyond their application to predict if two transactions may conflict. This removes the need to create an oracle that predicts which transactions can be executed in parallel (see Section 2.3.3).

Bunyip detects transaction conflicts by tracking all reads and writes to a key-value store. After the transactions successfully committed, Bunyip determines the most recently committed transaction after which the committed transaction must execute. Bunyip uses this transaction execution dependency information to create groups of transactions that can execute in parallel. These groups as designed to allow backup replicas to detect if a misbehaving primary has put transactions into an execution group that conflict with one another.

(3) Next, we show how a permissioned ledger that utilises a Byzantine fault tolerant consensus protocol can utilize the automatically generated transaction conflict information to parallelize transaction execution on all replicas. Bunyip expands the IA-CCF permissioned ledger, utilizing the primary replica's early-execution to calculate transaction conflicts within a batch. The primary replica then sends the conflict information to the backup replicas, allowing them to execute groups of non-conflicting transactions in parallel.

In addition to the research contributions, Bunyip generalizes DropBear's inference proxy to enable transaction requests to be sent to a single replica. This removes the requirement for a client to implement a networking library to communicate with Bunyip. Instead of the networking library, the client can utilize open-source tools, such as, Curl [65].

5.2 Motivation

Computational workloads have steadily increased in complexity surpassing the rate at which a single CPU core can execute [237]. Historically, this has been solved by CPUs becoming faster and able to process more operations per second [95]. This was the era in which PBFT [49] was proposed. PBFT, and its derivative proposals, depend on state machine replication where replicas independently execute a batch of transactions and agree on the result. This determinist execution was, done in part, by sequentially executing transactions on a single CPU core.

Within the last decade, the speed of a single CPU core has stopped increasing. CPU manufacturers have alternatively included multiple cores onto a single CPU [227]. This resulted in Byzantine consensus protocols not taking advantage of multi-core CPUs.

5.2.1 Applications

Modern distributed systems run complex user applications. In contrast, permissioned ledgers and blockchains primarily run simple applications, such as transferring funds and tokens [178]. One key factor that stops permissioned ledgers from being adopted as a platform to run complex workloads is the performance of transaction execution. Applications that a performant permissioned ledger could handle include:

Transparent supply chain. Distribution and inventory tracking is recognized as a target application for ledger-based platforms [91, 21]. However, many of the efforts to build ledger-based systems that record supply chain events have been hampered by performance issues [112].

Customer relationship management. Sales, recruitment, and other departments in private, public, and governmental organizations maintain a central database to record interactions. These databases process a large number of computationally inexpensive transactions. However, mixed in with the computationally inexpensive transactions are online analytic processing transactions (OLAP). OLTP transactions may access a large amount of data with which to perform computationally expensive operations. Modern databases utilize multi-core CPUs to accelerate OLAP workloads [170].

Medical records. Medical record databases store patient data which, similarly, to customer relationship management data, is often updated and processed by analytics transactions. The medical record databases performance is commonly accelerated by parallelizing transaction execution on multiple CPU cores.

5.2.2 Threat model

Bunyip inherits the IA-CCF threat model (see Section 3.2). We assume an attacker that can compromise replicas, clients, auditors, and members to make them behave arbitrarily, but cannot break the cryptographic primitives. We trust the enforcer to assign blame to replicas and the members that operate them only when it verifies a valid uPoM or fails to obtain data for auditing. Bunyip provides linearizability and liveness if fewer than 1/3 of the replicas are compromised [49].

5.3 Bunyip API

Bunyip, as IA-CCF, has *members* and *clients* who have different responsibilities. Members are responsible for running replicas, changing the members and replica set, and the stored procedures that can be evoked by clients (see Section 3.2). Both model owners and clients must obtain an API endpoint to communicate with a Bunyip deployment. Bunyip utilises the same procedure as DropBear to obtain an endpoint URL (see Section 4.4). Members and clients call the `get_API_endpoints` function, which contacts a trusted discovery service at a well-known location and returns a list of URL endpoints (`API_URL`) together with their public keys. These identify the currently active set of N replicas that make the Bunyip service. API requests can be sent to any of the returned URL endpoints, but, unlike IA-CCF, Bunyip API requests are only sent to one endpoint.

Stored procedures. Interactions with Bunyip are all performed by members or clients executing stored procedures. To execute a stored procedure, a member or client creates a transaction request that includes all the input parameters required by the target stored procedure. In addition, the transaction request also includes the stored procedure's identifier. The transaction request is serialized into a well known format, e.g., JSON, and signed with the member or client's private key. Next, the member or client must determine how to send the serialized transaction request to the deployed Bunyip instance. The client proxy's URL endpoint (`API_URL`) is

API function	Description
`start_tx``()`	Begins a new transaction.
`commit_tx``(response)`	Commits the active transaction and if successful serializes `response` and sends the serialized value to the client.
`abort_tx``()`	Aborts the active transaction.
`kv_get_value``(key)` → value	Reads the key-value store value at key (`key`) within the scope of the active transaction.
`kv_set_value``(key, value)`	Set value (`value`) for key-value store key (`key`) within the scope of the active transaction.
`kv_has_value``(key)` → bool	Check if key-value store key (`key`) is set.

Table 5.1: Bunyip stored procedure API

```
1 def add_to_key(usr_num, usr_key, alt_key):
2   try:
3     start_tx()
4     if not kv_has_value(usr_key):
5       return_value = usr_num
6     else:
7       return_value = int(kv_get_value(alt_key)) + usr_num
8     kv_set_value(usr_key, return_value)
9     commit_tx(return_value)
10  except:
11    abort_tx()
```

Listing 5.1: Example use of Bunyip stored procedure

obtained by the member or client from the list of `API_URL`s returned by the discovery service. The serialized transaction request is then sent to the client proxy at one of the `API_URL` URLs.

After a client proxy deployed by an instance of Bunyip receives a transaction request, the client proxy forwards the transaction request to all the replicas. The replicas then execute the stored procedure. As Bunyip utilises L-PBFT a replica executes a transaction request after the request has been included in a batch (see Section 3.3.1).

Listing 5.1 shows a stored procedure that may be executed by a Bunyip replica where the stored procedure `add_to_key` is called with 3 parameters that must be included in the serialized transaction request (line 1). The stored procedure utilises the function call `start_tx` from the Bunyip stored procedure API (Table 5.1) to start a transaction (line 3) and then uses the `kv_has_value` function to check if the key-value store has the key `usr_key` (line 4). If `usr_key` does not exist in the key-value store `return_value` is set to `usr_num` (line 5) otherwise `return_value` is the sum of `usr_num` and a value from the key-value store (line 7). Finally, `kv_set_value` sets the key `usr_key` to `return_value` (line 8) and the transaction is committed and, if the transaction successfully commits, the variable `return_value` is sent back to the client (line 9). If an exception is thrown, the transaction is aborted when the function `abort_tx` is called (line 11).

Opacity. The L-PBFT consensus protocol implements a replicated state machine. A key requirement of replicated state machines is that all replicas have identical transaction execution. This is required for all transactions regardless if they successfully commit or eventually abort.

Bunyip reduces the burden placed on members who write Bunyip stored procedures by including opacity in the transaction engine. Opacity [104] is a property that provides strict serializability for all transactions, regardless if they commit successfully or abort. This simplification can be seen in Listing 5.1 as no application logic is required to check that `usr_key` and `alt_key` are from the same snapshot. Without opacity, some transactions that read an inconsistent snapshot could raise an error when performing the integer conversion on line 7.

5.4 Bunyip design

In this section, we describe Bunyip's design (Section 5.4.1) and explain the process the primary replica follows to create groups of transactions, which can be executed in parallel without compromising linearizability (Section 5.4.2), and how the backup replicas execute these groups and then confirm that no transactions within the group conflict with one another (Section 5.4.3). Next, we describe how a client receives a response and receipt via the client proxy (Section 5.4.4). We finish with a discussion of Bunyip (Section 5.4.5).

5.4.1 Overview

As shown in Figure 5.1, a deployment of Bunyip consists of N replicas distributed across multiple data centres connected by a WAN connection. Bunyip extends IA-CCF, and as such a deployment provides a service to clients. Clients send requests to execute transactions by calling stored procedures that define the service logic. Transactions are executed by replicas against a strictly-serializable key-value store that supports roll-back at transaction granularity. A transaction request reads and/or writes multiple key-value pairs and produces a transaction result.

A Bunyip deployment is managed by a consortium of members. A member operates a subset of replicas and is permitted to issue governance transaction requests that change the consortium membership, add or remove replicas, or update the stored procedures.

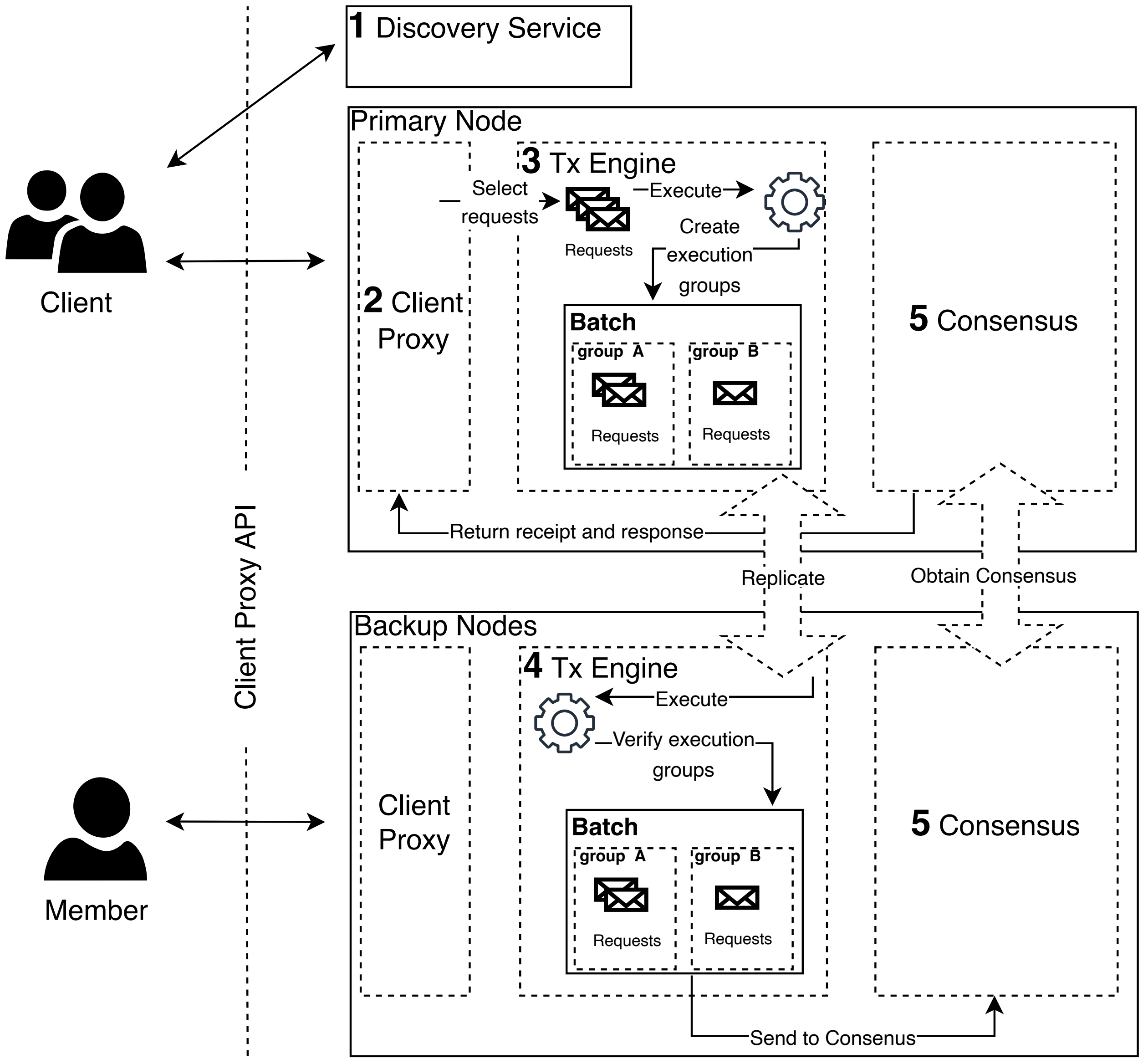


Figure 5.1: Bunyip architecture overview

Bunyip differs from IA-CCF by implementing a client proxy on every replica. The client proxy allows a client or member to send a transaction request to a single replica and the client proxy forwards the transaction request to all replicas in the deployment. After the transaction request is executed and committed, the client proxy collects the replies, collates the replies into a receipt, and sends the receipt along with the response message to the sender of the transaction request. Importantly, Bunyip executes groups of transaction requests in parallel on both the primary and backup replicas. If a replica's client proxy is compromised a client can resend their transaction to another replica's client proxy, where the request would be sent at most $2f + 1$ times to find an honest and healthy proxy. We discuss the security concerns of the inference proxy, that matches the concerns of the client proxy, in section 4.5.5.

A Bunyip replica is identified uniquely by its public/private signing keys. Clients use a trusted **discovery service ❶** to learn about node identities. They can then send transaction requests to any replica. A replica receives transaction requests through its **client proxy ❷**, which acts as an endpoint for Bunyip and hides the distributed nature of the service. A client proxy deployed on a replica uses the replica's public key to establish a secure TLS communication channel with the client and authenticates it.

After receiving a transaction request, the client proxy forwards the transaction request to all replicas. The **Tx engine ❸** selects transaction requests from those forwarded by the client proxy to the replica. After selecting the batches, the Tx engine executes the requests in parallel, creating a total order of executed and committed transaction requests. After execution, the Tx engine considers the dependencies of the executed transaction requests and creates groups of transaction requests that can be executed in parallel. The batch information along with which groups of transaction requests can be executed in parallel is replicated to the Tx engine on the backup replicas. The **Tx engine** on the backup replicas ❹ executes the groups of request transactions in parallel and after execution, the backup replica verifies that none of the transaction requests within a group have a dependency on one another. Finally, after all the transaction requests in a batch have been executed the batch is passed to the **consensus protocol ❺** which executes the *prepare* and *reply & commit* phases of the L-PBFT consensus protocol (see Section 3.3.1). During L-PBFT's reply & commit phase, the replicas send the reply and replyx message to the replica hosting the client proxy that received the transaction request from the client or member. The client proxy that received the reply and replyx messages converts them to a single transaction response message and a receipt (see Section 3.3.3). The transaction response and receipt are sent to the client. The client then validates the receipts and confirms the validity of the transaction response before acting on the information provided in the transaction response message.

5.4.2 Primary execution

The execution of a transaction request begins with a member or client contacting the discovery service to obtain the URLs and public keys of all the client proxies. The member or client then creates the transaction request and forwards it to a client proxy. After receiving a transaction request, the client proxy forwards the transaction requests to *all* replicas. Just as with L-PBFT, the transaction request on the primary replica joins a pool of requests that are ready to be ordered and executed. Eventually, the primary replica's Tx engine selects the transaction request to be included in a batch. After the primary replica selects a collection of transaction requests to be ordered and executed within a batch, it executes the selected transaction requests.

Algorithm 5.1: Verify parallel execution

```
1  on execute_transaction_request($tx\_request$)
2      $stored\_proc\_id, params$ ←get_execution_info ($tx\_request$)
3      $committed\_kv\_store$ ← get_committed_kv_store()
4      $tx\_kv = committed\_kv\_store$
5      $tx$ ← create_tx()
6      execute_stored_procedure($stored\_proc\_id, params, tx, tx\_kv$)
7      lock_kv_updates()
8      $committed\_kv\_store$ ← get_committed_kv_store()
9      foreach $kv\_pair \in$ get_read_kv_pairs($tx\_kv$) do
10         if get_key_read_version($kv\_pair$) $\neq$
             get_key_version($kv\_pair.key, committed\_kv\_store$) then
11             unlock_kv_updates()
12             return $Conflict$
13     unlock_kv_updates()
14     return $Success$
```

Algorithm 5.1 shows this execution.

Committed key-value store. Each replica maintains multiple versions of the key-value store, where a new version is added after a transaction has committed. In addition, each key is versioned and any update to the key-value pair increments the version of the key, however, a key is incremented at most once within a transaction. This key-value store is known as the committed key-value store.

Batch execution. The primary replica executes multiple transaction requests in parallel (multiple threads) and ensures that the execution of every transaction request is isolated from the execution of any other transaction request. This protection is achieved by the primary replica creating a copy of the most up to date version of the key-value store against which a transaction is executed (Tx key-value store). This is shown in Algorithm 5.1, lines 2–3. The transaction invokes a stored procedure, which is executed utilizing the Tx key-value store. The stored procedure's execution updates any number of key-value pairs by updating, creating, or deleting them (line 5). A sample stored procedure that may be invoked by a transaction is shown in Listing 5.1.

After a transaction completes its execution the primary replica's Tx engine determines if the transaction execution conflicts with the most up-to-date version of the key-value store. This is done by the Tx engine on the primary replica preventing new versions of the committed key-value store from being added (line 6). The primary replica's Tx engine checks if any of the keys accessed in the Tx key-value store had their version increased in the committed key-value store (lines 8–9) since the Tx key-value store was created as a copy of the committed key-value

Algorithm 5.2: Create execution groups (where $\mathcal{B}$ is the ordered list of transactions (tx) within a batch)

```
1  on create_execution_group($\mathcal{B}$, $\mathcal{G}$)
2      $\mathcal{C} \leftarrow \{\}$
3      $start\_tx\_id =$ get_first_tx_id($\mathcal{B}$)
4      foreach $tx \in \mathcal{B}$ do
5          $tx\_id =$get_tx_id($tx$)
6          $kv\_seen\_tx\_id =$ get_max_seen_tx_id($tx$)
7          if $start\_tx\_id \geq kv\_seen\_tx\_id$ then
8              $\mathcal{G} = \mathcal{G} + \{C\}$
9              $\mathcal{C} \leftarrow \{\}$
10             $start\_tx\_id = tx\_id$
11         $\mathcal{C} \leftarrow \mathcal{C} + tx\_id$
12     $\mathcal{G} = \mathcal{G} + \{C\}$
```

store. If the Tx engine discovers that the committed key-value store and the Tx key-value store updated the same keys, the transaction is considered to have a conflict (line 11). The conflicted transaction is then re-executed with a new copy of the Tx key-value store. This process is repeated until there is no longer a conflict. If there are no conflicts, the transaction execution has been completed successfully (line 13).

A transaction that is determined to not conflict with the most recent version of the committed key-value store's state is ordered within the batch, and the key-value store updates made by the transaction are reflected in the committed key-value store. The primary replica's Tx engine is responsible for setting the order of transactions within the batch which the Tx engine does by assigning a monotonically increasing number (tx_id) to the committed transaction. Next, the Tx engine updates the key-value store using the committed transaction's Tx key-value store. The Tx engine iterates through the transaction's updated key-value pairs and updates the last accessed parameter for each key-value pair in the committed key-value store with the tx_id of the transaction. If the transaction committed successfully, i.e., it did not abort, the values of the key-value pair in the committed key-value store are also updated along with the version of the key. After the updates to the committed key-value are complete, other updates to the committed key-value store are no longer prevented.

Execution groups. After all the transaction requests within a batch have been executed, the primary replica's Tx engine creates groups of transactions that the Tx engine on the backup replicas can execute in parallel (multiple threads). Algorithm 5.2 shows how the groups of transactions, within a batch, which can be executed in parallel, are created. The process of creating groups of transactions begins by iterating through the transaction (tx) within a batch ($\mathcal{B}$) and is shown on line 4. When considering a transaction the Tx engine calls `get_max_seen_tx_id` to determine the highest *tx_id* (*kv_seen_tx_id*) of the last transaction to access the key-value pairs read, written, or updated by transaction *tx* (line 6). *kv_seen_tx_id* is then compared to the *start_tx_id* (line 3) to determine if one of the transactions (*tx*) depends on a key-value pair created by a transaction in the current group $\mathcal{C}$ (line 7). If transaction *tx* does depend on a transaction in the current group the transactions in the current group $\mathcal{C}$ are added to a list of execution groups $\mathcal{G}$ and the current group $\mathcal{C}$ is reset (lines 8–10). Then, regardless if the execution group is reset or not, *tx* is added to the current group $\mathcal{C}$, which may at this point be empty (line 11). The above iteration continues for every transaction in $\mathcal{B}$.

After creating the transaction groups $\mathcal{G}$, the Tx engine on the primary replica extends L-PBFT's pre-prepare message by including the hash of $\mathcal{G}$. The primary replica's Tx engine then sends the extended pre-prepare message and G to the Tx engines on the backup replicas.

5.4.3 Backup execution

After a backup replica receives a batch of transaction requests, which include the primary replica Tx engine's assertion of which transaction requests can be executed in parallel without compromising the system's linearizability, the backup replica's Tx engine executes the transactions. During execution, the backup replica's Tx engine verifies that executing transactions in their execution groups does not violate linearizability.

Batch execution. When the backup replica's Tx engine receives a batch of transaction requests, it executes them in a similar manner to the primary replica's Tx engine. The transaction groups are executed in the order specified by the primary replica's Tx engine and all the transactions from a transaction group must have been committed before the next transactions in the next transaction group begin executing. The transaction on the backup replicas follows the same pattern as the primary replica in that, each time a transaction is created, it is assigned a copy of the key-value store and all access by the transaction is to the copied key-value (transaction key-value store).

The backup replica's Tx engine monitors transaction request execution to ensure that transactions commit or abort. If a transaction execution results in the transaction key-value store conflicting with the committed key-value store, the Tx engine considers the execution groups sent by the primary replica's Tx engine to be malicious. This malicious behaviour results in the consensus protocol on the backup replica utilizing L-PBFT's view-change protocol to change the primary replica.

Verifying execution groups. After all the transaction requests within a batch have executed, the backup replica's Tx engine verifies that the execution groups sent by the primary replica's Tx engine do not create a linearizability violation. Algorithm 5.3 shows how the execution groups associated with a batch are validated to be correct.

Algorithm 5.3: Verify an execution group is valid

```
on verify_execution_group($\mathcal{G}$)
    foreach $\mathcal{C} \in \mathcal{G}$ do
        $start\_tx\_id$ = get_first_tx_id($\mathcal{G}$)
        foreach $tx \in \mathcal{C}$ do
            $kv\_seen\_tx\_id$ = get_max_seen_tx_id($tx$)
            if $kv\_seen\_tx\_id \geq start\_tx\_id$ then
                return $false$
    return $true$
```

The Tx engine begins verifying that the transaction groups are correct by iterating through all the transactions within the batch (lines 2, 4). The iteration occurs in the order in which the primary replica's Tx engine committed the transaction requests. While iterating through the transactions the backup replica's Tx engine considers each group (line 4) and determines *start_tx_id*, the tx_id of the first transaction in the group (line 3). The tx_id of the first transaction of the execution group is compared to the highest *tx_id* (*kv_seen_tx_id*) of the transaction, which was the last to access the key-value pairs read, updated, or modified during the execution of the transaction *tx* (lines 5–6). If *tx* does access a key that is also accessed by another transaction within the same execution group, i.e., $kv\_seen\_tx\_id \geq start\_tx\_id$, then the execution groups were created by a dishonest primary replica (line 7). Otherwise, the Tx engine continues to iterate through the transactions, ensuring that they do not access a key that was accessed by a transaction within the same transaction group.

Upon not finding any conflicts, the backup replica's Tx engine informs the replica's consensus protocol that the pre-prepare message was executed correctly. The consensus protocol is then given the responsibility to commit the batch described in the pre-prepare message sent by the Primary replica's Tx engine.

5.4.4 Client response

Bunyip's consensus protocol is an extension of L-PBFT (see Section 3.3.1). The consensus protocol's responsibility is to order and execute requests, create a ledger, and ensure liveness when there are f or fewer dishonest replicas. However, unlike in IA-CCF, Bunyip's consensus protocol does not directly respond to the client. Rather, Bunyip utilises the client proxy as the communication endpoint with the client. The client proxy enables clients to use common tools, such as Curl, to send transaction requests to Bunyip, a common request by potential users.

Ordering requests. After the backup replica's Tx engine completes executing the transaction requests within a batch and ensures that the execution matches that of the primary replica, the Tx engine passes the batch information to the consensus protocol. The consensus protocol on the backup replicas generated prepare messages with the same format as the prepare messages generated by IA-CCF replicas (see Section 3.3.1). The backup replica's consensus protocol, which is an implementation of the L-PBFT consensus protocol, sends the prepare messages to the consensus protocol on all other replicas.

When a replica's consensus protocol has received a pre-prepare message and at least $N - f - 1$ prepare messages for a batch (where one of the messages was created by the replica), the consensus protocol considers itself to have prepared for the sequence number and view of the batch (see Section 3.3.1). The consensus protocol of a prepared replica creates a commit message with the sequence number and view of prepared batch.

The replica's consensus protocol then sends the commit message to all other replicas. After a replica has created its own commit message for a batch and received commit messages from $N - f - 1$ distinct replicas for the same batch, the batch is considered to be committed. After the batch has committed and if the replica is the primary, its Tx engine is ready to create the next batch. Alternatively, the Tx engine on the backup replicas can start executing the next pending batch.

Client proxy. The reply messages follow a similar pattern to L-PBFT, however, they are not directly sent to the client. A Bunyip replica's consensus protocol creates a reply message at the same time as it creates a commit message. The consensus protocol sends the reply or replyx message to the client proxy on the replica that received the request message from the client.

After receiving a replyx message and $N - f$ reply messages, the client proxy converts the messages into a receipt (see Section 3.3.3). The receipt and the payload from one of the reply messages are sent to the client by the client proxy. The client is responsible for verifying the receipt by ensuring that it is signed by $N - f$ valid replicas before acting on the information within the payload of the reply message.

5.4.5 Discussion

Linearizability. Bunyip modifications to the L-PBFT consensus protocol and parallel execution do not affect the consensus protocol's linearizability property. As described in this section, parallel transaction execution provides the same linearizability property as sequential transaction execution.

In addition, parallel execution requires that the pre-prepare message includes a single additional hash. This cannot affect linearizability. Thus, the modifications to L-PBFT do not affect the linearizability property.

View changes. Bunyip implements the same view-change protocol as L-PBFT, maintaining the same safety and liveness properties. The Bunyip consensus protocol only modifies how transaction requests are executed. Both protocols do not execute transaction requests in parallel to performing view change operations and both protocols delay any view changes related to decisions while transaction requests from a batch are being executed. We postulate that the delay introduced from a replica waiting for transaction execution to complete is not significant to users. The delay would be several orders of magnitude smaller than the time required for a view-change to complete, during which no new transactions are executed.

Opacity. Transactions in Bunyip provide snapshot isolation irrespective if the transaction commits, aborts, or conflicts with another transaction. This property is known as opacity [104]. Bunyip provides opacity, because the Tx engine makes a copy of the most recently committed key-value store for every transaction. The copy of the key-value store is only read or updated within the context of a single transaction.

Dishonest client proxy. When a malicious client proxy receives a transaction request from a client, it either: (i) forwards the transaction request to a subset of the replicas, and if one of the replicas is honest the replica will forward the transaction request to all the other replicas [48, 57]; or (ii) does not forward the transaction request, thus not returning a valid receipt to the client. After a timeout, the client sends the transaction request to another client proxy. After at most $f+1$ attempts, the client will find an honest client proxy.

This defense against a dishonest client proxy is the same strategy used to protect against an untrustworthy inference proxy (see Section 4.5.5).

Optimizations. In this section, we presented the parallel execution protocol. However, there are several optimizations that may reduce the number of transactions that the protocol considers to conflict and must be placed into separate execution groups.

Last access. The parallel execution protocol described in Sections 5.4.2 and 5.4.3 extends the IA-CCF key-value store to track the identifier of the last transaction to access a key. Tracking only the last accessed transaction results in two transactions that read but do not update the same key-value pair to be considered conflicting. Thus, an optimization to parallel execution is for every key-value pair to track the transactions that last read and update the value at a key.

Execution threads. Bunyip allows for transactions to execute on multiple threads. If the restriction is made that a transaction executes on a statically allocated number of threads it is possible to implement the following optimization. Each transaction request is associated with a statically allocated thread ID and both the primary and backup replicas execute the transaction requests that are assigned the same thread ID on the same physical thread. Ensuring transactions execute on the same thread on both the primary and backup replicas means that two transactions that execute on the same thread cannot conflict with one another.

This optimization is implemented by tracking which threads last updated a key-value pair

Table 5.2: Size of ledger entries (SmallBank and TPC-C)

Ledger entry type	Size (bytes) f = 1	f = 3
Transaction (SmallBank)	216–358	
Transaction (TPC-C)	224–356	
Pre-prepare	285	
Prepare Evidence	298	995
Nonces	32	64

in the key-value store. If a transaction updates a key-value pair that was last updated by a transaction executing on the same thread, Bunyip would not consider there to be a conflict, and the two transactions would not need to be executed in different execution groups.

5.5 Evaluation

We evaluate Bunyip to understand the impact of parallelizing transaction execution (Section 5.5.1), the impact of transaction complexity on parallelizing transaction execution (Section 5.5.2), its scalability (Section 5.5.3), and the network overhead of sending transaction execution groups (Section 5.5.3).

Testbeds. Our experimental setup consists of one environment: a dedicated cluster with 16 machines, each with an 8-core 3.7-Ghz Intel E-2288G CPU with 16 GB of RAM and a 40 Gbps network with full bi-section bandwidth. All machines run Ubuntu Linux 18.04.4 LTS.

Implementation. Our Bunyip prototype is based on IA-CCF, which is based on CCF v0.13.2 [169]. Our prototype was implemented in approximately 1000 lines of C++ code. It uses the formally-verified Merkle trees and SHA functions of EverCrypt [197], the MbedTLS library [161] for client connections, and secp256k1 [250] for all secure signatures. Replicas create secure communication channels using a Diffie–Hellman key exchange.

Pipelining batch execution (P in Algorithm 3.1) improves Bunyip's throughput. We use P=2 with maximum batch sizes of 300 requests, respectively. Checkpoints are created every 100,000 batches.

Benchmarks. We use two benchmarks in our evaluation:

SmallBank [7] models a bank with 500,000 customer accounts. Clients randomly execute 5 transaction types: deposit, transfer, and withdraw funds; check account balances; and amalgamate accounts. The size of the ledger entries is shown in Table 5.2 where only the Prepare Evidence and Nonces entries depend on f.

TPC-C [62] models a wholesale supplier in a generic industry that manages, sells, or distributes

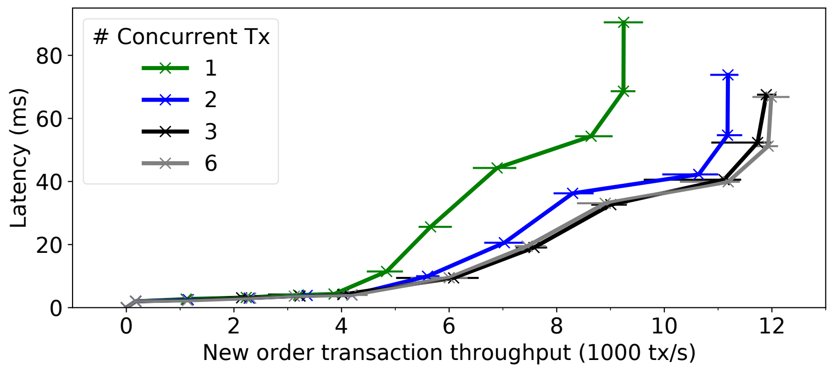


Figure 5.2: Executed new order transaction per second throughput/latency (f=1, TPC-C)

one or more products. We built our implementation of the TPC-C benchmarks by modifying the Implementation by Evan Jones [130]. We use a database with 10 warehouses to facilitate a larger number of conflicting transactions. We report "new order" transactions which represent approximately 45% of the transaction mix as specified by the benchmark. We run the full mix and report on the performance of the successfully committed "new order" transactions.

Since Bunyip's does not change IA-CCF view change protocol, we omit results from experiments with failures.

Transaction throughput is measured at the primary replica and latency at the clients. All experiments are compute-bound. Results are averaged over 5 runs, with min/max error bars.

5.5.1 Transaction throughput

We begin by evaluating the impact on throughput and latency of executing transactions in parallel. We look at 4 configurations of Bunyip where we limit the maximum number of transactions that can be executed in parallel. The 4 configurations that are considered allow 1, 2, 3, or 6 transactions to execute in parallel, and the results are shown in Figure 5.3. All 4 configurations execute the TPC-C benchmark and the experiments are executed on our testbed.

We start by looking at our baseline. When at most 1 transaction is allowed to execute in parallel, our implementation of Bunyip is equivalent to our implementation of IA-CCF. The baseline configuration obtains a peak throughput of 9,245 executed new order transactions per second.

Next, we increase the number of hardware threads that may execute transactions within a transaction group. When 2 threads are available the peak throughput increases approximately 17% to 11,183 new order transactions per second. With 3 threads available to execute transactions, throughput increases to 11,896, a further 6% increase over executing with 2 available transactions.

We observe that, beyond 3 threads, the addition of extra threads yields minimal to no

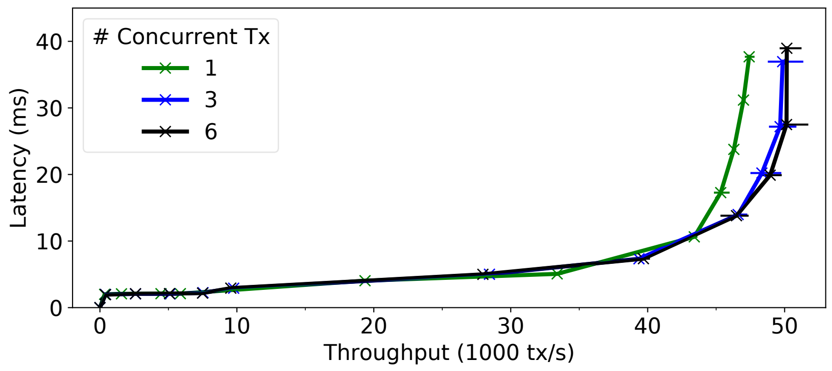


Figure 5.3: Transaction throughput/latency (f=1, SmallBank)

gain in transaction throughput. As such, when 6 threads are available, there is only a 1% throughput increase from 3 threads.

This shows that allowing both the primary and backup replicas to execute transactions in parallel provides a considerable improvement in performance. We can see that throughput improves over 23% when 6 threads are available to execute transactions as compared to our baseline of IA-CCF.

5.5.2 Transaction execution complexity

Next, we consider the impact of the total execution time of a transaction request and how this affects the benefit of parallel transaction execution. We compare the execution of the longer running OLTP TPC-C transactions to the short execution of SmallBank transactions.

Figure 5.3 shows the results when running Bunyip with the SmallBank benchmark. In this experiment, we consider 3 configurations that allow 1, 3, or 6 transactions to execute in parallel.

We start by considering our baseline, when at most 1 transaction is allowed to execute in parallel. The baseline configuration obtains a peak throughput of 47,411 transactions per second. In comparison, when 3 threads are available, the peak throughput is 49,859 transactions per second and, with 6 threads, the peak throughput is 50,160 transactions per second.

We now compare the transaction execution throughput when running SmallBank to the computationally more expensive transaction execution of TPC-C (Section 5.5.1). When running the SmallBank benchmark, we increase the number of available threads from 1 to 6 and observe that the number of executed transactions increases by 6%. When comparing this result to TPC-C, the increase of transaction execution is 23% when the available threads changes from 1 to 6. We conclude that parallel execution provides greater benefits when the system executes computationally expensive transactions.

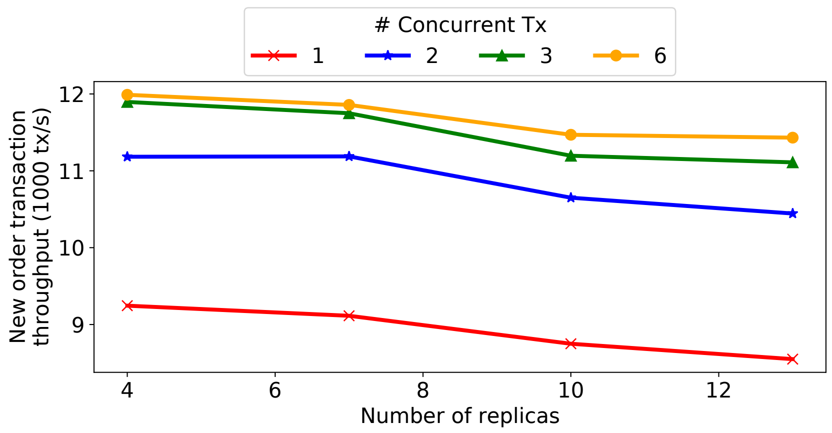


Figure 5.4: New order transaction throughput vs replica count (TPC-C)

5.5.3 Scalability

Now, we consider if the throughput improvement shown in Section 5.5.1 holds when the number of replicas onto which an instance of Bunyip has deployed increases. In this experiment, we use the TPC-C benchmark and consider 4 configurations that allow 1, 2, 3, or 6 transactions to execute in parallel on deployments where f ranges from 1 to 4.

The results are shown in Figure 5.4. We observe that, when f is 1, there is a 23% performance improvement when increasing the number of threads on which transactions can be executed from 1 to 6 (see Section 5.5.1). When the instance of Bunyip is run with f set to 4, this ratio becomes 25%. This improvement is explained by the variance in the results produced on our experimental setup.

Thus, we conclude that parallel execution on both the primary and backup replicas improves throughput, even as we increase the number of replicas.

5.5.4 Network overhead

Finally, we attempt to understand the network overhead created by sending the execution group information by the primary replica to the backup replicas. The primary replica sends execute group information as an array of 16-bit integers. Such that, every integer in the execution group array is an index into the ordered list of the hashes of the transaction requests sent by the primary replica to the backup replicas (see $\mathcal{B}$ in Algorithm 3.1). This means that, when no transactions conflict, there are no entries in the array and the maximum number of entries in the array is the number of entries in $\mathcal{B}$.

When executing TPC-C and SmallBank with a client load that obtains peak throughput, Bunyip observes that, on average, 4.5% and 0.1% of all transaction requests within a batch conflict, respectively. This results in the primary replica sending the transaction group informa-

Table 5.3: Size of execution group array with a maximum batch size of 300 transaction requests

Conflict rate (workload)	Size (bytes)
0%	0
50%	298
100%	598
4.5% (TPC-C)	26
0.1% (SmallBank)	2

tion $\mathcal{G}$ (see Section 5.4.2) of 26 bytes and 2 bytes for the TPC-C and the SmallBank workload, respectively. This is minimal overhead for both the LAN and WAN scenarios.

5.6 Related work

Sharding. OmniLedger [140] creates multiple shards that store user data. The system allows clients to act as transaction coordinators when executing transactions across multiple shards. However, OmniLedger only allows users to execute transactions that transfer coins (UTXO), and a client can only coordinate a transaction where the client transfers their own funds. Basil [226] similarly requires clients to act as coordinators but allows clients to execute general purpose ACID transactions. Both of these systems pass the computationally expensive transaction execution process to the clients.

Chainspace [6] and RapidChain [258] allocate multiple shards which both store data and execute transactions. The systems require multiple rounds of the BFT consensus protocols to commit a transaction. The replicas that spend a considerable amount of time executing transactions could be optimized with the parallel execution techniques presented in this design.

Oracle based multi-threading. All about Eve [135] and CBASE [142] require the user to write an oracle that predicts which transactions can and cannot be executed in parallel. If the oracle service utilized by CBASE incorrectly states that 2 transactions can be executed in parallel, system linearizability may be compromised. Alternatively, Eve is able to detect if two transactions are incorrectly executed in parallel. In such a scenario, Eve re-executes the transactions sequentially. Both systems require that a client knows if two transactions' execution would conflict before the transaction is executed. These types of predictions are often inaccurate.

Crash fault multi-threading. Rex [109] introduces an automated way of tracking transaction dependencies. However, it only considers crash fault failures and does not provide a technique to detect if the transaction dependency information sent by the primary is correct and honest.

Multi-threading with transaction re-execution. Block-STM allows for replicas to execute transactions in parallel and relies on aborting transactions and re-executing them when they do not commit in the total ordered provided by the primary. While many techniques are employed to reduce the number of times a transaction is re-executed when a large number of transaction execution conflicts occur, a significant performance overhead is introduced by this technique. The authors of Block-STM report that they observed a 30% increase in latency when executing transactions that have a high rate of contention when compared to executing the same transactions sequentially.

5.7 Summary

This chapter presented Bunyip, the first permissioned ledger that utilises a Byzantine consensus protocol that allows parallel execution of transaction requests on both the primary and backup replicas.

We first presented the motivation for executing requests in parallel on the primary and backup replicas. Next, we described the Bunyip API. We started by describing the steps required for a client or member to discover the location of the Bunyip replicas and their keys. Then we described how a client or member invokes a stored procedure by sending a request message to a client proxy on one of the replicas.

We continued to explore the system's API by looking at how stored procedures are written and then described a sample stored procedure. We concluded the exploration of the Bunyip API with a discussion of the importance of opacity when writing Bunyip stored procedures.

We next introduced the design of Bunyip. We started with an overview, exploring all the major system components. We then explained how the primary executes requests and constructs groups of transaction requests that can be executed in parallel.

After that, we covered how the backup replica executes the transaction requests in parallel using the transaction group information provided by the primary. We completed our exploration of parallel execution on the backup replica by looking at the algorithm that is used to verify that the parallel execution on a backup replica did not potentially compromise the system's linearizability.

Finally, we explored the performance impact of adding parallel execution to both the primary and backup replicas. We first considered the throughput and latency of transaction execution when running computationally expensive workloads with an increasing number of threads executing transaction requests. We continued exploring transaction execution throughput and latency by executing a computationally inexpensive workload. We then looked at the impact on transaction throughput when increasing the number of replicas within a Bunyip deployment. We finished our performance evaluation by exploring the network overhead on the primary replica and creating transaction execution groups.

CHAPTER 6

Conclusions

Permissioned ledgers have become the best choice for building trustworthy distributed systems. They bring together several key technologies, which are missing in other less secure distributed systems. Permissioned ledgers allow multiple organizations to come together and perform mutually verifiable computations where the organizations verify each other's work to reach an agreement on transaction execution. Permissioned ledgers can guarantee liveness and safety when a limited number of organizations hosting replicas misbehave. In addition, they provide a persistent ledger that is replicated across multiple replicas controlled by multiple organizations.

The trustworthiness of permissioned ledgers extends further than just tolerating a limiting number of misbehaving replicas. Permissioned ledgers also support auditing. Auditing allows a user to replay the ledger and detect if a number of the replicas misbehaved and compromised the safety of the ledger. Auditing is further enhanced by a permissioned ledger issuing client receipts, allowing clients to prove that their transaction was executed. When coupled together auditing the ledger and receipts ensures that a permissioned ledger becomes accountable when the system's safety property is compromised.

Unfortunately, permissioned ledgers are missing several key features. Accountability within the scope of a permissioned ledger is not able to act as a deterrent for misbehaviour, because it is possible to compromise more than a threshold of replicas to rewrite the ledger and hide misbehaviour. Thus, permissioned ledgers are susceptible to patient attackers. In addition, even if misbehaviour is detected in a large number of circumstances, it is not possible to know which replicas misbehaved.

Another common issue that arises when deploying permissioned ledgers in production environments is their performance when executing transactions. Permissioned ledgers commonly use Byzantine fault tolerant consensus protocols to order and execute transaction requests. Practical Byzantine fault tolerance was designed in an era of single-core CPUs where performance improvements could be obtained by the passage of time and Moore's law. Today's servers and applications depend on multi-core CPUs and other hardware accelerators to execute client workloads. The Byzantine consensus protocols used in permissioned ledgers are unable to take full advantage of these multi-core CPUs and accelerators.

6.1 Thesis summary

This thesis began with an introduction and motivation for building "Auditable and Performant Byzantine Consensus for Permissioned Ledgers". The first chapter was a brief history of public blockchains. After that, we introduced the consensus protocols used by public blockchains and explained how they began to include Byzantine fault tolerant consensus protocols. We continued the introduction by discussing the evolution of public blockchains and how they inspired permissioned ledgers and provided a discussion on the evolution of permissioned ledgers.

Chapter 2 provided the background on the research contributions of the thesis. In this chapter, we considered some seminal works and considered work from both academia and industry whose ideas and considerations had an impact on the content of this thesis. We further divided the chapter into three sections to complement the contribution of this thesis.

The background chapter first looked at Byzantine consensus protocols. We first explored Practical Byzantine fault tolerance (PBFT), the seminal work for Byzantine fault tolerant consensus protocols. We then surveyed Byzantine consensus protocols used by permissioned ledgers and then focused on notable permissioned ledgers and explored their consensus protocols, some of which cannot tolerate Byzantine faults.

Next, we looked at accountability in distributed systems and permissioned ledgers. As with Byzantine consensus protocols we began by discussing PeerReview, a seminal work in distributed systems accountability. We then considered accountability for permissioned ledgers — the accountability properties of well known permissioned ledgers and consensus protocols. Next, we examined recent work on adding accountability to newly designed permissioned ledgers.

In the final section of the background chapter, we described techniques that have been used to improve transaction execution in Byzantine consensus protocols, permissioned ledgers, and other state-machine replication protocols. In this background section, we focused on the sharding of Byzantine consensus protocols and how introducing independent shards can improve transaction throughput. Next, we looked at work that considered how database sharding techniques can be used to improve transaction throughput in a permissioned ledger. Finally, we considered systems that consider how a replica in a Byzantine consensus protocol can execute transactions in parallel. We observed that many systems depend on oracle services, which predict which transactions may conflict with one another. Finally, we discussed crash-fault tolerant systems, which allow parallel transaction execution on both the primary and backup replicas.

In Chapter 3, we explored accountability in a permissioned ledger and how members that run replicas within a permissioned ledger can always be made accountable for their replica's misbehaviour if a permissioned ledger's safety is compromised. To explore this problem space, we introduced IA-CCF, a permissioned ledger that provides accountability regardless of the number of replicas that misbehave. Our goal in building IA-CCF was to provide a platform to explore permissioned ledger accountability and understand the performance impact of accountability.

We started our exploration of accountability in permissioned ledgers by specifying the requirements of such a system. The key requirement we identified for accountability to be a deterrent for misbehaviour is that at least $f + 1$ replicas must be identifiable by an auditor if the permissioned ledger's safety was compromised. This led to our secondary requirement: we must support accountability regardless of how many replicas misbehave and replicas and members can be added or removed from the permissioned ledger.

We presented our research contribution by looking at L-PBFT, the Byzantine consensus protocol used by IA-CCF. We described the consensus protocol, the protocol messages, and how L-PBFT builds the ledger and generates receipts that are sent to the IA-CCF users. In addition, we explained the changes L-PBFT makes to view-changes and a series of optimization designed to ensure that IA-CCF has high transaction throughput.

After we described the consensus protocol and other components in IA-CCF we moved our focus to how the IA-CCF ledger can be audited. We explained how it is possible to audit the ledger regardless of the number of replicas that misbehave. After this, we showed how an external enforcement entity could be provided with proof of which specific replicas misbehaved so that the members that control those replicas can be held accountable. We continued this exploration of auditing by describing a reconfiguration protocol and how the aforementioned accountability properties continued to hold. In addition, we provided a proof that the described auditability properties are enforced with and without governance in Appendix B.

We concluded the chapter with a performance evaluation of IA-CCF. We demonstrated that our prototype obtains high throughput and low latency, and, in comparison with related work, IA-CCF obtained higher transaction throughput with lower transaction request latency when measured from the perspective of the client.

In the previous chapter, we showed how to build an auditable permissioned ledger that has a high transaction throughput. However, the workloads with which IA-CCF and other permissioned ledgers are benchmarked are not representatives of real-world workloads. Thus, this thesis focused on making auditable permissioned ledgers performant for real-world workloads.

Chapter 4 considered the execution of workloads that require the assistance of a hardware accelerator to obtain high throughput. We focused on deep neural network (DNN) inference workloads that are executed with a GPU. We realized our ideas as part of the DropBear system, which provides trustworthy results for inference requests and obtains high performance by efficiently utilizing GPUs as part of transaction request execution in DropBear's Byzantine consensus protocol.

We began the chapter by providing an introduction to cloud based systems that execute inference requests on trained models and the trust issues that arise when a user does not control the infrastructure on which their inference requests are executed. The chapter continued to provide examples of workloads that require trustworthy inference followed by the threat model for DropBear.

After defining the threat model and providing examples of customers that would benefit from trustworthy inference, the chapter continued to defined trustworthy inference. We first described how trustworthy inference is best obtained by utilizing a Byzantine consensus protocol. Next, we provided a precise mathematical definition of trustworthy inference. We started to make this mathematical definition more practical by describing the DropBear API and how it realizes the definition.

Next, we provided a system overview and described the execute-agree-attest execution model used to accelerate ML inference calculation in a permissioned ledger that implements a Byzantine consensus protocol. We described how the replicas first execute the inference request waiting for the GPU to return the inference results before scheduling the executed inference request for consensus protocol ordering. Critically, we discussed how separating execution and ordering allows DropBear to use different batch sizes for inference execution and the batching of protocol requests. We concluded the system description by explaining how inference certificates are generated, verified, and how they show the inference results and endorsements from $N-f$ nodes.

The chapter continued with an explanation, of how to audit the DropBear permissioned ledger and explored the permissioned ledger's performance. We looked at how DropBear throughput and latency compared to untrustworthy inference as a service systems. After consider several configurations of DropBear, we concluded the chapter by describing how the auditability ideas explored in IA-CCF are translated to DropBear.

This thesis then continued to explore different types of accelerators that are utilized by users. We moved our focus to how modern multi-core CPUs can be fully utilized by Byzantine fault tolerant consensus protocols employed by permissioned ledgers. We started our exploration by looking at the historical reasoning behind why CPUs with more than one core were not considered when Byzantine fault tolerant protocols were designed. Then, we considered the workloads that would benefit from utilizing multi-core CPUs

After a motivating example, we considered an API that exposes users to capabilities that allow parallel execution of requests, thus making the exploitation of the multi-core CPUs transparent to users. While exploring such an API, we saw that a key-value store can offer user opacity. We also observed that users would not need to define which transactions can execute in parallel.

Next, we looked at the design of Bunyip, specifically focusing on executing transaction requests in parallel on both the primary and backup replicas. First, we described the primary replica's execution and how groups of transactions that can be executed in parallel are formed. This was followed up with the description of backup replicas executing said groups and validating that the parallel execution could not have resulted in the system violating its linearizability property.

We concluded the chapter with the evaluation of the Bunyip prototype. First, we considered the throughput and latency of Bunyip transaction requests when running the TPC-C benchmark. This was followed by comparing TPC-C execution to SmallBank execution to understand the impact of transaction complexity on throughput when transactions are executed in parallel. Finally, we explained the effect of increasing the replica count and the amount of additional data that must be transmitted by the primary replica when it sends the parallel execution group information.

6.2 Future work

This thesis considers and contributes to several topics required for building "Auditable and Performant Byzantine Consensus for Permissioned Ledgers", but there is still more work that remains open for further study.

Ledger packing. In Chapter 3 of this thesis, we describe how a permissioned ledger can ensure that, through a combination of the ledger and receipts, misbehaviour can be detected and punished. However, it is possible for dishonest members or clients to make this impractical and affect the liveness of auditing. Auditing the ledger requires re-executing a number of transactions between two receipts and, if the members or clients execute an extremely large number of transactions between the two receipts, re-execution can become impractical.

There are several techniques that could address this issue: (a) introduce a mechanism where there is a cost to executing a transaction. This cost can be financial or require that a time interval has passed, for example, solving a proof-of-work challenge, however, a proof-of-work solution would raise numerous additional concerns (see 1.2.1); (b) have a bound on the number of transactions that can be executed within a time window. We leave the details of the solution for future work.

In addition to ledger packing Chapter 3, does not consider the performance of ledger auditing beyond sequentially reading and verifying the read content. One simple technique would be to distribute cryptographic operations across multiple CPU cores. In addition, it is possible to divide the ledger into sections, where each section exists between two checkpoints. This would allow for validating ledger subsections in parallel.

Alternative accelerators. In chapters 4 and 5, we explored and provided solutions to how a permissioned ledger can utilize the hardware resources of CPUs and GPUs. However, there are accelerators that have not been covered in this thesis, including FPGAs, and custom ASICs. We expect that the techniques explored in this these to generalize to other hardware accelerators, but, we leave testing this hypotheses for future work.

A limitation of DropBear is that we assume that it is possible to know all possible states used when executing a request before it is ordered. While this is not a limitation for machine learning inference workloads, for other workloads this could result in excessive computation. We leave exploring workloads that could be used with DropBear outside of machine learning inference to future work.

Forcing sequential transaction execution. In Chapter 5, the primary replica is responsible for selecting the order in which transactions are executed. A dishonest primary replica could select an ordering of transaction requests that decreases the number of transactions that can be executed in parallel. One possible solution is to take away the ability of the primary replica to order transactions, as done in Pompē [262]. We leave this problem open for future work.

Resolving dishonest ledger state. In chapters 3, 4, and 5, we explored and discussed how clients, members, and the auditor participates in uncovering and assigning blame to dishonest replicas. However, this thesis does not address the process and techniques required to bring the ledger to a *healthy* state that reflects the assigned punishments after dishonest behaviour is discovered. We leave the problem of modifying the ledger, so that it returns to a *healthy* state, to future work.

Bibliography

[1] A30 tensor core GPU for AI inference — NVIDIA. `https://www.nvidia.com/en-gb/data-center/products/a30-gpu/`. (Accessed on 11/04/2021).

[2] Joe Abou Jaoude and Raafat George Saade. Blockchain applications – usage in different domains. *IEEE Access*, 2019.

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous Byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[4] Kareem S Aggour, Piero P Bonissone, William E Cheetham, and Richard P Messmer. Automating the underwriting of insurance applications. *AI magazine*, 2006.

[5] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR: Fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[6] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[7] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008.

[8] Amazon sagemaker - machine learning - amazon web services. `https://aws.amazon.com/sagemaker/`. (Accessed on 07/28/2021).

[9] Explore AWS AI and machine learning services. `https://aws.amazon.com/machine-learning/`. (Accessed on 07/28/2021).

[10] Zachary Amsden, R Arora, S Bano, M Baudet, S Blackshear, A Bothra, G Cabrera, C Catalini, K Chalkias, E Cheng, et al. The Libra blockchain. *https://developers.diem.com/papers/the-diem-blockchain/2020-05-26.pdf*, 2019.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[12] anzeel Akhtar. Louis vuitton, cartier, prada to use bespoke blockchain to tackle counterfeit goods. `https://www.coindesk.com/business/2021/04/20/louis-vuitton-cartier-prada-to-use-bespoke-blockchain-to-tackle-counterfeit-goods/`. (Accessed on 02/05/2022).

[13] Andrés Arévalo, Jaime Niño, German Hernández, and Javier Sandoval. High-frequency trading strategy based on deep neural networks. In *International conference on intelligent computing*. Springer, 2016.

[14] Ridhi Arora, Vipul Bansal, Himanshu Buckchash, Rahul Kumar, Vinodh J Sahayasheela, Narayanan Narayanan, Ganesh N Pandian, and Balasubramanian Raman. AI-based diagnosis of COVID-19 patients using x-ray scans with stochastic ensemble of CNNs. *Physical and Engineering Sciences in Medicine*, 2021.

[15] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018.

[16] J Aythora, R Burke-Agüero, A Chamayou, S Clebsch, M Costa, J Deutscher, N Earnshaw, L Ellis, P England, C Fournet, et al. Multi-stakeholder media provenance management to counter synthetic media risks in news publishing. In *Proc. Intl. Broadcasting Convention (IBC)*, 2020.

[17] J Aythora, R Burke-Agüero, A Chamayou, S Clebsch, M Costa, N Earnshaw, L Ellis, P England, C Fournet, M Gaylor, et al. Multi-stakeholder media provenance management to counter synthetic media risks in news publishing. In *Proc. International Broadcasting Convention (IBC)*, 2020.

[18] Adam Back. Hashcash. `http://www.cypherspace.org/hashcash/`, 1997. (Accessed on 01/30/2019).

[19] Vangelis Bacoyannis, Vacslav Glukhov, Tom Jin, Jonathan Kochems, and Doo Re Song. Idiosyncrasies and challenges of data driven learning in electronic trading. *arXiv preprint arXiv:1811.09549*, 2018.

[20] Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open neural network exchange. `https://github.com/onnx/onnx`.

[21] Arnab Banerjee. Blockchain technology: supply chain insights from ERP. In *Advances in computers*. Elsevier, 2018.

[22] Bank of England. Central bank digital currencies. `https://www.bankofengland.co.uk/research/digital-currencies`. (Accessed on 02/05/2022).

[23] American bar association. Arbitration. `https://www.americanbar.org/groups/dispute_resolution/resources/DisputeResolutionProcesses/arbitration/`. (Accessed on 03/27/2021).

[24] American bar association. How courts work. `https://www.americanbar.org/groups/public_education/resources/law_related_education_network/how_courts_work/discovery/`. (Accessed on 03/27/2021).

[25] Jeff Barr, Attila Narin, and Jinesh Varia. Building fault-tolerant applications on AWS. *Amazon Web Services*, 2011.

[26] John M Bates and Clive WJ Granger. The combination of forecasts. *Journal of the Operational Research Society*, 1969.

[27] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[28] Thomas J Beckman. AI in the IRS. In *1989 The Annual AI Systems in Government Conference*. IEEE Computer Society, 1989.

[29] johannes behl, tobias distler, and rüdiger kapitza. Hybrids on steroids: SGX-based high performance BFT. In *proceedings of the twelfth European conference on computer systems*, pages 222–237, 2017.

[30] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, New York, NY, USA, 2017. ACM.

[31] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[32] Bitcoin. Block size increase to 2mb. `https://github.com/bitcoin/bips/blob/master/bip-0101.mediawiki`. (Accessed on 04/28/2019).

[33] Bitcoin. Increase maximum block size. `https://github.com/bitcoin/bips/blob/master/bip-0101.mediawiki`. (Accessed on 04/28/2019).

[34] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[35] blockchain.com. Blockchain explorer. `https://www.blockchain.com/charts/transactions-per-second`. (Accessed on 02/05/2022).

[36] Bloomberg. Betting on blockchain: $50 trillion supply chain industry bets on digital tech. `https://www.bloomberg.com/news/videos/2022-01-26/the-50-trillion-industry-making-a-huge-bet-on-blockchain`. (Accessed on 02/05/2022).

[37] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001.

[38] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. *Cryptology ePrint Archive*, 2014.

[39] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, The University of Guelph, 2016.

[40] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. *arXiv preprint arXiv:2108.04575*, 2021.

[41] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[42] Carole Cadwalladr. Another huge data breach, another stony silence from facebook. `https://www.theguardian.com/technology/2021/apr/11/another-huge-data-breach-another-stony-silence-from-facebook`. (Accessed on 05/04/2021).

[43] Ivan Camilleri. Blockchain in compliance. `https://www2.deloitte.com/mt/en/pages/risk/articles/mt-blockchain-in-compliance.html`. (Accessed on 05/08/2023).

[44] Longbing Cao. AI in finance: A review. *Available at SSRN 3647625*, 2020.

[45] Capital one innovator. `https://aws.amazon.com/solutions/case-studies/capital-one/`. (Accessed on 11/17/2021).

[46] Lemuria Carter and Jolien Ubacht. Blockchain applications in government. In *Proceedings of the 19th Annual International Conference on Digital Government Research: governance in the data age*, 2018.

[47] Miguel Castro. personal communication.

[48] Miguel Castro. Practical Byzantine fault tolerance, 2001.

[49] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, Berkeley, CA, USA, 1999. USENIX Association.

[50] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[51] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*. Springer, 1983.

[52] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[53] 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 contract execution. *arXiv preprint arXiv:1804.05141*, 2018.

[54] Michael Chui, Bryce Hall, Helen Mayhew, Alex Singla, and Alex Sukharevsky. The state of ai in 2022—and a half decade in review — mckinsey. `https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai-in-2022-and-a-half-decade-in-review`. (Accessed on 05/08/2023).

[55] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019.

[56] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. BFT: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008.

[57] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[58] Emily Cochrane. Justice department settles with tea party groups after I.R.S. scrutiny. `https://www.nytimes.com/2017/10/26/us/politics/irs-tea-party-lawsuit-settlement.html`. (Accessed on 08/17/2021).

[59] Cary Coglianese and Lavi Ben Dor. AI in adjudication and administration, 2020.

[60] ConsenSys. Blockchain in government and the public sector. https://consensys.net/blockchain-use-cases/government-and-the-public-sector/. (Accessed on 02/05/2022).

[61] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016.

[62] The Transaction Processing Council. Tpc-c benchmark (revision 5.8.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2006.

[63] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless Byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018.

[64] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017.

[65] curl. https://curl.se/. (Accessed on 08/21/2021).

[66] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[67] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient BFT consensus. *arXiv preprint arXiv:2105.11827*, 2021.

[68] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, New York, NY, USA, 2019. ACM.

[69] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[70] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in Neural Information Processing Systems*, 2020.

[71] Manoj Kumar Dash, Gayatri Panda, Anil Kumar, and Sunil Luthra. Applications of blockchain in government education sector: a comprehensive review and future research potentials. *Journal of Global Operations and Strategic Sourcing*, 2022.

[72] Deepesh Data, Linqi Song, and Suhas N Diggavi. Data encoding for Byzantine-resilient distributed optimization. *IEEE Transactions on Information Theory*, 2020.

[73] Gergely Debreczeni. Neural network exchange format, 2019.

[74] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009.

[75] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May, 1996.

[76] ONNX Runtime developers. ONNX runtime. `https://www.onnxruntime.ai`, 2021.

[77] Diem association. An independent membership organization. `https://diem.com/en-US/association/`. Accessed: 2021-03-19.

[78] Thomas G Dietterich et al. Ensemble learning. *The handbook of brain theory and neural networks*, 2002.

[79] Qingyun Duan, Newsha K Ajami, Xiaogang Gao, and Soroosh Sorooshian. Multi-model ensemble hydrologic prediction using bayesian model averaging. *Advances in Water Resources*, 2007.

[80] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 1988.

[81] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*. Springer, 1992.

[82] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*. Springer, 2015.

[83] Michael Eischer and Tobias Distler. Egalitarian Byzantine fault tolerance. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2021.

[84] Yospeh Elkaim. Blockchain and regulatory compliance: A match made in heaven. `https://www.shieldfc.com/resources/blog/blockchain-and-regulatory-compliance-a-match-made-in-heavenor-is-the-honeymoon-over/`. (Accessed on 05/08/2023).

[85] Paul England, Henrique S Malvar, Eric Horvitz, Jack W Stokes, Cédric Fournet, Rebecca Burke-Aguero, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, John Deutscher, et al. Amp: Authentication of media via provenance. In *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021.

[86] What is Ethereum?, 2017. [Online; accessed 31. Jan. 2019].

[87] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016.

[88] Hyperledger Fabric. The ordering service —- hyperledger-fabric. `https://hyperledger-fabric.readthedocs.io/en/release-2.4/orderer/ordering_service.html`. (Accessed on 02/11/2022).

[89] Directory persons – questions and answers. `https://www.fca.org.uk/publication/documents/directory-persons-questions-answers.pdf`, Feb 2021. (Accessed on 05/08/2023).

[90] Carina Federico and Travis Thompson. Do IRS computers dream about tax cheats? artificial intelligence and big data in tax enforcement and compliance. *J. TAX PRAC. & PROC*, 2019.

[91] Kristoffer Francisco and David Swanson. The supply chain has no clothes: Technology adoption of blockchain for supply chain transparency. *Logistics*, 2018.

[92] Gartner identifies four trends driving near-term artificial intelligence innovation. `https://www.gartner.com/en/newsroom/press-releases/2021-09-07-gartner-identifies-four-trends-driving-near-term-artificial-intelligence-innovation`. (Accessed on 11/04/2021).

[93] Seán Gauld, Franz von Ancoina, and Robert Stadler. The Burst Dymaxion.

[94] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

[95] Pat Gelsinger. Moore's law–the genius lives on. *IEEE Solid-State Circuits Society Newsletter*, 2006.

[96] Serenity Gibbonsm. 2023 business predictions as ai and automation rise in popularity. https://www.forbes.com/sites/serenitygibbons/2023/02/02/2023-business-predictions-as-ai-and-automation-rise-in-popularity/?sh=6e23b20a744b, Feb 2023. (Accessed on 05/08/2023).

[97] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.

[98] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[99] Dan Goodin. https://arstechnica.com/information-technology/2017/09/equifax-website-hack-exposes-data-for-143-million-us-consumers/. (Accessed on 05/04/2021).

[100] Cloud inference api. https://cloud.google.com/inference. (Accessed on 07/28/2021).

[101] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. *arXiv preprint arXiv:1901.00910*, 2019.

[102] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[103] gtf.org. Making decentralized economic policy. http://gtf.org/garzik/bitcoin/BIP100-blocksizechangeproposal.pdf. (Accessed on 04/28/2019).

[104] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[105] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in Byzantium. In *International Conference on Machine Learning*. PMLR, 2018.

[106] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019.

[107] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A

scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019.

[108] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020.

[109] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[110] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *OSDI*, 2010.

[111] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 2007.

[112] Kim Sundtoft Hald and Aseem Kinra. How the blockchain enables and constrains supply chain performance. *International Journal of Physical Distribution & Logistics Management*, 2019.

[113] Shanique Hall. How artificial intelligence is changing the insurance industry. *The Center for Insurance Policy & Research*, 2017.

[114] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers. *arXiv preprint arXiv:1906.11307*, 2019.

[115] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[116] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*. Springer, 2016.

[117] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, 2016.

[118] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990.

[119] Noel L Hillman. The use of artificial intelligence in gauging the risk of recidivism. *The Judges' Journal*, 2019.

[120] Chris Hourihan and Bryan Cline. A look back: US healthcare data breach trends. *Health Information Trust Alliance. Retrieved from https://hitrustalliance.net/content/uploads/2014/05/HITRUST-Report-US-Healthcare-Data-Breach-Trends.pdf*, 2012.

[121] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[122] Crypto Hustle. Krypton recovers from a new type of 51% network attack. `http://cryptohustle.com/krypton-recovers-from-a-new-type-of-51-network-attack/`. (Accessed on 12/06/2020).

[123] Daniel P Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence*, 1993.

[124] Hyperledger. Hyperledger Besu enterprise Ethereum client (Hyperledger Besu). `https://besu.hyperledger.org/en/stable/`. (Accessed on 12/06/2020).

[125] Introduction to Sawtooth PBFT – Hyperledger, 2019. [Online; accessed 23. May 2019].

[126] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[127] IBM. we.trade — ibm. `https://www.ibm.com/case-studies/wetrade-blockchain-fintech-trade-finance`. (Accessed on 05/04/2021).

[128] Intel. PoET 1.0 Specification – Sawtooth v1.0.5 documentation, 2018. [Online; accessed 9. Feb. 2019].

[129] Ashraf Jaradat, Omar Ali, and Ahmad AlAhmad. Blockchain technology: A fundamental overview. In *Blockchain Technologies for Sustainability*. Springer, 2022.

[130] Evan Jones. TPCCBench. `https://github.com/evanj/tpccbench`. (Accessed on 03/26/2022).

[131] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter

performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[132] FDIC Jul. Risk assessment tools and practices for information system security, 1999.

[133] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheap-BFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.

[134] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[135] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve:{Execute-Verify} replication for {Multi-Core} servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[136] Jayden Khakurel, Birgit Penzenstadler, Jari Porras, Antti Knutas, and Wenlu Zhang. The rise of artificial intelligence under the lens of sustainability. *Technologies*, 2018.

[137] Moon Kwon Kim and Soo Dong Kim. Inference-as-a-service: A situation inference service for context-aware computing. In *2014 International Conference on Smart Computing*. IEEE, 2014.

[138] Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 2012.

[139] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[140] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[141] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[142] Ramakrishna Kotla and Michael Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004.

[143] Mahtab Kouhizadeh, Sara Saberi, and Joseph Sarkis. Blockchain technology and the sustainable supply chain: Theoretically exploring adoption barriers. *International Journal of Production Economics*, 2021.

[144] C Krittanawong. The rise of artificial intelligence and the uncertain future for physicians. *European journal of internal medicine*, 2018.

[145] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 2012.

[146] Logan Kugler. AI judges and juries. *Communications of the ACM*, 2018.

[147] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[148] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[149] Consortium member definition: 387 samples. `https://www.lawinsider.com/dictionary/consortium-member`. (Accessed on 05/08/2023).

[150] June-Goo Lee, Sanghoon Jun, Young-Won Cho, Hyunna Lee, Guk Bae Kim, Joon Beom Seo, and Namkug Kim. Deep learning in medical imaging: general overview. *Korean journal of radiology*, 2017.

[151] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[152] Hemi Leibowitz, Ania M Piotrowska, George Danezis, and Amir Herzberg. No right to remain silent: Isolating malicious mixes. In *28th USENIX security symposium (USENIX security 19)*, 2019.

[153] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[154] Huiying Liang, Brian Y Tsui, Hao Ni, Carolina CS Valentim, Sally L Baxter, Guangjian Liu, Wenjia Cai, Daniel S Kermany, Xin Sun, Jiancong Chen, et al. Evaluation and accurate diagnoses of pediatric diseases using artificial intelligence. *Nature medicine*, 2019.

[155] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. *arXiv preprint arXiv:1612.07766*, 2016.

[156] Chang Liu, Stephen J Gardner, Ning Wen, Mohamed A Elshaikh, Farzan Siddiqui, Benjamin Movsas, and Indrin J Chetty. Automatic segmentation of the prostate on ct images using deep neural networks (DNN). *International Journal of Radiation Oncology* Biology* Physics*, 2019.

[157] Huiyan Luo, Guoliang Xu, Chaofeng Li, Longjun He, Linna Luo, Zixian Wang, Bingzhong Jing, Yishu Deng, Ying Jin, Yin Li, et al. Real-time artificial intelligence for detection of upper gastrointestinal cancer by endoscopy: a multicentre, case-control, diagnostic study. *The Lancet Oncology*, 2019.

[158] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[159] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 2011.

[160] Marc Maier, Hayley Carlotto, Freddie Sanchez, Sherriff Balogun, and Sears Merritt. Transforming underwriting in the life insurance industry. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[161] SSL Library mbed TLS / PolarSSL. `https://tls.mbed.org/`. (Accessed on 12/09/2020).

[162] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 1987.

[163] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 1987.

[164] microsoft/merklecpp at 9c92bdb396abe89b9d2796ae0f2093b5f75e17c5. `https://github.com/microsoft/merklecpp/tree/9c92bdb396abe89b9d2796ae0f2093b5f75e17c5`. (Accessed on 11/13/2021).

[165] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 1999.

[166] Azure machine learning - ml as a service. `https://azure.microsoft.com/en-us/services/machine-learning/#features`. (Accessed on 07/28/2021).

[167] Microsoft. Confidential ledger - distributed ledger technology. https://azure.microsoft.com/en-us/services/azure-confidential-ledger/. (Accessed on 02/04/2022).

[168] Microsoft. Microsoft/CCF: Confidential Consortium Framework. https://github.com/microsoft/ccf. (Accessed on 02/01/2022).

[169] Microsoft. Release ccf-0.13.2 · microsoft/ccf. https://github.com/microsoft/CCF/releases/tag/ccf-0.13.2. (Accessed on 01/13/2022).

[170] Thread and task architecture guide - sql server — microsoft docs. https://docs.microsoft.com/en-us/sql/relational-databases/thread-and-task-architecture-guide?view=sql-server-ver15. (Accessed on 04/11/2022).

[171] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[172] BFT project homepage. https://pmg.csail.mit.edu/bft/. (Accessed on 03/27/2022).

[173] blockchain performance issues and limitations. https://mixbytes.io/blog/blockchain-performance-issues-limitations#rec145461377, 2019 Jul. (Accessed on 05/08/2023).

[174] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017.

[175] Henrique Moniz. The Istanbul BFT consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.

[176] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[177] Takuya Nakaike, Qi Zhang, Yohei Ueda, Tatsushi Inagaki, and Moriyoshi Ohara. Hyperledger Fabric performance characterization and optimization using GoLevelDB benchmark. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020.

[178] Satoshi Nakamoto et al. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin.org/bitcoin. pdf*, 2008.

[179] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramaniam, and Murali Annavaram. Privacy-preserving inference in machine learning services using trusted execution environments. *arXiv preprint arXiv:1912.03485*, 2019.

[180] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramanian, and Murali Annavaram. Origami inference: Private inference using hardware enclaves. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021.

[181] Simone Natale and Andrea Ballatore. Imagining the thinking machine: Technological myths and the rise of artificial intelligence. *Convergence*, 2020.

[182] Netherlands: Court prohibits government's use of AI software to detect welfare fraud — library of congress. https://www.loc.gov/item/global-legal-monitor/2020-03-13/netherlands-court-prohibits-governments-use-of-ai-software-to-detect-welfare-fraud/. (Accessed on 10/24/2021).

[183] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

[184] Svein Ølnes, Jolien Ubacht, and Marijn Janssen. Blockchain in government: Benefits and implications of distributed ledger technology for information sharing, 2017.

[185] Diego Ongaro. *Consensus: Bridging theory and practice.* Stanford University, 2014.

[186] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, 2014.

[187] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[188] Github - ONNX/models: A collection of pre-trained, state-of-the-art models in the ONNX format. https://github.com/onnx/models. (Accessed on 10/24/2021).

[189] https://openauthentication.org. https://openauthentication.org/. (Accessed on 08/17/2021).

[190] Types of agreements – office of research and innovation. https://research.utdallas.edu/researchers/contracts/types-of-agreements. (Accessed on 05/08/2023).

[191] Panavia. Introduction. https://www.panavia.de/company/introduction/. (Accessed on 05/04/2021).

[192] Zachary Papanastasopoulos, Ravi K Samala, Heang-Ping Chan, Lubomir Hadjiiski, Chintana Paramagul, Mark A Helvie, and Colleen H Neal. Explainable AI for medical imaging: deep-learning CNN ensemble for classification of estrogen receptor status from breast mri. In *Medical Imaging 2020: Computer-Aided Diagnosis*. International Society for Optics and Photonics, 2020.

[193] David Parkins. The great chain of being sure about things - Blockchains. `https://www.economist.com/briefing/2015/10/31/the-great-chain-of-being-sure-about-things`, 2015. (Accessed on 01/30/2019).

[194] Radio Perlman. An overview of PKI trust models. *IEEE network*, 1999.

[195] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 2006.

[196] Calvin Price. Introduction to Quorum: Blockchain for the financial sector. `https://medium.com/blockchain-at-berkeley/introduction-to-quorum-blockchain-for-the-financial-sector-58813f84e88c`. (Accessed on 02/05/2022).

[197] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[198] Model zoo — PyTorch/Serve master documentation. `https://pytorch.org/serve/model_zoo.html`. (Accessed on 10/24/2021).

[199] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 18–34, 2021.

[200] Quorum. A permissioned implementation of Ethereum supporting data privacy. `https://github.com/ConsenSys/quorum`. Accessed: 2020-11-27.

[201] Alejandro Ranchal-Pedrosa and Vincent Gramoli. ZLB: A blockchain to tolerate colluding majorities. *arXiv preprint arXiv:2007.10541*, 2020.

[202] Sonia Rao. Matt damon, jimmy fallon and the celebrity-cryptocurrency industrial complex. `https://www.washingtonpost.com/arts-entertainment/2022/02/04/celebrities-cryptocurrency-nfts/`. (Accessed on 02/05/2022).

[203] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, 2021.

[204] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[205] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 2015.

[206] 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.

[207] Amit Sabne. XLA: Compiling machine learning for peak performance, 2020.

[208] Omer Sagi and Lior Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2018.

[209] Roberto Saltini and David Hyland-Wood. Correctness analysis of IBFT. *arXiv preprint arXiv:1901.07160*, 2019.

[210] Roberto Saltini and David Hyland-Wood. IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*, 2019.

[211] Abdurrashid Ibrahim Sanka and Ray CC Cheung. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications*, 195, 2021.

[212] Doruk Sardag. Market applications of blockchain with a focus on government and public sector. `https://readwrite.com/market-applications-of-blockchain-with-a-focus-on-government-and-public-sector/`. (Accessed on 02/05/2022).

[213] Scott Likens. Making sense of bitcoin and blockchain technology. `https://www.pwc.com/us/en/industries/financial-services/fintech/bitcoin-blockchain-cryptocurrency.html`. (Accessed on 02/05/2022).

[214] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, et al. {IA-CCF}: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[215] Alex Shamis, Peter Pietzuch, Antoine Delignat-Lavaud, Andrew Paverd, and Manuel Costa. Dropbear: Machine learning marketplaces made trustworthy with Byzantine model agreement. *arXiv preprint arXiv:2205.15757*, 2022.

[216] Sourabh Shastri, Kuljeet Singh, Sachin Kumar, Paramjit Kour, and Vibhakar Mansotra. Deep-lstm ensemble framework to forecast COVID-19: an insight to the global pandemic. *International Journal of Information Technology*, 2021.

[217] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. *arXiv preprint arXiv:2010.06785*, 2020.

[218] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[219] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[220] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000.

[221] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[222] Rebecca Smith-Bindman, Marilyn L Kwan, Emily C Marlow, Mary Kay Theis, Wesley Bolch, Stephanie Y Cheng, Erin JA Bowles, James R Duncan, Robert T Greenlee, Lawrence H Kushi, et al. Trends in use of medical imaging in US health care systems and in Ontario, Canada, 2000-2016. *Jama*, 2019.

[223] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[224] Michael J Steindorfer and Jurgen J Vinju. Fast and lean immutable multi-maps on the JVM based on heterogeneous hash-array mapped tries. *arXiv preprint arXiv:1608.01036*, 2016.

[225] Dmitrii Sumkin, Sameer Hasija, and Serguei Netessine. Does blockchain facilitate responsible sourcing? An application to the diamond supply chain. *An Application to the Diamond Supply Chain*, 2021.

[226] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[227] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 2005.

[228] Kenji Suzuki. Overview of deep learning in medical imaging. *Radiological physics and technology*, 2017.

[229] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[230] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[231] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[232] Serving models — TFX — tensorflow. `https://www.tensorflow.org/tfx/guide/serving?hl=nb`. (Accessed on 11/04/2021).

[233] New York Times. Bitcoin uses more electricity than many countries. how is that possible? `https://www.nytimes.com/interactive/2021/09/03/climate/bitcoin-carbon-footprint-electricity.html`. (Accessed on 02/02/2022).

[234] Torchserve — PyTorch/Serve master documentation. `https://pytorch.org/serve/`. (Accessed on 11/14/2021).

[235] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[236] Triton-inference-server/server: The Triton inference server provides an optimized cloud and edge inferencing solution. `https://github.com/triton-inference-server/server`. (Accessed on 11/04/2021).

[237] Steve Tsou. The need for computing power in 2020 and beyond. `https://www.forbes.com/sites/forbesbusinesscouncil/2020/01/24/the-need-for-computing-power-in-2020-and-beyond/?sh=2a22e79473c5`. (Accessed on 04/10/2023).

[238] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018. See also technical report Foreshadow-NG [245].

[239] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[240] Manuela Veloso, Tucker Balch, Daniel Borrajo, Prashant Reddy, and Sameena Shah. Artificial intelligence research in finance: discussion and examples. *Oxford Review of Economic Policy*, 2021.

[241] Visa. Small business retail. [Online; Accessed on 04/28/2019].

[242] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[243] Zhou Wang and Eero P Simoncelli. Translation insensitive image similarity in complex wavelet domain. In *Proceedings.(ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005*. IEEE, 2005.

[244] Jonathan Waring, Charlotta Lindvall, and Renato Umeton. Automated machine learning: Review of the state-of-the-art and opportunities for healthcare. *Artificial Intelligence in Medicine*, 2020.

[245] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [238].

[246] Guihua Wen, Zhi Hou, Huihui Li, Danyang Li, Lijun Jiang, and Eryang Xun. Ensemble of deep neural networks with probability-based fusion for facial expression recognition. *Cognitive Computation*, 2017.

[247] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for NVIDIA GPUs. *rn (A+ B)*, 2011.

[248] Bitcoin wiki. Scalability - bitcoin wiki. [Online; Accessed on 04/28/2019].

[249] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017. Accessed: 2018-01-03.

[250] Pieter Wuille. libsecp256k1. `https://github.com/bitcoin/secp256k1`. 2018.

[251] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[252] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *International Conference on Machine Learning*. PMLR, 2018.

[253] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[254] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[255] Ted Yin and Dahlia Malkhi. GitHib - HotStuff. `https://github.com/hot-stuff/libhotstuff/commit/df8328be09baeb81b7aaa037022eedaa7a416598`. (Accessed on 04/15/2021).

[256] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[257] Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*. Citeseer, 2005.

[258] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[259] Gengrui Zhang and Hans-Arno Jacobsen. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Middleware*, 2021.

[260] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[261] Yunhao Zhang. GitHub - yhzhang0128/archipelago-hotstuff: the artifact for our OSDI'20 paper. `https://github.com/yhzhang0128/archipelago-hotstuff`. (Accessed on 04/27/2021).

[262] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[263] Xiao-lin Zheng, Meng-ying Zhu, Qi-bing Li, Chao-chao Chen, and Yan-chao Tan. Fin-Brain: When finance meets AI 2.0. *Frontiers of Information Technology & Electronic Engineering*, 2019.

[264] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, 2011.

APPENDIX A

Proof of L-PBFT Linearizability

We present a correctness proof for L-PBFT. In particular, we show that *early execution* (Theorem A.1.2) and the *nonce commitment* scheme (Theorem A.1.3) are equivalent to their counterpart features in PBFT. In Theorem A.1.4, we show linearizability of L-PBFT.

A.1 Correctness proof

Lemma A.1.1 (Rollback)**.** *Any honest L-PBFT replica can roll back a suffix of the sequence of previously executed transaction batches.*

Proof. L-PBFT's state is distributed across several entities: a key-value store kv; a Merkle tree M; a ledger $\mathcal{L}$; a set of requests waiting to be ordered $\mathcal{T}$; a message store $\mathcal{M}$; and a nonce store $\mathcal{K}$. Therefore, to roll back a batch of transactions, it must be possible to roll back all of these entities.

Key-value store kv**.** The key-value store maintains a roll back transaction log. This enables transactions to be rolled back at a single transaction granularity. Thus, the last executed batch of transactions can be rolled back.

Merkle tree M**.** When a new node is added to L-PBFT's Merkle tree, it becomes the right-most leaf of the tree. The value of a node in the tree is never updated, and a node can only be deleted if it is the right-most node in the tree. Thus, during roll back, it is possible to remove the nodes from the right of the tree that represent the last batch of executed transactions (in reverse order).

Ledger $\mathcal{L}$. The ledger is represented by a file written to the disk by each replica. L-PBFT stores the index of all entries written to the ledger. To roll back the last executed batch, a L-PBFT replica truncates the ledger file to just before the first entry of the batch.

Transaction store $\mathcal{T}$. It is not necessary to undo changes to the transaction store. Transaction requests that are removed can be retransmitted by the client or other replicas if needed.

Message store $\mathcal{M}$, nonce store $\mathcal{K}$. All items in the transaction and nonce stores are indexed by sequence number and view. Since roll back occurs only during a view change, and each item is associated with a view, it is not necessary to modify the message and nonce stores, because honest replicas never send more than one item of a given type for the same sequence number and view.

Therefore, it is possible to roll back a suffix of the sequence of transaction batches executed by L-PBFT replicas.

□

Lemma A.1.2 (Early execution). *L-PBFT's early execution and PBFT execution agree on all committed transactions.*

Proof. In both PBFT and L-PBFT, the primary determines the order of request execution by ordering requests into batches and assigning numbers to batches in pre-prepare messages. In PBFT, requests are executed after commit and clients only accept results after transactions commit. In L-PBFT, requests are executed earlier, before the request even prepare, but the replicas only reply to clients after they prepare the requests and clients wait for matching replies from $N-f$ replicas. This ensures that they only obtain the transaction results after they commit as in PBFT.

As in PBFT, a faulty primary may cause requests for which pre-prepares are sent not to commit. L-PBFT deals with this case by rolling back early execution (see Theorem A.1.1).

□

Lemma A.1.3 (Nonce commitment). *The nonce commitment scheme is equivalent to replicas signing commit messages.*

Proof. L-PBFT, like PBFT, signs pre-prepare and prepare messages. Unlike PBFT, L-PBFT does not sign commit messages. Replicas sample a fresh random nonce for each pre-prepare or prepare message with sequence number s at view v, and add a hash of this nonce to the signed payloads. Later in the protocol, replicas include the nonce in the commit message, instead of an extra signature.

We show that this provides the same standard cryptographic security as the signature scheme (namely, resistance to existential forgery against chosen-message attacks) as long as the cryptographic hash function is second pre-image resistant on random inputs. Since the addition of a nonce to the signed payloads is injective, a forgery of a L-PBFT authenticator for a pre-prepare or prepare message yields a forgery against the signature scheme. A forgery of an authenticator for a commit message, i.e., a value with the same hash as a fresh random nonce that has not yet been revealed, is a second pre-image collision. □

Theorem A.1.4. *L-PBFT is linearizable.*

Proof. L-PBFT changes the PBFT algorithm by adding early execution and the nonce commitment scheme. Lemmas A.1.2 and A.1.3 show that these preserve the behaviour of PBFT. □

APPENDIX B

Proof of Auditing Correctness

First, we present the correctness proof for auditing without governance transactions and reconfiguration (Appendix B.1). Then, we extend the proof to include governance transactions and reconfiguration (Appendix B.2).

B.1 Correctness of auditing without reconfiguration

We begin with a description of terminology and notation. In Appendix B.1.1 and Lemma B.1.1, we then prove that, given a set of receipts, the auditor, with the help of the enforcer, can obtain a ledger package that is complete in relation to the receipts (or assign blame to $f+1$ misbehaving or slow replicas). A complete ledger package contains all evidence that is necessary for the auditor to assign blame to misbehaving replicas if the receipts reflect any linearizability violation. In Appendix B.1.2 and Lemma B.1.2, we show that, if a receipt does not appear correctly in a ledger package that is complete in relation to it, the auditor can assign blame to at least $f+1$ misbehaving replicas. In Appendix B.1.3 and Lemma B.1.3, using the previous lemmas, we first prove that the auditor can assign blame correctly if it is given a set of receipts that reflects a serializability violation. Finally, Theorem B.1.4 proves that, if a set of receipts reflects any linearizability violation, the auditor can assign blame to $f+1$ misbehaving or slow replicas.

Minimum ledger index. Each client transaction request includes a field that specifies the minimum ledger index that it can be executed at. Correct replicas do not order a transaction t

at ledger index i, unless $i \geq m_i$ where m_i is the minimum index value of t. Correct clients set the minimum index of a transaction to at least $M_i{+}1$ where M_i is the largest value of the ledger index that they know of from the receipts that they have collected. The minimum index value is used to capture transaction dependencies efficiently and to reduce the amount of information that needs to be stored and transmitted to audit linearizability violations.

Ledger well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries when there are at most f misbehaving replicas.

A ledger fragment is *well-formed* if either (i) it is valid, or (ii) it would be valid if not for the incorrect execution of one or more transactions, one or more incorrect checkpoint digests, or one or more invalid signatures or nonces.

A well-formed ledger matches the structural specifications of the L-PBFT protocol, i.e.,

- it specifies a serial ordering of transactions/entries, which respects their minimum ledger indices; and
- it includes evidence, and checkpoints at the required places.

A valid ledger is always well-formed, but a well-formed ledger can be invalid. A correct replica will never have a malformed ledger fragment, because replicas check the well-formedness of ledgers that they fetch. A correct replica may have an invalid ledger fragment. A ledger fragment can be well-formed but invalid only if there have at some point existed more than $N{-}f{-}1$ misbehaving replicas.

Notation. Given a receipt $\langle\langle t_j, i_j, o_j\rangle, x_j\rangle$, we denote $\langle t_j, i_j, o_j\rangle$ by tio_j. Unless explicitly defined otherwise, s_j refers to the sequence number in x_j of the receipt $\langle \text{tio}_j, x_j\rangle$.

We say that a replica has "signed a receipt" if its signature is recorded in the receipt in the pre-prepare/prepare signatures' fields (σ_p or in $\sum_s$).

Receipt validity. A receipt is *valid* if it is verifiable by Algorithm 3.3.

Preparement evidence for a batch. The *preparement* evidence for a batch is $N{-}f$ signed pre-prepare/prepare messages for the batch, i.e., $\mathcal{P}$ in Section 3.3.

Checkpoint sequence numbers. Let $\langle \text{tio}_j, x_j\rangle$ be a valid receipt, d_{C_j} be the checkpoint digest in x_j, and C be the checkpoint interval. Anyone can calculate the sequence number at which the digest of the checkpoint is expected to be equal to d_{C_j} as follows: checkpoints are always taken at sequence numbers that are multiples of C and the digest in the receipt refers to the digest at the sequence number of the penultimate checkpoint transaction before s_j (except the first C transactions, which have the digest at genesis). So given s_j, the sequence number with the corresponding checkpoint digest, s_{cp}, can be calculated as

$$s_{cp} = \begin{cases} 0 & \text{if } s_j < C \\ C\left(\lceil \frac{s_j}{C} \rceil - 2\right) & \text{otherwise.} \end{cases}$$

Note that the value of the digest itself is recorded in the last checkpoint transaction before s_j (except the first C transactions), i.e., the checkpoint transaction that follows the one at s_{cp}. That checkpoint transaction is at

$$\begin{cases} 0 & \text{if } s_j < C \\ s_{cp} + C & \text{otherwise.} \end{cases}$$

We assume that the genesis transaction *gt* is at sequence number 0.

Fetching checkpoints. Slow replicas can be brought up to date by fetching checkpoints and ledger fragments. When a correct replica fetches a checkpoint at sequence number s, it retrieves the ledger up to $s + C + P$. It first verifies the signatures in the evidence for the checkpoint transactions at s and $s + P$. Note that the replicas that signed the checkpoint transaction at s vouch for the validity of the ledger fragment between $s - C$ and s, whereas the replicas that signed the checkpoint transaction at $s + C$ vouch for the digest of the checkpoint at s.

A correct replica, then, verifies that the digest of the checkpoint that it fetched matches the value recorded at $s + C$. It also checks, for each checkpoint transaction at sequence number s' in the ledger, that the ledger's Merkle root at s' matches the root in the evidence for the transaction at s'. Finally, the replica replays the ledger fragment between $s + 1$ and $s + C$.

As noted previously, a correct replica may have a well-formed ledger fragment that includes invalid signatures as replicas do not verify all signatures in the ledger fragments that they fetch. Therefore, when contacted for an audit, a correct replica never returns a ledger fragment that it fetched with a checkpoint at sequence number s, without including the checkpoint transaction at $s + C$ and the evidence for that transaction.

B.1.1 Obtaining the ledger

Ledger package. A *ledger package* from a replica consists of one to four components:

1. a ledger fragment $\mathcal{F}$ that contains entries that locally prepared at the replica;
2. an optional *suffix* $\mathcal{U}$ that contains entries that were preprepared atomically after a view-change but not yet prepared at the replica;
3. an optional *message box* $\mathcal{E}$ that contains some of the messages from the replica's message box $\mathcal{M}$; and

4. an optional *checkpoint* cp.

Complete ledger package. Let $\mathcal{R}$ be a set of valid receipts; $s_{\max}$ be the maximum sequence number in $\mathcal{R}$; $s_{\min}$ be the sequence number of the checkpoint whose digest is expected to equal the checkpoint digest in the receipt with the smallest sequence number in $\mathcal{R}$ ($s_{\min}$ can be calculated as described in the previous section); $v_{\min}$ and $v_{\max}$ be the minimum and maximum view numbers in the receipts in $\mathcal{R}$, respectively.

A ledger package is *complete* in relation to $\mathcal{R}$ if all of the following are true:

- $\mathcal{F} + \mathcal{U}$ is well-formed;
- if $s_{\min} = 0$, cp contains the checkpoint at genesis (empty); otherwise, the digest of cp is equal to the one in the second checkpoint transaction in $\mathcal{F} + \mathcal{U}$;
- $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to $v_{\min} + 1$ ($v_{\min}$ requirement), and one set of *view-change* and *new-view* messages for a view greater than or equal to $v_{\max}$ ($v_{\max}$ requirement);
- All signatures in $\mathcal{F} + \mathcal{U}$ and $\mathcal{E}$ are valid.

and one of the following is true:

- $\mathcal{F}$ includes entries between $s_{\min}$ and $s_{\max} + P$;
- $\mathcal{F}$ includes entries between $s_{\min}$ and $s_{\max} + c$ where $c \in [0, P)$. $\mathcal{E}$ contains $P - c$ valid preparement evidence for entries from $s_{\max} - c$ to $s_{\max}$; or
- $\mathcal{F}$ includes entries between $s_{\min}$ and $e = \max(s_{\min}, s_{\max} - c)$ where $c \in [1, P]$. $\mathcal{E}$ contains valid preparement evidence for entries from $\max(s_{\min}, e - P)$ to e. The suffix $\mathcal{U}$ contains entries between $e + 1$ and $s_{\max}$ that are preprepared but not prepared in some view $v' \geq v_{\max}$ and $\mathcal{E}$ contains preparement evidence from a view $< v'$ for entries between $e+1$ and $s_{\max}$.

Lemma B.1.1 (Obtaining a complete ledger package). *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f + 1$ misbehaving or slow replicas.*

Proof. Select from the receipts in $\mathcal{R}$, the receipts with the highest view number $v_{\max}$. Then, from those receipts select the receipts with the highest sequence number. Finally, among those, let $R_{v\max}$ be the receipt with the highest index number. (We assume there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.)

The enforcer asks all replicas that signed $R_{v\max}$ for a ledger package that is complete in relation to $\mathcal{R}$. We assume that correct replicas or members respond to the enforcer before the agreed deadline. Once the enforcer has responses from $f+1$ replicas, it relays the responses to the auditor; otherwise at the deadline, the enforcer assigns blame to at least $f+1$ misbehaving or slow replicas.

We show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$ or a ledger package with which the auditor can assign blame to $f+1$ misbehaving replicas. Therefore, after checking $f+1$ responses, the auditor either finds a complete ledger package, or assigns blame to $f+1$ misbehaving replicas.

Note that a correct replica that is contacted by the enforcer can always satisfy the first three conditions of completeness: (1) correct replicas always maintain well-formed ledgers and they record/can recalculate checkpoints; (2) the $v_{\min}$ requirement can always trivially be satisfied by including the set of *view-change* and *new-view* messages for view 0 in $\mathcal{F}$. In practice, for efficiency, correct replicas would satisfy this requirement by including the set of *view-change* and *new-view* messages for some view v', where v' is the latest possible in $[0, v_{\min}+1]$; and (3) since the replicas that are asked are the replicas that signed $R_{v\max}$, they must have *view-change* and *new-view* messages for view $v_{\max}$. Therefore, any replica that returns a ledger package that violates any of the first three conditions can be assigned blame.

The fourth condition of completeness requires that all signatures and the matching nonces in the ledger package are correct. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package returned by a replica. If $\mathcal{U}$ or $\mathcal{E}$ contains a message or transaction with an invalid signature, the auditor can assign blame to the replica. $\mathcal{E}$ contains messages from the replica's message box and $\mathcal{U}$ contains batches that pre-prepared at the replica. A correct replica never considers a message or pre-prepares a batch that includes an invalid signature. Otherwise, let s_w be a sequence number where there is a transaction or message with an invalid signature. The auditor can look for the first checkpoint transaction that follows s_w that has no invalid signatures in its evidence. If one exists, the auditor can assing blame to all $N-f$ replicas that signed that checkpoint transaction. If no such checkpoint transaction exists, the auditor can assign blame to the responding replica, since a correct replica never returns a ledger fragment that it has fetched with a checkpoint without including the committed checkpoint transaction that records that checkpoint's digest. So given a ledger package from a replica, the auditor can always verify all signatures and nonces in the package or assign blame to the responding replica or $N-f$ misbehaving replicas. So below, for brevity, we can assume that the ledger package that a replica returns has no invalid signatures or nonces.

Additionally, for a correct replica that is contacted by the enforcer, one of the following must hold:

- **The correct replica has locally prepared entries up to at least $s_{\max}$:** In this case, the replica can form a complete ledger package that includes either:

  (i) a well-formed ledger fragment $\mathcal{F}$ that contains entries from $s_{\min}$ to $s_{\max} + P$; or

  (ii) a well-formed $\mathcal{F}$ that contains entries from $s_{\min}$ to $s_{\max} + c$ where $c \in [0, P)$, and $\mathcal{E}$ that contains $P - c$ valid preparement evidence for entries from $s_{\max} - c$ to $s_{\max}$.

- **The correct replica has not locally prepared entries up to $s_{\max}$ and it has locally prepared entries up to $e = s_{\max} - c$ where $c \geq 1$:** In this case, (1) a correct replica can include entries between $s_{\min}$ and e in a well-formed ledger fragment $\mathcal{F}$, and it can include the necessary preparement evidence in $\mathcal{E}$ (if $s_{\min} \leq e$); and (2) if the replica has any batches that it has preprepared but not prepared due to a view-change, it can include the related *view-change* and *new-view* messages in $\mathcal{F}$ and the batches in $\mathcal{U}$. Let p be the last sequence number for which there is a batch in $\mathcal{F} + \mathcal{U}$. If $p > e$, the correct replica can include the preparement evidence for entries between $e + 1$ and p in $\mathcal{E}$ as well. A correct replica can form a ledger package as described above. If $p \geq s_{\max}$, the ledger package is complete, and the replica can return it.

  Otherwise, $p < s_{\max}$. Let $R_{s\max}$ be the receipt in $\mathcal{R}$ with the largest sequence number $s_{\max}$ and let $v_{s\max}$ be the view number in $R_{s\max}$. Note that $v_{s\max} \leq v_{\max}$ by definition, and in the correct replicas' ledger, there must exist at least one set of *view-change* and *new-view* messages for a view $v' > v_{s\max}$ such that none of the *view-change* messages include a pre-prepare message for any batch at $s_{\max}$. The correct replica can return a ledger package that contains these *view-change* and *new-view* messages. The auditor can use the returned ledger package to assign blame to the intersection of replicas that signed $R_{s\max}$ and that sent the set of *view-change* messages for v', as these replicas have prepared a batch at $s_{\max}$ but did not report it during the view change.

Thus, for each of the $f + 1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the misbehaving responder, or at least $f + 1$ misbehaving replicas. □

By definition of completeness, if a ledger package is complete in relation to a set of valid receipts $\mathcal{R}$, it is complete in relation to any subset of $\mathcal{R}$.

Finding preparement evidence. For a batch at s_r, the auditor can find the preparement evidence for the batch as follows:

- if $\mathcal{F}$ contains an entry at $s_r + P$, it is collected from there;
- if $\mathcal{F}$ contains the entry at s_r but not at $s_r + P$, it is collected from $\mathcal{E}$; and

- if $\mathcal{F}$ does not contain an entry at s_r but $\mathcal{U}$ contains an entry at s_r, it is also collected from $\mathcal{E}$, albeit it is for the same batch from a prior view.

B.1.2 Incompatibility

Let $R = \langle \text{tio}_r, x_r \rangle$ be a valid receipt at sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch that is at s_r in $\mathcal{F} + \mathcal{U}$. R is *incompatible* with B_l if any of the following hold:

- t_r does not appear in B_l;
- it does not appear in the i_rth position; or
- o_r is different.

Lemma B.1.2 (Receipt-ledger incompatibility)**.** *Let $R = \langle tio_r, x_r \rangle$ be a valid transaction receipt for sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch in the package at s_r. If R is incompatible with B_l, the auditor can assign blame to at least $f + 1$ misbehaving replicas.*

Proof. The auditor can calculate the set of replicas that signed B_l using the preparement evidence that can be found as described above. These replicas are called $\mathcal{E}_l$.

Let $\mathcal{E}_r$ be the set of replicas that have signed the receipt. Let v_r be the view number in the receipt and v_l be the view number in the preparement evidence of B_l.

- $v_r = v_l$**:** Correct replicas never sign pre-prepare or prepare messages for different batches in the same view. Therefore, the auditor can assign blame to the replicas in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_l$, and $|\mathcal{E}_r \cap \mathcal{E}_l| \geq f + 1$.
- $v_l > v_r$**:** Correct replicas include the pre-prepare messages for the last P prepared batches in their *view-change* messages until the batches commit or a different batch is prepared at the sequence number. A correct primary always re-preprepares the latest batch that it finds in the set of $N - f$ *view-change* messages that it receives. Thus, there exists at least one view $v_c \in [v_r + 1, v_l]$ where zero of the $N - f$ *view-change* messages for v_c contain a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R, so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to $v_r + 1$ (the v_{min} requirement). It must also include the set of *view-change* and *new-view* messages for v_c as $v_l \geq v_c \geq v_r + 1$.

  Let $\mathcal{E}_c$ be the set of replicas that have sent the *view-change* messages to the primary for view v_c. The auditor can assign blame to the replicas that are in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_c$ and $|\mathcal{E}_r \cap \mathcal{E}_c| \geq f + 1$.

- **$v_l < v_r$:** There exists at least one view $v_c \in [v_l+1, v_r]$ where zero of the $N-f$ *view-change* messages for v_c contains a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view greater than or equal to v_r, so it must include the set of *view-change* and *new-view* messages for v_c as $v_l + 1 \leq v_c \leq v_r$ (the $v_{\max}$ requirement). Similar to previous case afterwards.

□

B.1.3 Violations

Ordering receipts. Given a set of valid receipts, the auditor can order them lexicographically based on the corresponding (sequence number, index number, view number) tuples. (We can assume that there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.) We say that a receipt R_1 is *earlier/later* than a receipt R_2, if it is ordered before/after R_2 with this scheme, respectively. For example, the earliest receipt in a set of valid receipts is the one with the lowest view number, among those with the lowest index number, among those with the lowest sequence number.

Lemma B.1.3 (Serializability violations). *Let $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ be a set of valid receipts that violates serializability. Then, the auditor can assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, it can assign blame to at least $f+1$ misbehaving or slow replicas by Lemma B.1.1. Note that, as the ledger package is complete in relation to $\mathcal{R}$, it is complete in relation to any receipt $R_j \in \mathcal{R}$.

Since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0, ..., t_k$ can produce $io_0, ..., io_k$. $\mathcal{F} + \mathcal{U}$ is well-formed, so there are two options for its validity:

Valid ledger. $\mathcal{F} + \mathcal{U}$ is a valid ledger, so every transaction in it is ordered and executed serially. However, the receipts in $\mathcal{R}$ violate serializability. Therefore, there must exist at least one receipt $\langle \text{tio}_w, x_w \rangle \in R$ that is incompatible with the batch at s_w in $\mathcal{F}+\mathcal{U}$. By Lemma B.1.2, the auditor can assign blame to at least $f+1$ misbehaving replicas.

Invalid ledger. $\mathcal{F} + \mathcal{U}$ is a well-formed but invalid ledger. So there exists at least one transaction t_w (which does not have to be in $\mathcal{R}$) that was executed incorrectly in some batch s_w, or one checkpoint that was created incorrectly.

The auditor can order $\mathcal{R}$ as described above. Let R_e be the earliest receipt in $\mathcal{R}$. Let d_{C_0} be the checkpoint digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest

d_{C_0}, calculated by the auditor using s_e and the checkpoint interval C as previously described. If $s_{C_O} = 0$, but the digest in R_e is not equal to the digest in the genesis transaction, the auditor can assign blame to all replicas that signed R_e. Otherwise, the ledger package is complete with respect to R_e, and $\mathcal{F} + \mathcal{U}$ is thus well-formed, so: (i) the entry at s_{C_0} in $\mathcal{F} + \mathcal{U}$ is a checkpoint transaction; and (ii) the checkpoint transaction in $s_{C_O} + C$ exists as $s_{C_0} < s_{C_O} + C < s_e$ and contains the digest of cp. If the digest of cp in the ledger package is not d_{C_0}, the auditor can assign blame to the replicas that signed both the checkpoint transaction at $s_{C_O} + C$ and R_e. The digest in that checkpoint transaction is for the previous checkpoint and the batches before the previous checkpoint have already committed since $C > P$.

Otherwise, the auditor replays the ledger starting from the checkpoint transaction at s_{C_0}, creating checkpoints at checkpoint sequence numbers. Doing so, the auditor either obtains $\langle t_w, i_w, o_a \rangle \neq \langle t_w, i_w, o_w \rangle$ or finds that an incorrect checkpoint digest is recorded at s_w. In either case, the auditor can assign blame to all replicas that signed for the batch at s_w. $\square$

Theorem B.1.4 (Linearizability violations). *Let $\mathcal{R}$ be a set of receipts that violate linearizability. Then, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas.*

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma B.1.3.

Otherwise, since the receipts violate linearizability but not serializability, the ordering of the transactions in $\mathcal{R}$ must violate the real-time ordering of the transactions. So there exists at least two transactions, t_a and t_b, in $\mathcal{R}$ such that the receipt for tio_a was received by the client before t_b was sent, but $i_a \geq i_b$. t_b was sent after $\langle \text{tio}_a, x_a \rangle$ was received, so a correct client sets the minimum index l of tio_b to at least $i_a + 1$. Since $i_b \leq i_a$, the auditor can assign blame to all replicas who have sent the receipt for tio_b. $\square$

B.2 Correctness of auditing with reconfiguration

In this section, we first summarize how reconfiguration happens, introduce new terminology, and update prior terminology. Then, in Lemma B.2.1, we prove that, if the auditor detects a fork in governance, it can assign blame to $f + 1$ misbehaving replicas. In Appendix B.2.1, we update the prior discussion on obtaining a complete ledger package. In Appendix B.2.2 and Lemma B.2.3, we prove that, if a receipt and the corresponding batch in a ledger package are prepared in different configurations, the auditor can assign blame to $f + 1$ misbehaving replicas. In Appendix B.2.3, using Lemma B.2.3, we update the prior lemma about incompatibility. Finally, Appendix B.2.4 updates the prior proofs on violations, and in Theorem B.2.6, we prove the correctness of auditing in the complete IA-CCF ledger system.

Summary of reconfiguration. A correct primary ends the batch it is working on once it executes a governance transaction. Therefore, each batch includes at most one governance transaction and i_g in a receipt refers to the last governance transaction executed before the transaction in the receipt. The final vote transaction that is necessary to pass a referendum triggers the configuration change. $2P$ *end-of-config* batches follow the final vote before the configuration change. The governance sub-ledger consists of batches and evidence for all governance transactions. It also includes, for each configuration, the P^{th} and $2P^{\text{th}}$ *end-of-config* batches, which commit the final vote transaction that triggers reconfiguration and the P^{th} *end-of-config* batch respectively. The P^{th} *end-of-config* batch links to the final vote transaction, because its pre-prepare message includes the Merkle root of the batch that includes the final vote transaction.

Updates to well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of configurations where in each configuration there are at most f failures.

In addition to the previous structural specifications, governance changes are serialized and include the required *end-of-config* and *start-of-config* messages.

Note that correct replicas check the validity of the governance sub-ledger fragments that they fetch, so their governance sub-ledgers are valid, in addition to well-formed.

Configuration number. The configuration number of a configuration $\mathcal{C}$ is the distance that it is from the configuration at the genesis. The genesis has configuration number 0. A configuration that follows the genesis configuration has number 1 and so on.

Supporting governance chain of a receipt. Every receipt R includes the index of the latest governance transaction. A correct client makes sure that it has a matching chain of valid governance transaction receipts for each receipt that it has. This includes the receipts for all governance transactions from the genesis up to the latest governance transaction, and the receipt for the P^{th} *end-of-config* batch for each configuration change. The supporting governance chain of a receipt R is the sequence of governance-related receipts that starts from the genesis transaction receipt and ends with the P^{th} *end-of-config* batch receipt before the configuration that signed R takes effect.

A supporting governance chain of a receipt matches a governance sub-ledger if each receipt in the chain is compatible with the governance sub-ledger. (For *end-of-config* batches, compatibility considers committed Merkle roots as well.) Similarly, a supporting governance chain can be a prefix of a governance sub-ledger.

Updates to receipt validity. A receipt is *valid* if it is verifiable by Algorithm 3.3, and it is attached a valid supporting governance chain.

Updates to calculating checkpoint sequence numbers. If a sequence number that is multiple of the checkpoint interval C falls into an *end-of-config/start-of-config* sequence, checkpointing is skipped. A checkpoint is taken at the beginning of each new configuration, and the digest of the first checkpoint in a configuration is included in the first checkpoint transaction, as opposed to the one that follows (this is similar to genesis).

Let $\langle \mathrm{tio}_j, x_j \rangle$ be a valid receipt and s_{fv} be sequence number of the final vote transaction for the last configuration change in the supporting governance chain of the receipt. The first checkpoint of the configuration that prepared the receipt is expected at $s_{fcp} = s_{fv} + 2P + 1$. (Except the genesis configuration, for which $s_{fcp} = 0$.)

So given s_j, the sequence number s_{cp} of the checkpoint whose digest is in x_j can be calculated with

$$s_{cp} = \begin{cases} s_{fcp} & \text{if } s_j < s_{fcp} + C \\ C\left(\lceil \frac{s_j - s_{fcp}}{C} \rceil - 2\right) & \text{otherwise.} \end{cases}$$

Updates to fetching checkpoints. Following a configuration change, a correct new replica fetches the checkpoint at the penultimate checkpoint sequence number s' in the previous configuration (or the first checkpoint sequence number if there is only one). It also retrieves the full ledger. It replays the ledger from s' before creating a checkpoint at the beginning of the configuration.

Equivalence of P^{th} *end-of-config* batches. Two P^{th} *end-of-config* batches are equivalent if they:

(i) are at the same index and sequence number; and

(ii) are preceded by the same valid governance sub-ledger (their pre-prepares include the same committed Merkle root).

Two receipts for P^{th} *end-of-config* batches are equivalent if the batches specified in them are equivalent.

Governance fork. There is a fork in governance if there is a fork in the governance sub-ledger. That is, there are at least two P^{th} *end-of-config* batches for the same configuration number that belong in valid governance sub-ledgers, but that are not equivalent.

We say that there is a fork between two valid supporting governance chains if there are receipts for two P^{th} *end-of-config* batches for the same configuration number that are not equivalent.

We say that there is a fork between a valid supporting governance chain and a valid governance sub-ledger, if for the same configuration number, the P^{th} *end-of-config* batch specified by the receipt in the chain is not equivalent to the P^{th} *end-of-config* batch in the sub-ledger.

Lemma B.2.1 (Governance fork)**.** *If there is a fork in governance, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. If there is a fork in governance, there are at least two P^{th} *end-of-config* batches for the same configuration number that are not equivalent, namely P_1 and P_2.

A correct replica only prepares a P^{th} *end-of-config* batch at sequence number s once the final vote transaction that passes the referendum is committed at sequence number $s-P$. Thus, all governance transactions preceding it are committed too. This final vote transaction triggers the configuration change.

So the auditor can assign blame to the replicas that prepared both P_1 and P_2, because a correct replica that prepares one will never prepare another non-equivalent P^{th} *end-of-config* batch in the same configuration number. □

Longest supporting governance chain. Let $\mathcal{R}$ be a set of valid receipts. If there is a fork between the supporting governance chains of the receipts in $\mathcal{R}$, the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma B.2.1. So the auditor can always obtain a *longest supporting governance chain* for the receipts in $\mathcal{R}$. This chain is the union of all supporting chains for receipts in $\mathcal{R}$.

Onwards, we assume that, given any set of valid receipts, the supporting governance chains are fork-free with each other and that there is a longest supporting governance chain; otherwise, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma B.2.1.

Transaction receipts. Onwards, we assume that a receipt is for a transaction and not for *end-of-config/start-of-config* batches. If the receipts for *end-of-config/start-of-config* indicate a fork in governance, misbehaving replicas can be blamed using Lemma B.2.1; otherwise, the *end-of-config/start-of-config* batches do not have any usage and do not affect the key-value store, so do not affect linearizability.

B.2.1 Updates to obtaining the ledger

Updated ledger package. A ledger package includes an additional required field:

- the committed governance sub-ledger $\mathcal{N}$ of the replica.

Updated definition of completeness. Let $\mathcal{R}$ be a set of valid receipts. Define $s_{\max}, v_{\min}, v_{\max}$ as previously. Calculate $s_{\min}$ using the receipt with the smallest configuration number, among those with the smallest sequence number in $\mathcal{R}$. Let $n_{g\max}$ be the longest supporting governance chain in $\mathcal{R}$.

A ledger package is *complete* in relation to $\mathcal{R}$ if, in addition to the prior conditions about well-formedness, length, and $v_{\min}/v_{\max}$ requirements:

- $n_{g\max}$ is a prefix of $\mathcal{N}$ (i.e. the package is obtained from a replica in a configuration which is equal to or succeeds all configurations in $\mathcal{R}$);
- $\mathcal{N}$ is valid; and
- $\mathcal{N}$ matches $\mathcal{F}$.

The condition for the checkpoint *cp* is updated as follows:

- if $s_{\min}$ is calculated as the first checkpoint transaction in a configuration (or zero), the digest of *cp* is equal to the one in the checkpoint transaction at $s_{\min}$; otherwise, the digest of *cp* is equal to the one in the second checkpoint transaction in $\mathcal{F} + \mathcal{U}$.

Lemma B.2.2 (Obtaining a complete ledger package with reconfiguration)**.** *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. As mentioned before, we assume that there is no fork between the supporting governance chains of the receipts in $\mathcal{R}$. Let $R_{g\max}$ be the receipt with the highest index number, among those with the highest sequence number, among those with the highest view number, among those with the longest supporting governance chain in $\mathcal{R}$. Let $n_{g\max}$ be the supporting governance chain of $R_{g\max}$.

We assume that there is a reliable mechanism to look up the most recent system configuration. Using this mechanism, the auditor looks up the most recent committed governance sub-ledger and the set of replicas that signed the first checkpoint transaction of the most recent configuration. If there is a fork between $n_{g\max}$ and the governance sub-ledger that is looked-up, the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma B.2.1; otherwise, the auditor checks whether the sub-ledger that is looked up is longer than $n_{g\max}$. If so, the enforcer asks all the replicas that signed the first checkpoint transaction of the most recent configuration for a ledger package; otherwise, the replicas that have signed $R_{g\max}$ are asked.

As in Lemma B.1.1, the enforcer asks replicas for a ledger package that is complete in relation to $\mathcal{R}$. At the deadline, the enforcer relays the responses to the auditor. There are at least $f+1$ responses, or the enforcer can assign blame to $f+1$ misbehaving or slow replicas.

As before, we show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$, or a ledger package with which the auditor can assign blame to $f+1$ misbehaving replicas.

First, note that a correct replica that is contacted by the enforcer can always satisfy the updated completeness conditions (related to $\mathcal{N}$), because the replica is part of the most recent configuration and the conditions all pertain to keeping a valid governance sub-ledger. Of the conditions described previously, the well-formedness and $v_{\min}$ conditions can be satisfied, and invalid signatures in the package can be handled, just as in Lemma B.1.1. Since the replicas that are asked are not necessarily the replicas that signed the receipt with the highest view in $\mathcal{R}$, it is possible that they cannot satisfy the $v_{\max}$ requirement even if they are correct.

So, for a correct replica that is contacted by the enforcer one of the following must hold:

- **The replica cannot satisfy the $v_{\max}$ requirement:** Let $R_{v\max}$ be the latest receipt when the receipts are ordered lexycographically by (view number, configuration number, sequence number, index number). Let $n_{v\max}$ be the supporting governance chain of $R_{v\max}$. If there is a fork between $n_{v\max}$ and the committed sub-ledger $\mathcal{N}$ of the replica, the replica can return its governance sub-ledger and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma B.2.1. Otherwise, $n_{v\max}$ must be a prefix of $\mathcal{N}$ since the enforcer asks replicas from the most recent configuration. There are two possibilities for the relationship between $n_{v\max}$ and $\mathcal{N}$:

  1. $\mathcal{N} = n_{v\max}$. So $R_{g\max} = R_{v\max}$.Therefore, the correct replica signed $R_{v\max}$. Any correct replica that signed $R_{v\max}$ has the *view-change* and *new-view* messages for $v_{\max}$, so this case is a contradiction.
  2. $\mathcal{N}$ is longer than $n_{v\max}$. Let $P_{v\max+1}$ be the P^{th} *end-of-config* batch that ends $R_{v\max}$'s configuration C. Since the replica is correct and cannot satisfy the $v_{\max}$ requirement, $P_{v\max+1}$ must be prepared in a view $< v_{\max}$. Any correct replica that prepared $P_{v\max+1}$ must have committed a final vote transaction that triggers the configuration change in their ledger in a view less than $v_{\max}$. Since correct replicas never reset their ledger by more than P sequence numbers, they do not pre-prepare any batch with view $v_{\max}$ in C. So, the auditor can assign blame to the intersection of replicas that signed $R_{v\max}$ and prepared $P_{v\max+1}$.

- **The replica can satisfy the $v_{\max}$ requirement:** If, additionally, the replica has prepared (or pre-prepared with view changes) batches up to at least $s_{\max}$, it can return a ledger package that is complete in relation to $\mathcal{R}$ just as in Lemma B.1.1.

  Otherwise, let $R_{s\max}$ be the receipt with the largest sequence number $s_{\max}$. Let $n_{s\max}$ be the supporting governance chain of $R_{s\max}$. If there is a fork between $n_{s\max}$ and the replica's $\mathcal{N}$, the replica can return $\mathcal{N}$ and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma B.2.1. Otherwise, $n_{s\max}$ must be a prefix of $\mathcal{N}$ since the replicas asked by the enforcer are from the most recent configuration. Again, there are two possibilities:

1. **$\mathcal{N}$ is longer than $n_{s\max}$:** Let $P_{s\max+1}$ be the P^{th} *end-of-config* batch that ends $R_{s\max}$'s configuration. Since the replica is correct and cannot satisfy the $s_{\max}$ requirement, $P_{s\max+1}$ must be prepared at a sequence number less than $s_{\max}$. Any correct replica that prepared $P_{s\max+1}$ must have committed a final vote transaction that triggers the configuration change at latest at sequence number $s_{\max} - (P + 1)$. Since a correct replica never resets its ledger by more than P sequence numbers, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and prepared $P_{s\max+1}$.
2. **$\mathcal{N} = n_{s\max}$:** The group of replicas asked by the enforcer are from the same configuration that signed $R_{s\max}$, which is the most recent configuration. Since the replica is correct and from the most recent configuration $v_{s\max} \le v_{\max}$ by definition. In $\mathcal{F}$, as before, there must exist at least one set of *view-change* and *new-view* messages for a view $v' > v_{s\max}$ such that none of the *view-change* messages includes a pre-prepare for any batch at $s_{\max}$. Note that the configuration of the replicas that have sent these *view-change* messages must be the same as the configuration that signed the receipt, as that is the most recent configuration in the system. So just as in Lemma B.1.1, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and that sent the set of *view-change* messages for v'.

So, for each of the $f + 1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the responder, or at least $f + 1$ misbehaving replicas. □

B.2.2 Mismatching configurations

Lemma B.2.3 (Receipt-ledger configuration mismatch)**.** *Let $R = \langle tio_r, x_r \rangle$ be a valid receipt that was produced in a configuration $\mathcal{C}_r$. Let B_l be the batch that is at s_r in a ledger package that is complete in relation to R. Let $\mathcal{C}_l$ be the configuration of the replicas that signed B_l. If $\mathcal{C}_r \neq \mathcal{C}_l$, the auditor can assign blame to at least $f + 1$ misbehaving replicas.*

Proof. Since R is a valid receipt, it has a valid supporting governance chain. Since the ledger package is complete, it includes a valid governance sub-ledger $\mathcal{N}$ that leads to $\mathcal{C}_l$, which is fork-free with the supporting governance chain of R.

One of the following must hold:

- **$\mathcal{C}_r < \mathcal{C}_l$: $\mathcal{C}_r$ precedes $\mathcal{C}_l$:** Let P_{r+1} be the P^{th} *end-of-config* batch that ends the configuration $\mathcal{C}_r$. This batch and its evidence is included in $\mathcal{N}$. Since the package is complete,

$\mathcal{N}$ is consistent with the ledger fragment in the package. Since that ledger fragment is well-formed and B_l is at s_r, P_{r+1} is at the latest at sequence number $s_r - (P+1)$. Any replica that prepared P_{r+1} must have committed a final vote transaction that triggers the configuration change at the latest at sequence number $s_r - (2P+1)$. A correct replica that has prepared a batch at s_r in $\mathcal{C}_r$ never resets its ledger to earlier than $s_r - P$ even with view changes. So the auditor can assign blame to the replicas that both prepared P_{r+1} and signed R.

- **$\mathcal{C}_r > \mathcal{C}_l$: $\mathcal{C}_r$ succeeds $\mathcal{C}_l$:** We show that this case is impossible given that R is valid, and there is no fork between its supporting governance chain and $\mathcal{N}$. Since the ledger package is complete in relation to R, $\mathcal{N}$ includes the P^{th} *end-of-config* batch leading to $\mathcal{C}_r$ and it matches the well-formed ledger fragment in the package. Since B_l is at s_r, that batch can at earliest be at sequence number $s_r + P$. So there cannot be a valid receipt produced in $\mathcal{C}_r$ at s_r.

□

B.2.3 Updates to incompatibility

Lemma B.2.4 (Receipt-ledger incompatibility with reconfiguration)**.** *Let $R = \langle tio_r, x_r \rangle$ be a valid transaction receipt at sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp, \mathcal{N} \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch in the package at s_r. If R is incompatible with B_l, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. Define $\mathcal{E}_l, \mathcal{E}_r, v_l, v_r$ as in Lemma B.1.2. Note that we can assume that both the receipt and B_l are prepared by the same configuration $\mathcal{C}$; if not, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma B.2.3.

- **$v_r = v_l$:** Same as Lemma B.1.2.
- **$v_l > v_r$:** Calculate $\mathcal{E}_c$ as described in Lemma B.1.2. If the replicas in $\mathcal{E}_c$ are also from the configuration $\mathcal{C}$, the auditor can assign blame just as in Lemma B.1.2; otherwise, if the replicas in $\mathcal{E}_c$ are from a preceding configuration, the first checkpoint transaction of $\mathcal{C}$ is at the latest at sequence number $s_r - (P+1)$ since B_l is prepared by $\mathcal{C}$ and $\mathcal{F} + \mathcal{U}$ is well-formed. Furthermore, that checkpoint transaction is prepared in a view $v' > v_r$. A correct replica never signs the receipt at s_r in a view v_r and then resets its ledger by more than P sequence numbers while view changing to v'. So, the auditor can assign blame to the replicas that signed both that checkpoint transaction and the receipt.
- **$v_l < v_r$:** Calculate $\mathcal{E}_c$ as described in Lemma B.1.2. If the replicas in $\mathcal{E}_c$ are also from the configuration $\mathcal{C}$, the auditor can assign blame just as in Lemma B.1.2; otherwise the

replicas in $\mathcal{E}_c$ are from a configuration that succeeds $\mathcal{C}$. In this case, the P^{th} *end-of-config* batch that ends the configuration $\mathcal{C}$ is at the earliest at sequence number $s_r + P$, since B_l is prepared by $\mathcal{C}$ and $\mathcal{F} + \mathcal{U}$ is well-formed. Furthermore, that batch is prepared in a view $v' < v_r$. A correct replica that prepares that P^{th} *end-of-config* batch commits to the configuration change; it never resets its ledger to earlier than s_r and signs R. So, the auditor can assign blame to the replicas that signed both that *end-of-config* batch and the receipt.

□

B.2.4 Updates to violations

Lemma B.2.5 (Serializability violations with reconfiguration). *Let $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ be a set of receipts that violates serializability. Then, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas.*

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E}, \mathcal{N} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, IA-CCF can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma B.2.2.

Just as in Lemma B.1.3, since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0, ..., t_k$ can produce $io_0, ..., io_k$. $\mathcal{F}$ is well-formed, so there are two options for its validity:

Valid ledger. Similar to Lemma B.1.3. By Lemma B.2.4, the auditor can assign blame to at least $f + 1$ misbehaving replicas.

Invalid ledger. Assume that receipts are ordered lexicographically based on the corresponding (sequence number, configuration number, index number, view number) tuples. (We can assume that there is no tie; otherwise the auditor can assign blame to the replicas that signed both tied receipts.)

Let R_e be the earliest receipt in the ordered $\mathcal{R}$. Let d_{C_0} be the digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest d_{C_0}. s_{C_0} can be calculated by the auditor using s_e, the checkpoint interval C, and the supporting governance chain. (Note that s_{C_0} is equal to $s_{\min}$ that is calculated while obtaining the ledger.)

We can assume that the batch at s_e is prepared by the same configuration that sent the receipt; otherwise the auditor can assign blame to $f + 1$ misbehaving replicas by Lemma B.2.3. We also know that the supporting governance chain of R_e matches $\mathcal{F} + \mathcal{U}$ and that $\mathcal{F} + \mathcal{U}$ is well-formed. So, the checkpoint transactions at s_{C_0} (and $s_{C_0} + C$ if it exists) are prepared by the same configuration as R_e by definition of s_{C_0}. So, if the digest at s_{C_0} is not d_{C_0}, the auditor can assign blame to $f + 1$ misbehaving replicas similar to Lemma B.1.3.

Since the supporting governance chains of all receipts match the ledger fragment by definition of completeness, the auditor can determine the correct stored procedures for each transaction to replay the ledger as in Lemma B.1.3. □

Theorem B.2.6 (Linearizability violations with reconfiguration)**.** *Let $\mathcal{R}$ be a set of receipts that violate linearizability. Then, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas.*

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma B.2.5; otherwise, the minimum ledger index argument in the proof of Theorem B.1.4 holds. □