Security and Fairness of Blockchain Consensus Protocols

Inaugural Dissertation

of the Faculty of Science,
University of Bern

Presented by

Jovana (Mićić) Milojević

from Serbia

Supervisor of the Doctoral Thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Security and Fairness of Blockchain Consensus Protocols

Inaugural Dissertation

of the Faculty of Science,
University of Bern

Presented by

Jovana (Mićić) Milojević

from Serbia

Supervisor of the Doctoral Thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Accepted by the Faculty of Science.

Bern, 7th March 2024

The Dean
Prof. Dr. Marco Herwegh

Acknowledgements

I express my deepest gratitude to my supervisor, Prof. Christian Cachin, for his invaluable guidance and support throughout my PhD journey. His mentorship shaped me not just as a researcher but also as a person. Dear Christian, thank you for your patience and encouragement and for always having an open door for me. I am grateful for the opportunity to work with you.

I would also like to thank all my colleagues from the team for their support, collaboration, and friendship. I would especially like to thank the people who were there from the beginning of my journey: Luca Zanolini, Orestis Alpos, and Ignacio Amores Sesar. Thank you for everything. I hope our paths will cross again.

On a personal note, I would like to thank my family for their unconditional love and support. I would like to thank my parents, who always believed in me. I would like to thank my sister Sara, who was always there for me, even though we were far apart. Additionally, I would like to thank my family in Bern, who made me feel at home. Special thanks to my friends from Novi Sad: Mina, Katarina, and Marija. You inspired me to start this journey and supported me all the way.

My first teacher, Ruzica Savanovic, marked the very beginning of my school. She taught me not just how to read and write but also instilled in me the work habits that got me here. Thank you for everything. I will never forget you.

Finally, I would like to thank my significant one and loving husband, Goran, who was always there for me. Thank you for your love, patience, and encouragement. Thank you for supporting and believing in me when I did not believe in myself. I am grateful for having you in my life. Thank you for everything. I love you.

Zahvalnost

Izražavam najdublju zahvalnost svom mentoru, Prof. Christian Cachin, za nezamenljivo mentorstvo i podršku tokom mog puta ka doktorskoj tituli. Njegovo mentorstvo me je oblikovalo ne samo kao istraživača, već i kao osobu. Dragi Christian, hvala ti na strpljenju i ohrabrivanju, i što si uvek imao otvorena vrata za mene. Zahvalana sam na prilici da radim s tobom.

Želim se zahvaliti i svim kolegama iz tima na podršci, saradnji i prijateljstvu. Posebno bih želela zahvaliti ljudima koji su bili uz mene od samog početka: Luca Zanolini, Orestis Alpos i Ignacio Amores Sesar. Hvala vam na svemu. Nadam se da će nam se putevi opet ukrstiti.

Sa privatne strane, želim se zahvaliti svojoj porodici na bezuslovnoj ljubavi i podršci. Hvala roditeljima, koji su uvek verovali u mene. Želim se zahvaliti i sestri Sari, koja je uvek bila uz mene, iako nas je daljina radzvajala. Dodatno, želim se zahvaliti svojoj porodici u Bernu, koja je učinila da se osećam kao kod kuće. Posebno hvala prijateljima iz Novog Sada: Mini, Katarini i Mariji. Inspirisale ste me da krenem na ovaj put i podržale me.

Moja prva učiteljica, Ružica Savanović, obeležila je sam početak mog školovanja. Naučila me je ne samo kako da čitam i pišem, već mi je i usadila radne navike koje su me dovele do ovde. Hvala ti na svemu. Nikada te neću zaboraviti.

Na kraju, želim se zahvaliti mom voljenom suprugu, Goranu, koji je uvek bio uz mene. Hvala ti na ljubavi, strpljenju i ohrabrivanju. Hvala ti što si me podržavao i verovao u mene kada i sama nisam verovala u sebe. Zahvalana sam što te imam u svom životu. Hvala ti za sve. Volim te.

Abstract

The increasing popularity of blockchain technology has created a need to study and understand consensus protocols, their properties, and security. As users seek alternatives to traditional intermediaries, such as banks, the challenge lies in establishing trust within a robust and secure system. This dissertation explores the landscape beyond *cryptocurrencies*, including *consensus* protocols and *decentralized finance (DeFi)*.

Cryptocurrencies, like Bitcoin and Ethereum, symbolize the global recognition of blockchain technology. At the core of every cryptocurrency lies a *consensus* protocol. Utilizing a proof-of-work consensus mechanism, Bitcoin ensures network security through energy-intensive mining. Ethereum, a representative of the proof-of-stake mechanism, enhances scalability and energy efficiency. Ripple, with its native XRP, utilizes a consensus algorithm based on voting for efficient cross-border transactions. The first part of the dissertation dives into Ripple's consensus protocol, analyzing its security. The Ripple network operates on a Byzantine fault-tolerant agreement protocol. Unlike traditional Byzantine protocols, Ripple lacks global knowledge of all participating nodes, relying on each node's trust for voting. This dissertation offers a detailed abstract description of the Ripple consensus protocol derived from the source code. Additionally, it highlights potential safety and liveness violations in the protocol during simple executions and relatively benign network assumptions.

The second part of this thesis focuses on *decentralized finance*, a rapidly growing sector of the blockchain industry. DeFi applications aim to provide financial services without intermediaries, such as banks. However, the lack of regulation leaves space for different kinds of attacks. This dissertation focuses on the so-called *front-running* attacks. Front-running is a transaction-ordering attack where a malicious party exploits the knowledge of pending transactions to gain an advantage.

To mitigate this problem, recent efforts introduced *order fairness* for transactions as a safety property for consensus, enhancing traditional agreement and liveness properties. Our work addresses limitations in existing formalizations and proposes a new *differential* order fairness property. The novel *quick order-fair atomic broadcast (QOF)* protocol ensures transaction delivery in a differentially fair order, proving more efficient than current protocols. It works optimally in asynchronous and eventually synchronous networks, tolerating up to one-third parties corruption, an improvement from previous solutions tolerating fewer faults. This work is further extended by presenting a modular implementation of the QOF protocol. Empirical evaluations compare QOF's performance to a fairness-lacking consensus protocol, revealing a marginal 5% throughput decrease and approximately 50ms latency increase. The study contributes to understanding the practical aspects of QOF protocol, establishing connections with similar fairness-imposing protocols from the literature.

The last part of this dissertation provides an overview of existing protocols designed to prevent transaction reordering within DeFi. These defense methods are systematically classified into four categories. The first category employs distributed cryptography to prevent side information leaks to malicious insiders, ensuring a *causal order* on the consensus-generated transaction sequence. The second category, *receive-order fairness*, analyzes how individual parties participating in the consensus protocol receive transactions, imposing corresponding constraints on the resulting order. The third category, known as *randomized order*, aims to neutralize the influence of consensus-running parties on transaction order. The fourth category, *architectural separation*, proposes separating the task of ordering transactions and assigning them to a distinct service.

Contents

Chapter 1

Introduction

The growing popularity of blockchain technology brings a significant challenge: how to make it secure and fair. Building trust among users who aim to break away from traditional intermediaries, like banks or brokers, relies on a robust and secure system. Recognizing the impact of significant blockchain platforms like Bitcoin [71] and Ethereum [16], the blockchain landscape has expanded beyond *cryptocurrencies* to include *decentralized finance (DeFi)* and many other use cases.

Cryptocurrencies represent blockchain technology's globally recognized application, providing decentralized alternatives to traditional currencies. Bitcoin[1], introduced in 2009, operates on a proof-of-work consensus mechanism wherein miners compete to solve complex mathematical puzzles, validate transactions, and add new blocks to the blockchain. This energy-intensive process ensures the security and immutability of the Bitcoin network by requiring computational effort to append new transactions to the decentralized ledger. Another known blockchain, Ethereum[2], transitioned in 2022 from a proof-of-work to a proof-of-stake consensus mechanism to enhance scalability and reduce energy consumption. In proof-of-stake, validators are chosen to create new blocks based on the amount of cryptocurrency they stake or lock up as collateral, promoting a more energy-efficient and environmentally friendly approach to securing the network. Ripple[3], often associated with its native cryptocurrency XRP, operates as a digital payment protocol for seamless and

[1] `https://bitcoin.org/`
[2] `https://ethereum.org/`
[3] `https://ripple.com/`

rapid cross-border transactions. Unlike many decentralized cryptocurrencies, Ripple utilizes a more traditional consensus algorithm based on voting rather than mining, providing a more energy-efficient and scalable solution for financial institutions and remittance services.

Decentralized Finance (DeFi) is a paradigm within the blockchain ecosystem that seeks to recreate conventional financial services in a decentralized and trustless manner. In essence, it leverages blockchain technology to remove the need for intermediaries such as banks, enabling users to engage in financial activities directly with one another. DeFi encompasses various financial services, including lending, borrowing, trading, and yield farming. Notable examples of DeFi services include lending platforms like Compound[4] and Aave[5], where users can lend or borrow digital assets; decentralized exchanges like Uniswap[6] and SushiSwap[7], enabling peer-to-peer trading without centralized authorities; and decentralized autonomous organizations (DAOs) such as MakerDAO[8], allowing for decentralized governance and decision-making within the ecosystem. These services collectively contribute to the democratization of finance, providing users with greater control and access to financial instruments without traditional parties.

This dissertation explores these two aspects, consensus and decentralized finance, of the blockchain ecosystem. The first part dives into the Ripple [26] consensus protocol that runs XRP cryptocurrency, examining the security and resilience of the protocol. The second part focuses on the challenge of reordering transactions within decentralized finance. The dissertation introduces a novel protocol named *quick order-fair atomic broadcast protocol*. The designed and implemented protocol addresses the challenge of transaction reordering within decentralized finance (DeFi). In addition to this novel protocol, the dissertation offers a comprehensive overview of defense mechanisms against transaction reordering attacks. By surveying existing defense strategies, this thesis aims to provide the blockchain community with insights into defending decentralized financial systems.

The dissertation is organized as follows. Chapter 2 presents prerequisites for this work. Chapter 3 provides a security analysis of the Ripple consensus protocol. Chapter 4 introduces the Quick Order Fairness

[4] `https://compound.finance/`
[5] `https://aave.com/`
[6] `https://uniswap.org/`
[7] `https://www.sushi.com/`
[8] `https://makerdao.com/`

(QOF) protocol. Chapter 5 presents the implementation and evaluation of the QOF protocol. Chapter 6 provides an overview of defense techniques preventing the reordering of transactions. Chapter 7 concludes the dissertation and outlines future work. We summarize the contributions of this dissertation by highlighting the key topics in the following.

Security analysis of Ripple consensus [5]. The Ripple network is one of the most prominent blockchain platforms, and its native XRP token currently has one of the highest cryptocurrency market capitalizations. At the moment of writing this thesis, Ripple's market capitalization is worth around 34 billion USD. The Ripple consensus protocol powers this network and is generally considered a Byzantine fault-tolerant agreement protocol, which can reach consensus in the presence of faulty or malicious nodes. In contrast to traditional Byzantine agreement protocols, there is no global knowledge of all participating nodes in Ripple consensus; instead, each node declares a list of other nodes that it trusts and from which it considers votes. Previous work has brought up concerns about the liveness and safety of the consensus protocol under the general assumptions stated initially by Ripple, and at the moment of writing this work, there was no appropriate understanding of its workings and properties in the literature. Chapter 3 closes this gap and makes two contributions. It first provides a detailed, abstract description of the protocol, which has been derived from the source code. Second, the work points out that the abstract protocol may violate safety and liveness in several simple executions under relatively benign network assumptions.

Quick Order Fairness [21]. Leader-based protocols for consensus, i.e., atomic broadcast, allow some parties to unilaterally affect the final order of transactions. This has become a problem for blockchain networks and decentralized finance because it facilitates front-running and other attacks. To address this, *order fairness* for transactions has been introduced as a new safety property for atomic broadcast complementing traditional *agreement* and *liveness*. We relate order fairness to the standard validity notions for consensus protocols and highlight some limitations with the existing formalization. Based on this, Chapter 4 introduces a new *differential* order fairness property that fixes these issues. We also present the *quick order-fair atomic broadcast protocol* that guarantees transaction delivery in a differentially fair order and is much more efficient than existing order-fair consensus protocols. It works for asyn-

chronous and eventually synchronous networks with optimal resilience, tolerating corruptions of up to one-third of the parties. Previous solutions required there to be less than one-fourth of faults. Furthermore, our protocol incurs only quadratic cost regarding amortized message complexity per delivered transaction.

Quick Order Fairness: implementation and evaluation [20]. Consensus with a fair order aims to prevent front-running attacks, and in particular, the differential order fairness property addresses this problem and connects fair ordering to the validity of consensus. Chapter 5 revisits the QOF protocol and describes a modular implementation that uses a generic consensus component. Moreover, an empirical evaluation is performed to compare the performance of QOF to a consensus protocol without fairness. Measurements show that the increased complexity comes at a cost: throughput decreases by at most 5%, and latency increases by roughly 50ms, using an emulated ideal network. This work contributes to a comprehensive understanding of practical aspects regarding differential order fairness with the QOF protocol. Also, it connects this with similar fairness-imposing protocols like Themis [49] and Pompē [91].

Reordering defense in decentralized finance. In Chapter 6, we take a deep dive into the world of decentralized finance (DeFi) and explore the strategies designed to tackle front-running attacks. Despite DeFi's unquestionable advantages, the issue of front-running calls for our attention. This chapter delves into the basics of the technical challenges and inventive solutions surrounding front-running attacks in the DeFi landscape. It also positions the contribution of this thesis within the broader context of the research community. We bring to the table an overview of defense techniques and a breakdown of these methods into four distinct groups. Whether leveraging distributed cryptography, ensuring a causal order, or exploring architectural separation as a standalone service, we aim to present the state-of-the-art in this field. Nevertheless, it is not just about listing defense methods; our focus extends to developing a comprehensive understanding of how these protocols compare against each other. Our goal is to offer the research community insights into the differences between these protocols. Additionally, we dive into assessing performance metrics. In doing so, we lay the groundwork for collaborative efforts beyond just mitigating front-

running attacks, contributing to the ongoing evolution of DeFi against emerging challenges.

Publications. The work presented in this thesis is based on the following papers:

- Ignacio Amores-Sesar, Christian Cachin and Jovana Mićić:
  Security Analysis of Ripple Consensus,
  24th International Conference on Principles of Distributed Systems (OPODIS 2020), December 2020.

- Christian Cachin, Jovana Mićić, Nathalie Steinhauer and Luca Zanolini:
  Quick order fairness,
  Financial Cryptography and Data Security - 26th International Conference (FC 2022), May 2022.

- Christian Cachin and Jovana Mićić:
  Quick Order Fairness: Implementation and Evaluation,
  arXiv preprint arXiv:2312.13107, December 2023.

Chapter 2

Preliminaries

This section introduces the basic concepts and definitions used throughout this thesis. We start by introducing the components of the system model, which includes assumptions about processes, failures, communication, and different timing assumptions that can be made about the system. Finally, we define the abstractions used in the algorithms presented in this thesis. Note that in this section, we use the notion of messages as the subject of communication abstractions. This is because abstractions in this section are generic. When these abstractions are applied to blockchain systems, we speak of transactions.

2.1 System model

2.1.1 Processes

We model our system as a set of n processes, also called *parties* or *nodes*, which communicate with each other over the network. We define them as $\mathcal{P} = \{p_1, \ldots, p_n\}$. Messages are exchanged between processes reliably. Each process runs a protocol that is defined by a set of instructions. Processes are computationally bounded, and protocols may use cryptographic primitives. We assume that each process has a unique identifier known to all other processes. In our system, we consider two types of processes. *Correct* or *honest* processes are those who follow the protocol as expected. On the contrary, the processes that may crash or deviate from the protocol are called *faulty*.

When we devise an algorithm to implement a distributed programming abstraction, we want to satisfy two properties: *safety* and *liveness*. Safety means that this property can be violated at some time t and never be satisfied again after that time. In other words, the algorithm should not do anything wrong. A liveness property ensures that, eventually, something good will happen. Practical distributed systems should satisfy both properties.

2.1.2 Failures

A *failure* happens whenever the process does not behave according to the algorithm. Possible failures include *crash* failures, where a process stops executing the algorithm, a crash with recovery, and arbitrary faults. This thesis focuses on arbitrary faults, also called *Byzantine* faults. When we use this notion, we make no assumptions about the behavior of faulty processes. They can deviate arbitrarily from the protocol. An arbitrary fault does not necessarily mean to be malicious. A software bug or a hardware failure can cause it.

2.1.3 Communication

The abstraction of a *link* is used to model the network components of a distributed system. Links in this work are called *point-to-point* links, i.e., they provide communication between pairs of processes. We assume that a low-level mechanism exists for sending messages over *reliable* and *authenticated* perfect links. Our protocol descriptions refer to this as "sending a message" and "receiving a message."

Authenticated perfect links (al) [18, Sec. 2.4] primitive is used to prevent the forgery of messages on the link between processes, which is achieved by using cryptographic authentication. It is accessed through two events: *al-send* and *al-deliver*. The first event requests to send message m to process q. The second event delivers message m sent by process p.

Definition 2.1 (Authenticated perfect links). A protocol solves *authenticated perfect links* if it satisfies the following conditions:

Reliable delivery: If a correct process p sends a message m to a correct process q, then q eventually delivers m.

No duplication: No message is delivered by a correct process more than once.

Authenticy: If a correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by process p.

2.1.4 Timing

An important aspect of distributed systems is the timing assumptions that can be made about the system. This refers to the behavior of processes and links concerning the passage of time. We distinguish between three types of timing assumptions: *asynchronous*, *synchronous*, and *partially synchronous* [18, Sec. 2.5].

Asynchronous system. In considering an asynchronous distributed system, the fundamental principle involves refraining from making specific timing assumptions concerning processes and links. In other words, no assumptions have been made regarding processes having access to any form of physical clock, and there have been no presumptions about limits on processing or communication delays. Even without access to a physical clock, we can still measure the passage of time based on the transmission and delivery of messages. Such time is called *logical time* and uses *logical clocks*.

Synchronous system. Working with a synchronous system comes to assuming the following properties:

- *Synchronous computation:* there is a known upper bound on processing delays.
- *Synchronous communication:* there is a known upper bound on message transmission delays.

In a synchronous system, the synchronization of clocks among various processes is achievable in a manner that ensures they are never apart by more than a specified constant δ, known as the clock synchronization precision. The alignment of clocks facilitates the coordination of actions among processes, enabling the execution of synchronized global steps. We can use synchronized clocks to timestamp events at the instant they occur. These timestamps can be used to order events that occur at different processes. If a system exists where delays are constant, it would be possible to have perfectly synchronized clocks. However, this is not possible in practice, so events cannot be ordered perfectly.

Partial synchrony. Most of the time, distributed systems appear to be synchronous. Still, there are periods when timing assumptions do not hold, i.e., the system is asynchronous. This is, for example, when the network is overloaded. These systems are called partially synchronous. So, instead of assuming that the system is always synchronous, we assume it is eventually synchronous after an asynchronous period. We expect periods when the system is synchronous and long enough to allow the algorithm to do something useful or terminate its execution.

2.2 Abstractions

2.2.1 Reliable broadcast

Sometimes, the sender may fail when sending a message in a distributed system. In this case, some processes might deliver the message, and others not. Therefore, they do not agree on the delivery of the message. The *reliable broadcast (rb)* [18, Ch. 3] primitive ensures, with respect to crash faults, that correct processes agree on the set of messages they deliver, even when the senders of these messages crash while sending them. Reliable broadcast has two events: *rb-broadcast* and *rb-deliver*. The first event is used by a process p to send a message m to all other processes. The second event is used by a process q to deliver a message m previously broadcast by some process p.

Definition 2.2 (Reliable broadcast). A protocol solves *reliable broadcast* if it satisfies the following conditions:

Validity: If a correct process p broadcasts a message m, then p eventually delivers m.

No duplication: No message is delivered more than once.

No creation: IIf a process delivers a message m wih sender s, then m was previously broadcast by process s.

Agreement: If a correct process delivers a message m, then m is eventually delivered by every correct process.

2.2.2 FIFO broadcast

Reliable broadcast does not guarantee that all processes deliver the messages in the same order. The *first-in first-out (FIFO) order* [18, Sec. 3.9]

ensures that messages broadcast by the same sender process are delivered in the order in which they were sent. FIFO-order broadcast has two events: *frb-broadcast* and *frb-deliver*. The first event is used by a process p to send a message m to all other processes. The second event is used by a process q to deliver a message m previously broadcast by some process p.

Definition 2.3 (FIFO-order broadcast). A protocol solves *FIFO-order broadcast* if it satisfies the properties of reliable broadcast (Def. 2.2) and following condition:

FIFO delivery: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

FIFO-order broadcast primitive guarantees that messages from the same sender are delivered in the same sequence as they were broadcast. However, this does not affect messages from different senders.

2.2.3 Byzantine consistent broadcast

As we saw in Section 2.1, Byzantine processes may deviate arbitrarily from the protocol. Therefore, an algorithm must be prepared to tolerate such behavior. The *Byzantine Consistent Broadcast (bcb)* [18, Sec. 3.10] primitive solves one of the basic problems in the fail-arbitrary model. It is accessed through two events: *bcb-broadcast* and *bcb-deliver*. The first event is executed only by a sender s, and it sends a message m to all other processes. The second event delivers a message m previously broadcast by p.

Definition 2.4 (Byzantine Consistent Broadcast). A protocol solves *Byzantine consistent broadcast* if it satisfies the following conditions:

Validity: If a correct process p broadcasts a message m, then every correct process eventually delivers m.

No duplication: Every correct process delivers at most one message.

Integrity: If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

Consistency: If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

2.2.4 Byzantine FIFO consistent broadcast channel

Multiple broadcast instances can be combined to the higher-level notion, which we call *Byzantine broadcast channel*. In scenarios where a high-level algorithm employs numerous instances of a low-level primitive, it becomes essential to tag the implementation-level messages generated by the primitive with a suitable identifier. In some cases, this identifier must also be incorporated into cryptographic operations.

Byzantine FIFO consistent broadcast channel (bcch) [18, Sec. 3.12] allows the processes to deliver multiple messages and ensures consistency despite Byzantine senders. The interface of such a channel provides two events. A process invokes *bcch-broadcast*(m) to broadcast a message m to all processes. An event *bcch-deliver*(p_j, l, m) delivers a message m with label l from a process p_j.

The label with every delivered message is an arbitrary bit string generated by the channel. Intuitively, the channel ensures that if a message is delivered with some label, then the message itself is the same at all correct processes that deliver this label.

Definition 2.5 (Byzantine FIFO Consistent Broadcast Channel). A *Byzantine FIFO consistent broadcast channel* satisfies the following properties:

Validity: If a correct process broadcasts a message m, then every correct process eventually delivers m.

No duplication: For every process p_j and label l, every correct process delivers at most one message with label l and sender p_j.

Integrity: If some correct process delivers a message m with sender p_j and process p_j is correct, then m was previously broadcast by p_j.

Consistency: If some correct process delivers a message m with label l and sender p_j, and another correct process delivers a message m' with label l and sender p_j, then $m = m'$.

FIFO delivery: If a correct process broadcasts some message m before it broadcasts a message m', then no correct process delivers m' unless it has already delivered m.

2.2.5 Validated Byzantine consensus

Validated Byzantine consensus(vbc) [19] defines an *external validity* condition. It requires that the consensus value is legal according to a global,

efficiently computable predicate P known to all processes. This allows the protocol to recognize proposed values that are acceptable to an external application. Note that it is not required that the decision value was proposed by a correct process, but all processes must be able to verify the validity. A consensus primitive is accessed through the events *vbc-propose*(v) and *vbc-decide*(v), where $v \in \mathcal{V}$ has a potentially large domain $\mathcal{V}$ and may contain a proof, which allows processes to verify the validity of v.

Definition 2.6 (Validated Byzantine Consensus). A protocol solves *validated Byzantine consensus* with validity predicate P if it satisfies the following conditions:

Termination: Every correct process eventually decides some value.

Integrity: No correct process decides twice.

Agreement: No two correct processes decide differently.

External validity: Every correct process only decides a value v such that $P(v) = \text{TRUE}$. Moreover, if all processes are correct and propose v, then no correct process decides a value different from v.

2.2.6 Atomic broadcast

Atomic broadcast [18, Sec. 6.1] ensures that all processes deliver the same messages and that all messages are output in the same order. This is equivalent to the processes agreeing on one sequence of messages that they deliver. Atomic broadcast is also called "total-order broadcast" or simply "consensus" in the context of blockchains because it is equivalent to running a sequence of consensus instances. Processes may broadcast a message m by invoking *a-broadcast*(m), and the protocol outputs messages through *a-deliver*(m) events.

Definition 2.7 (Atomic Broadcast). A protocol for *atomic broadcast* satisfies the following properties:

Validity: If a correct process *a-broadcasts* a message m, then every correct process eventually *a-delivers* m.

No duplication: No message is *a-delivered* more than once.

No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.

Agreement: If a message m is *a-delivered* by some correct process, then m is eventually *a-delivered* by every correct process.

Total order: Let m and m' be two messages such that p_i and p_j are correct processes that *a-deliver* m and m'. If p_i *a-delivers* m before m', then p_j also *a-delivers* m before m'.

Chapter 3

Security Analysis of the Ripple Consensus Protocol

3.1 Introduction

The XRP Ledger of the Ripple network is one of the oldest and most established blockchains; its XRP token is nowadays ranked sixth in market capitalization (January 2024). The Ripple network primarily aims at fast global payments, asset exchange, and settlement. Its distributed consensus protocol is implemented by a peer-to-peer network of validator nodes that maintain a history of all transactions on the network [83]. Unlike Nakamoto's consensus protocol [71] in Bitcoin or Ethereum, the Ripple consensus protocol does not rely on "mining," but uses a voting process based on the identities of its validator nodes to reach consensus. This makes Ripple much more efficient than Bitcoin for processing transactions (up to 1500 transactions per second) and lets it achieve very low transaction settlement times (4–5 seconds).

However, Ripple's consensus protocol does not follow the established models and algorithms for *Byzantine agreement* [57], [75] or *Byzantine fault-tolerant (BFT) consensus* [24]. Those systems start from a common set of nodes communicating with each other to reach consensus, and the corresponding protocols have been investigated for decades. Instead, the Ripple consensus protocol introduces the idea of subjective

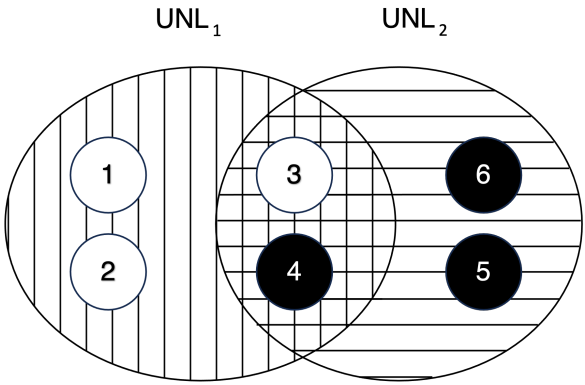


Figure 3.1: Example of a Ripple network configuration with six nodes and two UNLs, $UNL_1 = \{1, 2, 3, 4\}$ and $UNL_2 = \{3, 4, 5, 6\}$. Nodes 1, 2, and 3 (white) trust UNL_1, and nodes 4, 5, and 6 (black) trust UNL_2. Notice that nodes 3 and 4 have more influence than the rest of the nodes since they are at the intersection of both UNLs.

validators, such that every node declares some *trusted validators* and effectively communicates only with those nodes to reach an agreement on transactions. With this mechanism, Ripple designers aimed to open up membership in the set of validator nodes compared to BFT consensus. The trusted validators of a node are defined by a *Unique Node List (UNL)*, which plays an important role in the formalization of the protocol. Every node maintains a static UNL in its configuration file and considers only the opinions of nodes in its UNL during consensus. Figure 3.1 shows an example network, where two UNLs are defined: $UNL_1 = \{1, 2, 3, 4\}$ and $UNL_2 = \{3, 4, 5, 6\}$; for instance, nodes 1, 2, and 3 may trust UNL_1, and nodes 4, 5, and 6 may trust UNL_2.

Consensus in Ripple aims at delivering the transactions submitted by clients to all participating nodes in a common global order, despite faulty or malicious (Byzantine) nodes [86]. This ensures that the sequence of transactions, grouped into so-called *ledgers* and then processed by each node, is the same for all nodes. Hence, the states of all correct nodes remain synchronized, according to the blueprint of state-machine replication [85].

Cachin and Vukolić [23] have earlier pointed out that it is important to assess the properties of blockchain consensus protocols formally.

Unfortunately, many systems have been designed and deployed without following the agreed-on principles of protocol analysis from the literature. Ripple is no exception to this, as we show in this work.

Specifically, we focus on two properties that every sound protocol must satisfy [3]: *safety* and *liveness*. Safety means nothing "bad" will ever happen, and liveness means something "good" eventually happens. Safety ensures that the network does not fork or double-spend a token. A liveness violation would mean that the network stops making progress and halts processing transactions, which creates as much harm as forking.

This work presents a complete, abstract description of the Ripple consensus protocol (Section 3.3). The model has been obtained directly from the source code. It is formulated in the language spoken by designers of consensus protocols to facilitate a better understanding of the properties of Ripple consensus. No formal description of Ripple consensus with comparable technical depth has been available so far (apart from the source itself).

Second, we exhibit examples of how safety and liveness may be violated in executions of the Ripple consensus protocol (Sections 3.4 and 3.5). In particular, the *network may fork* under the standard condition on UNL overlap stated by Ripple and in the presence of a constant *fraction* of Byzantine nodes. The malicious nodes may simply send conflicting messages to the correct nodes and delay the reception of other messages among the correct nodes. Furthermore, the consensus protocol may *lose liveness* even if all nodes have the same UNL and there is only *one* Byzantine node. If this occurs, the system has to be restarted manually.

Given these findings, we conclude that the consensus protocol of the Ripple network is brittle and does not ensure consensus in the usual sense. It relies heavily on synchronized clocks, timely message delivery, the presence of a fault-free network, and an apriori agreement on common trusted nodes. The role of the UNLs, their overlap, and the creation of global consensus from subjective trust choices remain unclear. If Ripple instead had adopted a standard BFT consensus protocol [18], as done by Tendermint [15], versions of Hyperledger Fabric [6], Libra [61] or Concord [42], then the Ripple network would resist a much wider range of corruptions, tolerate temporary loss of connectivity, and continue operating despite loss of synchronization.

3.2 Related work

Despite Ripple's prominence and its relatively high age among blockchain protocols — the system was first released in 2012 — there are only few research papers investigating the Ripple consensus protocol compared to the large number of papers on Bitcoin. The original Ripple white paper of 2014 [86] describes the UNL model and illustrates some ideas behind the protocol. It claims that under the assumption of requiring an 80%-quorum for declaring consensus, the intersection between the UNLs of any two nodes u and v should be larger than 20% of the size of the larger of their UNLs, i.e.,

$$|UNL_u \cap UNL_v| \;\geq\; \frac{1}{5}\max\{|UNL_u|, |UNL_v|\}.$$

The only earlier protocol analysis in the scientific literature of which we are aware was authored by Armknecht *et al.* in 2015 [7]. This work analyzes the Ripple consensus protocol and outlines the security and privacy of the network compared to Bitcoin. The authors prove that a 20%-overlap, as claimed in the white paper, cannot be sufficient for reaching a consensus, and they increase the bound on the overlap to at least 40%, i.e.,

$$|UNL_u \cap UNL_v| \;>\; \frac{2}{5}\max\{|UNL_u|, |UNL_v|\}$$

In a preprint of 2018, Chase and MacBrough [26] further strengthen the required UNL overlap. They introduce a high-level model of the consensus protocol and describe some of its properties, but many details appear unclear or are left out. This work concludes that the overlap between UNLs should actually be larger than 90%. The paper also gives an example with 102 nodes that shows how liveness can be violated, even if the UNLs overlap almost completely (by 99%) and there are no faulty nodes. The authors conclude that manual intervention would be needed to resurrect the protocol after this.

An analysis whose goal is similar to that of our work has been conducted by Mauri *et al.* [68]. Based on the source code, they give a verbal description of the consensus protocol, but do not analyze dynamic protocol properties. Our analysis, in contrast, provides a detailed, formal description with pseudocode and achieves a much better understanding of how the "preferred ledger" is chosen. Moreover, our work shows possible violations of safety and liveness, whereas Mauri *et al.* address only on the safety of the consensus protocol through sufficient conditions.

Other academic work mostly addresses network structure, transaction graph, and privacy aspects of payments on the Ripple blockchain [65], [69], which is orthogonal to our focus.

3.3 A description of the Ripple consensus protocol

The main part of our analysis consists of a detailed presentation of the Ripple consensus protocol in this section and formally in Algorithms 1–4. Before describing this, we define the task the protocol intends to solve.

3.3.1 Specification

Informally, the goal of the Ripple consensus protocol is "to ensure that the same transactions are processed, and validated ledgers are consistent across the peer-to-peer XRP Ledger network" [82]. More precisely, this protocol implements the task of synchronizing the nodes so that they proceed through a common execution by appending successive *ledgers* to an initially empty history and where each ledger consists of a number of *transactions*. This is the problem of replicating a service in a distributed system, which goes back to Lamport et al.'s pioneering work on Byzantine agreement [57], [75]. The problem has a long history, and a good summary can be found in the book "30-year perspective on replication" [25].

For replicating an abstract service among a set of nodes, the service is formulated as a deterministic state machine that executes *transactions* submitted by clients or, for simplicity, by the nodes themselves. The *consensus protocol* disseminates the transactions among the nodes, such that each node locally executes the *same sequence of transactions* on its copy of the state. The task provided by this protocol is also called *atomic broadcast*, indicating that the nodes actually disseminate the transactions. When each node locally executes the same sequence of transactions as directed by the protocol, and since each transaction is deterministic, all nodes will maintain the same copy of the state [85].

More formally, *atomic broadcast* is characterized by two events dealing with transactions: *a-broadcast* and *a-deliver*, which may each occur multiple times. In the context of Ripple, every node may submit a transaction tx by invoking $submit(tx)$, and atomic broadcast applies tx

to the application state on the node through *execute*(*tx*). We recall the definition of the atomic broadcast below.

Definition 2.7 (Atomic Broadcast). A protocol for atomic broadcast satisfies the following properties:

Validity: If a correct party *a-broadcasts* a transaction *tx*, then every correct party eventually *a-delivers tx*.

No duplication: No transaction is *a-delivered* more than once.

Agreement: If a transaction *tx* is *a-delivered* by some correct party, then *tx* is eventually *a-delivered* by every correct party.

Total order: Let *tx* and tx' be two transactions such that p_i and p_j are correct parties that *a-deliver tx* and tx'. If p_i *a-delivers tx* before tx', then p_j also *a-delivers tx* before tx'.

Our specification does not refer to the heterogeneous trust structure defined by the UNLs and simply assumes all nodes should execute the same transactions. This corresponds to the implicit assumption in Ripple's code and documentation. We note that the question of establishing a global consistency in a distributed system with subjective trust structures is a topic of current research, as addressed by asymmetric quorum systems [22] or in the context of Stellar's protocol [62], for example.

3.3.2 Overview

The following description was obtained directly from the source code. Its overall structure retains many elements and function names found in the code so that it may serve as a guide to the source for others and to explain how it works. If the goal had been to compare Ripple consensus to the existing literature on synchronous Byzantine agreement protocols, the formalization would differ considerably.

The protocol is highly *synchronous* and relies on a common notion of time. It is structured into successive *rounds of consensus*, whereby each round agrees on a *ledger* (a set of transactions to execute). Each round roughly takes a predefined amount of time and is driven by a heartbeat timer, which triggers a state update once per second. This contrasts with the Byzantine consensus protocols with partial synchrony [33], such as PBFT [24], which can tolerate arbitrarily long periods of asynchrony and rely on clocks or timeouts only for liveness. The Ripple protocol

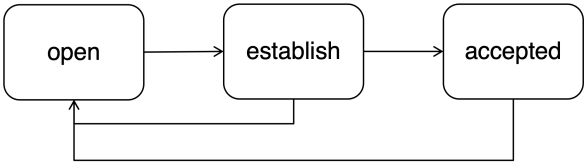


Figure 3.2: Phases and transitions between phases of a consensus round.

aims to agree on a transaction set within each synchronized round. The round ends when all nodes collectively declare to have reached consensus on a *proposal* for the round. The protocol is then said to *close* and later *validate* a ledger containing the agreed-on transaction set. However, the transactions in the ledger are executed only after another protocol step once the ledger has become *fully validated*; this occurs in an asynchronous process in the background. Transaction execution is only logically synchronized with the consensus round.

A *ledger* consists of a batch of transactions that result from a consensus round and contains a hash of the logically preceding ledger. Ledgers are stored persistently and roughly play the role of blocks in other blockchain protocols. Each node locally maintains three different ledgers: the *current ledger*, which is in the process of building during a consensus round, the *previous ledger*, representing the most recently closed ledger and the *valid ledger*, which is the last fully validated ledger in the network.

In more detail, a consensus round has three *phases*: *open*, *establish*, and *accepted*. According to the state diagram shown in Figure 3.2, the usual phase transition goes from *open* to *establish* to *accepted* and then proceeded to the next consensus round, which starts again from *open*. However, it is also possible that the phase changes from *establish* to *open*, if a node detects that it has been forked from the others to a wrong ledger and resume processing after switching to the ledger agreed by the network.

Nodes may submit transactions at any time concurrently to executing the consensus rounds. They are disseminated among the nodes through a *gossip layer* that ensures only weak consistency. All transactions that have been received from gossip are placed into a buffer. Apparently, the original design assumed that the gossip layer ensures consistency that prevents Byzantine nodes from equivocating in the sense that correct nodes never receive different messages from them. This assumption has

been dropped later [26].

The protocol rounds and their phases are implemented by a state machine invoked every second when the global *heartbeat* timer ticks. Messages from other nodes are received asynchronously in the background and processed during the next timer interrupt.

The timeout handler (L56) first checks if the local *previous ledger* is the same as the *preferred ledger* of a sufficient majority of the nodes in the network. If not, the node has been forked or lost synchronization with the rest of the network and must bring itself back to the state agreed by the network. In this case, it starts a new consensus round from scratch.

When the node enters a new round of consensus, it sets the phase to *open*, resets round-specific data structures and simply waits for the buffer to fill up with submitted transactions. Once the node has been in the *open* phase for more than half of the duration of the previous consensus round, the node moves to the *establish* phase (L63–L64; function *closeLedger*). It locally closes the ledger, which means to initialize its proposal for the consensus round and to send this to the other nodes in its UNL.

During the *establish* phase, the nodes exchange their proposals for the transactions to decide in this consensus round (using PROPOSAL messages). Obviously, these proposals may contain different transaction sets. All transactions on which the proposals from other nodes differ become *disputed*. Every node keeps track of how many other nodes in its UNL have proposed a disputed transaction and represents this information as *votes* by the other nodes. The node may remove a disputed transaction from its proposal or add one to its proposal based on the votes of the others and based on the time that has passed. Specifically, the node increases the necessary threshold of votes for changing its vote on a disputed transaction depending on the duration of the *establish* phase concerning the time taken by the previous consensus round.

The node leaves the *establish* phase when it has found that there is a consensus on its proposal (L69–L71; functions *haveConsensus* and *onAccept*). The node constructs the next ledger (the "last closed ledger") by "applying" the decided transactions. This ledger is signed and broadcast to the other nodes in a VALIDATION message.

The node then moves to the *accepted* phase and initializes a new consensus round immediately. Concurrently, the node receives VALIDATION messages from the nodes in its UNL. It verifies them and counts how many other nodes in its UNL have issued the same validation. When

this number reaches 80% of the nodes in its UNL, the ledger becomes fully validated, and the node executes the transactions contained in it.

3.3.3 Details

Phase open. The function *beginConsensus* starts a consensus round for the next ledger (L50). Each ledger (L11) contains a hash (*ID*) that serves as its identifier, a sequence number (*seq*), a hash of the parent ledger (*parentID*), and a transaction set (*txns*), denoting the transactions applied by the ledger.

The node records the time when the *open* phase started (*openTime*, L54), so that it can later calculate how long the *open* phase has taken. This is important because the duration of the *open* phase determines when to close the ledger locally. If the time that has passed since *openTime* is longer or equal to half of the previous round time (*prevRoundTime*), consensus moves to phase *establish* by calling the function *closeLedger* (L64). Meanwhile, all nodes submit transactions with the gossip layer (L46), and each node stores the transaction received via gossip messages in its transaction set S (L48). We model transactions as bit strings. In some places, and as in the source code, we use a short, unique *transaction identifier* (of type int) for each transaction $tx \in \{0,1\}^*$, computed by a function $TxID(tx)$. A transaction set is a set of binary strings here, but the source code maintains a transaction set using a hash map containing the transaction data indexed by their identifiers.

Phase establish. When the node moves from *open* to *establish*, it calls *closeLedger* that creates an initial proposal (stored in *result.proposal*) containing all transactions received from the gossip layer (L79) that have not been executed yet. A proposal structure (L16) contains the hash of the previous ledger (*prevLedgerID*), a sequence number (*seq*), the actual set (*txns*) of proposed transactions (in the source code named *position*), an identifier of the node (*node*) that created this proposal, and a timestamp (*time*) when this proposal is created (L79).

The node then broadcasts the new proposal as a PROPOSAL message (L81) to all nodes in its UNL. When they receive it, they will store its contents in their *currPeerProposals* collection of proposals (L86), if the message originates from a node in their respective UNL. The *closeLedger* function also sets *result.roundTime* to the current time (L80). This

measures the duration of the *establish* phase and will be used later to determine how far the consensus process has converged.

Based on the proposals from other nodes, each node computes a set of disputed transactions (L89). A disputed transaction (DisputedTx, L5) contains the transaction itself (*tx*), a binary vote (*ourVote*) by the node on whether this transaction should be included in the ledger, the number of "yes" and "no" votes from other nodes on the transaction (*yays* and *nays*), taken from their PROPOSAL messages, and the list of votes on this transaction from the other nodes (*votes*).

A transaction becomes disputed when the node proposes it, and some other node does not, or vice versa. The node determines these by comparing its transaction set with the transaction sets of all other nodes (L91). Every disputed transaction is recorded (as a DisputedTx structure) in the collection *result.disputes* (L92–L101).

During the *establish* phase, the node constantly updates its votes on all disputed transactions (L68; L103; L124) for responding to further PROPOSAL messages that have been received. A vote may change based on the number of nodes in favor of the transaction, the *convergence ratio* (*converge*) and a *threshold*. Convergence measures the expected progress in one single consensus round and is computed from the duration of the *establish* phase, the duration of the the previous round, and an assumed maximal consensus-round time (L67). The value for the threshold is predefined. The further, the consensus converges, the higher is the threshold that the number of opposing votes needs to reach so that the node changes its own vote (L134). Whenever the node's proposal is updated, the node broadcasts its new proposal to the other nodes (L120) and the disputed transactions are recomputed (L122).

Afterward, the node checks if consensus on its proposed transaction set *result.txns* is reached by calling the function *haveConsensus* (L69). The node counts agreements (L138) and disagreements (L139) with *result.txns*. If the fraction of agreeing nodes is at least 80% with respect to the UNL (L141), then consensus is reached. The node proceeds to the *accepted* phase by calling the function *onAccept* (L71).

Phase accepted. The function *onAccept* (L142) "applies" the agreed-on transaction set and thereby creates the next ledger (called the "last closed ledger" in the source code; L143). This ledger is then signed (L145) and broadcast to the other nodes as a VALIDATION message (L146). This marks the end of the *accepted* phase, and the node ini-

tiates a new consensus round (L149).

Meanwhile, in the background, the node receives VALIDATION messages from other nodes in its UNL and tries to verify them (L150). This verification checks the signature and if the sequence number of the received ledger is the same as the sequence number of the own ledger. All validations that satisfy both conditions and contain the node's own agreed-on ledger are counted (L155); this comparison uses the cryptographic hash of the ledger structure in the source code. Again, if 80% of the nodes have validated the same ledger, and if the sequence number of that ledger is larger than that of the last fully validated ledger (L156), the ledger becomes fully validated (L157). The node executes the transactions in the ledger (L160). In other words, the consensus decision has become final.

Preferred ledger. A node participating in consensus regularly computes the *preferred ledger*, which denotes the current ledger on which the network has decided. Due to possible faults and network delays, the node's *prevLedger* may have diverged from the preferred ledger, which is determined by calling the function *getPreferred(validLedger)* (L161). Should the network have adopted a different ledger than the *prevLedger* of the node, the node switches to this ledger and restarts the consensus round with the new ledger.

Notice that the validated ledgers from all correct nodes form a tree rooted in the initial ledger (*genesisLedger*). Each node stores all valid ledgers it receives in a tree-structured variable *tree*. Whenever the node receives a VALIDATION message containing a ledger L', it adds L' to *tree* (L152). In order to compute the preferred ledger, we define the following functions, which are derived from the ledgers in *tree* and in the received VALIDATION messages:

- $\mathit{tip\text{-}support}(L)$ for a ledger L is the number of validators in the *UNL* that have validated L. In other words,

$$\mathit{tip\text{-}support}(L) = \big|\{\ j \in \mathit{UNL} \mid \mathit{validations}[j] = L\}\big|.$$

- $\mathit{support}(L)$ for a given ledger L is the sum of the tip support of L and all its descendants in *tree*, i.e.,

$$\mathit{support}(L) = \mathit{tip\text{-}support}(L) + \sum_{L' \text{ is a child of } L \text{ in } \mathit{tree}} \mathit{tip\text{-}support}(L').$$

- *uncommitted*(L) for a ledger L denotes the number of validators whose last validated ledger has a sequence number that is strictly smaller than the sequence number of L. More formally,

$$\mathit{uncommitted}(L) = \big|\{j \in \mathit{UNL} \mid \mathit{validations}[j].\mathit{seq} < L.\mathit{seq}\}\big|.$$

With these definitions, we now explain how *getPreferred(Ledger L)* proceeds (L161–L172). If L has no children in *tree*, it returns L itself. Otherwise, the function considers the child of L that has the highest support among all children (M). If the support of M is still smaller than the number of validators yet uncommitted at this ledger-sequence number, then L is still the preferred ledger (L167). Otherwise, if the support of M is guaranteed to exceed the support of any of its siblings N, even when the uncommitted validators would also support N, then the function recursively calls *getPreferred* on M, which outputs the preferred ledger for M and returns this as the preferred ledger for L. Otherwise, L itself is returned as the preferred ledger. Observe that when M has no siblings, conditions in L166 and L168 are equivalent. Then, it is enough to check if support of M is greater than uncommitted od M.

Functions. For the simplicity of pseudocode, some functions are not fully explained. These functions are:

- *startTimer*(*timer*, *duration*) starts *timer*, which expires after the time passed as *duration*.
- *clock.now()* returns the current time.
- *Hash* creates a unique identifier (often denoted *ID*) of a data structure by converting the data to a canonical representation and applying a cryptographic hash function to this.
- $\mathcal{A} \triangle \mathcal{B}$ denotes the symmetric set difference.
- *boolToInt*(b) converts a logical value b to an integer and returns *b? 0:1*.
- $\mathit{sign}_i(L)$ creates a cryptographic digital signature for ledger L by node i.
- $\mathit{verify}_i(L, \sigma)$ checks if the digital signature on L from node i is valid.

- *siblings(M)* returns the set of nodes, different from M, that have the same parent as M.

Remarks on the pseudocode. Next to every function name, a comment points to a specific file and line in the source code that contains its implementation. The Ripple source contains a large number of files, and most of the consensus protocol implementation is actually spread over multiple header (.h) files, which complicates the analysis of the code. The references in this work are based on version 1.4.0[1] of `rippled` [81].

3.4 Violation of safety

In this section, we address the safety of the Ripple consensus protocol. We first describe a simple scenario in which consensus is violated in an execution with seven nodes, one Byzantine. Secondly, we show how this problem can be generalized to executions with more nodes.

3.4.1 Violating agreement with seven nodes

We use the following scenario with seven nodes to show that the Ripple consensus protocol violates safety and may let two correct nodes execute different transactions. Figure 3.3 gives a graphical representation of our scenario, and we will refer to it later in the text.

Nodes are named by numbers. We let $UNL_1 = \{1, 2, 3, 4, 5\}$ and $UNL_2 = \{3, 4, 5, 6, 7\}$, as illustrated by the two hatched areas in the figure. Nodes 1, 2, and 3 (white) trust UNL_1, nodes 5, 6, and 7 (black) trust UNL_2, and they are all correct; node 4 (gray) is Byzantine. With this setup, we achieve a 60% overlap between the UNLs of any two nodes.

The key idea is that the Byzantine node (4) changes its behavior depending on the group of nodes it communicates with. It will cause nodes 1, 2, and 3 (white) to propose some transaction $tx$and nodes 5, 6, and 7 (black) to propose a transaction tx' for the next ledger. No other transaction exists. The Byzantine node (4) follows the protocol as if it had proposed $tx$when interacting with the white nodes and behaves as if it had proposed tx' when interacting with the black nodes.

[1]This release was current at the time when this analysis was performed (2019-2020).

Algorithm 1 Ripple consensus protocol for node i

```
1: Type
2:     Enum Phase = {open, establish, accepted}
3:     Tx = {0,1}*                                              // a transaction
4:     TxSet = 2^Tx
5:     DisputedTx(                                              // DisputedTx.h:50
6:         Tx tx,                                               // disputed transaction
7:         bool ourVote,                                        // binary vote
8:         int yays,                                            // number of yes votes from others
9:         int nays,                                            // number of no votes from others
10:        HashMap[int → bool] votes)                           // collection of votes indexed by node
11:    Ledger(                                                  // Ledger.h:77
12:        Hash ID,                                             // identifier
13:        int seq,                                             // sequence number of this ledger
14:        Hash parentID,                                       // identifier of ledger's parent
15:        TxSet txns)                                          // set of transactions applied by ledger
16:    Proposal(                                                // ConsensusProposal.h:52
17:        Hash prevLedgerID,                                   // hash of the previous ledgerx
18:        int seq,                                             // sequence number
19:        TxSet txns,                                          // proposed transaction set
20:        int node,                                            // node that proposes this
21:        milliseconds time)                                   // time when proposal is created
22:    ConsensusResult(                                         // ConsensusTypes.h:201
23:        TxSet txns,                                          // set of transactions consensus agrees on
24:        Proposal proposal,                                   // proposal containing transaction set
25:        HashMap[int → DisputedTx] disputes,                  // disputed transactions
26:        milliseconds roundTime)                              // duration of the establish phase

27: State
28:    Phase phase                                              // phase of the consensus round for agreeing on one ledger
29:    Tree tree                                                // tree representation of received valid ledgers
30:    Ledger L                                                 // current working ledger
31:    Ledger prevLedger                                        // last agreed-on ("closed") ledger
32:    Ledger validLedger                                       // ledger that was most recently fully validated
33:    TxSet S                                                  // submitted transactions that have not yet been executed
34:    ConsensusResult result                                   // data relevant for the outcome of consensus
35:    HashMap[int → Proposal] currPeerProposals                // collection of proposals
36:    HashMap[int → Ledger] validations                        // collection of validations
37:    milliseconds prevRoundTime                               // initialized to 15s
38:    float converge ∈ [0,1]                                   // ratio of round time to prevRoundTime
39:    UNL ⊆ {1,...,M}                                          // validator nodes trusted by node i
40:    milliseconds openTime                                    // time when the last open phase started

41: function initialization()
42:    prevLedger ← genesisLedger                               // the first ledger in the network
43:    S ← {}
44:    beginConsensus()                                         // start the first round of consensus
45:    startTimer(heartbeat, 1s)                                // NetworkOPs.cpp:673
```

Algorithm 2 Ripple consensus protocol for node i (continued)

46: **upon** submission of a transaction tx**do**
47: send message [SUBMIT, tx] with the gossip layer

48: **upon** receiving a message [SUBMIT, tx] from the gossip layer **do**
49: $S \leftarrow S \cup \{tx\}$

50: **function** *beginConsensus()* // Consensus.h:663
51: $phase \leftarrow open$ // Consensus.h:669
52: $result \leftarrow (\{\}, \perp, [\,], 0)$ // Consensus.h:674
53: $converge \leftarrow 0$ // Consensus.h:675
54: $openTime \leftarrow clock.now()$ // save the time when this round started
55: $currPeerProposals \leftarrow [\,]$ // reset the proposals for this consensus round

56: **upon** *timeout*(*heartbeat*) **do** // Consensus.h:818
57: $L' \leftarrow getPreferred(validLedger)$
58: **if** $L' \neq prevLedger$ **then**
59: $prevLedger \leftarrow L'$
60: *beginConsensus(prevLedger)*
61: **if** $phase = open$ **then** // Consensus.h:829
62: **if** $(clock.now() - openTime) \geq \frac{prevRoundTime}{2}$ **then**
63: $phase \leftarrow establish$
64: *closeLedger()* // initialize consensus value in *result*
65: **else if** $phase = establish$ **then** // Consensus.h:833
66: $result.roundTime \leftarrow clock.now() - result.roundTime$
67: $converge \leftarrow \frac{result.roundTime}{\max\{prevRoundTime, 5s\}}$
68: *updateOurProposals()* // update consensus value in *result*
69: **if** *haveConsensus()* **then**
70: $phase \leftarrow accepted$
71: *onAccept()* // note this immediately sets $phase = open$
72: **else if** $phase = accepted$ **then** // Consensus.h:821
73: // do nothing
74: *startTimer*(*heartbeat*, $1s$)

75: // transition from *open* to *establish* phase
76: **function** *closeLedger()* // Consensus.h:1309
77: $L \leftarrow (\perp, prevLedger.seq + 1, \perp, \{\})$
78: $result.txns \leftarrow S$ // propose the current set of submitted transactions
79: $result.proposal \leftarrow (Hash(prevLedger), 0, result.txns, i, clock.now())$
80: $result.roundTime \leftarrow clock.now()$
81: broadcast message [PROPOSAL, *result.proposal*]
82: // a dispute exists for a transaction not proposed by all UNL nodes
83: $result.disputes \leftarrow [\,]$
84: **for** $j \in UNL$ **such that** $currPeerProposals[j] \neq \perp$ **do**
85: *createDisputes*($currPeerProposals[j].txns$) // Consensus.h:1334

86: **upon** receiving a message [PROPOSAL, *prop*] **such that** $prop = (nl, \cdot, \cdot, j, \cdot)$
87: **and** $j \in UNL$ **and** $nl = Hash(prevLedger)$ **do**
88: $currPeerProposals[j] \leftarrow prop$ // Consensus.h:781

Algorithm 3 Ripple consensus protocol for node i (continued)

```
89: function createDisputes(TxSet set)                          // Consensus.h:1623
90:     // all transactions that differ between result.txns and set
91:     for tx ∈ result.txns △ set do
92:         dt ← (tx, (tx ∈ result.txns), 0, 0, [])    // dt is a disputed transaction
93:         for k ∈ UNL such that currPeerProposals[k] ≠ ⊥ do
94:             if tx ∈ currPeerProposals[k].txns then
95:                 // records node's vote for the disputed transaction
96:                 dt.votes[k] ← 1
97:                 dt.yays ← dt.yays + 1
98:             else
99:                 / records node's vote against the disputed transaction
100:                dt.votes[k] ← 0
101:                dt.nays ← dt.nays + 1
102:        result.disputes[TxID(tx)] ← dt

    // phase establish
103:function updateOurProposals()                                // Consensus.h:1361
104:    for j ∈ UNL
105:        such that (clock.now() − currPeerProposals[j].time) > 20s do
106:            currPeerProposals[j] ← ⊥              // remove stale proposals
107:    // current set of transactions, to update from disputed ones
108:    T ← result.txns
109:    for dt ∈ result.disputes do                    // dt is a disputed transaction
110:        if updateVote(dt) then
111:            dt.ourVote ← ¬dt.ourVote
112:            if dt.ourVote then     // should the transaction be included?
113:                T ← T ∪ {dt.tx}
114:            else
115:                T ← T \ {dt.tx}
116:    if T ≠ result.txns then              // if txns changed, then update result
117:        result.txns ← T
118:        result.proposal ← (Hash(prevLedger), result.proposal.seq + 1,
119:                           result.txns, i)
120:        broadcast message [PROPOSAL, result.proposal]
121:        result.disputes ← [ ]              // recompute disputes after updating
122:        for j ∈ UNL such that currPeerProposals[j] ≠ ⊥ do
123:            createDisputes(currPeerProposals[j].txns)

124:function updateVote(DisputedTx dt)                           // DisputedTx.h:197
125:    // set threshold based on duration of the establish phase
126:    if converge < 0.5 then
127:        threshold ← 0.5
128:    else if converge < 0.85 then
129:        threshold ← 0.65
130:    else if converge < 2 then
131:        threshold ← 0.7
132:    else
133:        threshold ← 0.95
134:    newVote ← ( (dt.yays + boolToInt(dt.ourVote)) / (dt.yays + dt.nays + 1) > threshold )
135:    return (newVote ≠ dt.ourVote)                       // the vote changes
```

Algorithm 4 Ripple consensus protocol for node i (continued)

136:**function** *haveConsensus()* // Consensus.h:1545
137: // count number of agreements and disagreements with our proposal
138: $agree \leftarrow |\{j | currPeerProposals[j] = result.proposal\}|$
139: $disagree \leftarrow |\{j | currPeerProposals[j] \neq \perp \wedge$
140: $currPeerProposals[j] \neq result.proposal\}|$
141: **return** $\left(\frac{agree+1}{agree+disagree+1} \geq 0.8\right)$ // 0.8 is defined in ConsensusParams.h

// phase *accepted*
142:**function** *onAccept()* // RCLConsensus.cpp:408
143: $L \leftarrow (prevLedger, result.txns)$ // L is the last closed ledger
144: $validations[i] \leftarrow L$
145: $\sigma \leftarrow sign_i(L)$ // validate the ledger, RCLConsensus.cpp:743
146: broadcast message $[\text{VALIDATION}, i, \sigma, L]$
147: $prevLedger \leftarrow L$ // store the last closed ledger
148: $prevRoundTime \leftarrow result.roundTime$
149: *beginConsensus()* // advance to the next round of consensus

150:**upon** receiving a message $[\text{VALIDATION}, j, \sigma, L']$ **such that**
151: $L'.seq = L.seq$ **and** $verify_j(L', \sigma)$ **do**
152: add L' to *tree*
153: $validations[j] \leftarrow L'$ // store received validation
154: // count the number of validations
155: $valCount \leftarrow |\{k \in UNL | validations[k] = L\}|$
156: **if** $valCount \geq 0.8 \cdot |UNL|$ **and** $L.seq > validLedger.seq$ **then**
157: $validLedger \leftarrow L$ // ledger becomes fully validated
158: $S \leftarrow S \setminus \{L.txns\}$
159: **for** $tx \in L.txns$ **do** // in some deterministic order
160: *execute(tx)*

161:**function** *getPreferred(Ledger L)* // LedgerTrie.h:677
162: **if** L is a leaf node in *tree* **then**
163: **return** L
164: **else**
165: $M \leftarrow \arg\max\{support(N) \mid N \text{ is a child of } L \text{ in the } tree\}$
166: **if** $uncommitted(M) \geq support(M)$ **then**
167: **return** L
168: **else if** $\max\{support(N) \mid N \in siblings(M)\}$
169: $+uncommitted(M) < support(M)$ **then**
170: **return** *getPreferred(M)*
171: **else**
172: **return** L

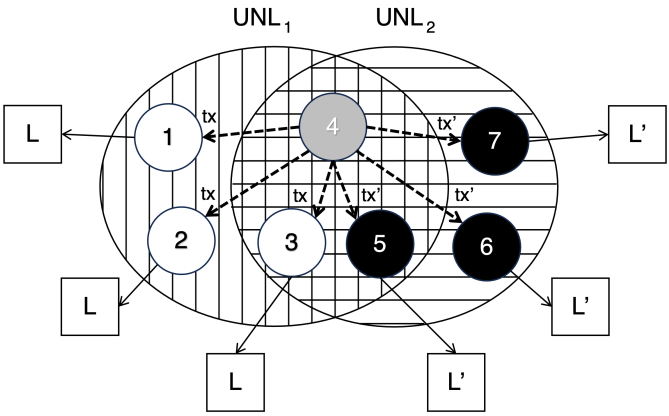


Figure 3.3: Example setup for showing a safety violation in the Ripple consensus protocol. The setup consists of seven nodes, one of them Byzantine, and two UNLs. Nodes 1, 2, and 3 (white) adopt UNL_1, vertically hatched, and nodes 5, 6, and 7 adopt UNL_2, horizontally hatched. Node 4 (gray) is Byzantine.

Assuming that all nodes start the consensus roughly at the same time and they do not switch the preferred ledger, the protocol does the following:

- The Byzantine node 4 submits tx and tx' using gossip and causes [SUBMIT, tx] to be received by nodes 1, 2, and 3 and [SUBMIT, tx'] to be received by nodes 5, 6, and 7 from the gossip layer. During the repeated *heartbeat* timer executions in the *open* phase, all correct nodes have the same value of *prevLedger* and send no further messages.

- Suppose at a common execution of the *heartbeat* timer execution (L56) all correct nodes proceed to the *establish* phase and call *closeLedger*. They broadcast the message [PROPOSAL, S], with S containing tx or tx', respectively (L81). Node 4 sends a PROPOSAL message containing tx to nodes 1, 2, and 3 and one containing tx' to nodes 5-7. Furthermore, every correct node executes *createDisputes* with the transaction set *txns* received in each PROPOSAL message, which creates *result.disputes* (L89). For nodes 1, 2, and 3, transaction tx' is disputed, and for nodes 5, 6, and 7,

transaction $tx$is disputed.

- During *establish* phase, all nodes update their vote for each disputed transaction (L124). Nodes 1, 2, and 3 consider tx' but do not change their *no* vote on tx' because only 20% of nodes in their UNL (namely, node 5) vote *yes* on tx'; this is less than required *threshold* of 50% or more (L134). The same holds for nodes 5, 6, and 7 with respect to transaction tx. Hence, *result.txns* remains unchanged, and no correct node sends another PROPOSAL message.

- Eventually, function *haveConsensus* returns TRUE for each correct node because the required $4/5 = 80\%$ of its *UNL* has issued the same proposal as the node itself (L136). Every correct node moves to the *accepted* phase.

- During *onAccept*, nodes 1, 2, and 3 send a VALIDATION message with ledger $L = (prevLedger, \{tx\})$, whereas nodes 5, 6, and 7 send a VALIDATION message containing $L' = (prevLedger, \{tx'\})$ (L146). Node 4 sends a VALIDATION message containing tx to nodes 1, 2, and 3 and a different one, containing tx', to nodes 5, 6, and 7.

- Every correct node subsequently receives five validation messages from all nodes in its UNL and finds that 80% among them contain the same ledger (L150). Observe that no node changes its preferred ledger after calling *getPreferred*. This implies that nodes 1, 2, and 3 fully validate L and execute tx, whereas nodes 5, 6, and 7 fully validate L' and execute tx'. Hence, the agreement condition of consensus is violated.

3.4.2 Generalization

We now generalize the previous scenario and show a violation of the agreement with an arbitrarily large number of nodes. As illustrated in Figure 3.4, the system consists of $M = 2n + f$ nodes, such that nodes $1, \ldots, n$ (white) each submit transaction tx, nodes $n+1, \ldots, n+f$ (gray) are Byzantine, and nodes $n+f+1, \ldots, 2n+f$ (black) each submit transaction tx'. Assume all correct nodes have one of two different UNLs, namely $UNL_{tx} = \{1, \ldots, n+f+\tilde{n}\}$ or $UNL_{tx'} = \{n-\tilde{n}+1, \ldots, 2n+f\}$, each of size $n + f + \tilde{n}$. As the names suggest, nodes $1, \ldots, n$, which submit tx, use UNL_{tx} and nodes $n+f+1, \ldots, 2n+f$, which submit tx', use $UNL_{tx'}$.

The execution proceeds analogously to the one in the previous section, with nodes $1, \ldots, n$ behaving like nodes 1, 2, and 3, the f Byzantine nodes here behaving like node 4, and nodes $n+f+1, \ldots, 2n+f$ behaving like nodes 5, 6, and 7. The strategy of the Byzantines nodes is to follow the protocol, as if they had submitted transaction *tx* when they interact with correct nodes $1, \ldots, n$, and to behave as if they had submitted transaction tx' when they interact with the correct nodes $n+f+1, \ldots, 2n+f$.

Theorem 3.1. *A system of $2n + f$ nodes, of which f are Byzantine, running the Ripple consensus protocol according to the scenario defined above may violate safety if*

$$\frac{n+f}{n+\tilde{n}+f} \geq 0.8. \tag{3.1}$$

Proof. To prove that safety can be violated, it is enough to show that the strategy of the Byzantine nodes is successful. This follows from the same argument as in the previous scenario with seven nodes, according to the pseudocode in Section 3.3. The condition (3.1) corresponds to the test for fully validating a ledger (L156). □

The bound (3.1) of the theorem corresponds directly to the condition in the source code. We can reformulate this using $\omega = \frac{2\tilde{n}+f}{n+\tilde{n}+f}$ to denote the relative overlap of the UNLs (i.e., the fraction of nodes that are common between the two UNLs).

Corollary 3.2. *The Ripple consensus protocol may violate safety in a system of $2n + f$ nodes if*

$$f \;\geq\; n\,\frac{5\omega-2}{6-5\omega} \tag{3.2}$$

or equivalently, recalling that the total number of correct nodes is $2n$,

$$f \;\geq\; 2n\,\frac{5\omega-2}{12-10\omega}. \tag{3.3}$$

Proof. Equation (3.2) follows directly from (3.1) by substituting $\tilde{n}$ in terms of the overlap ω. Furthermore, (3.1) follows from (3.2) by replacing the *UNL* size through the total number of nodes. □

Corollary 3.2 illustrates the number of Byzantine nodes required to break the protocol's safety using the presented strategy. The number of Byzantine nodes required to show the violation is proportional to n, the number of correct nodes.

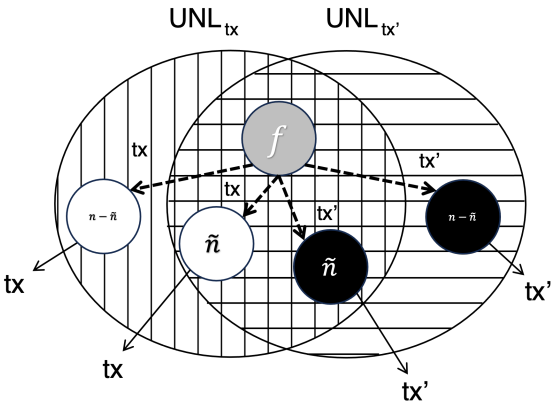


Figure 3.4: The generalized attack scenario with $2n + f$ nodes. The n white nodes submit tx and have UNL_{tx}, while the n black nodes submit tx' instead and have $UNL_{tx'}$. The f Byzantine nodes (gray) behave differently, depending on whether they interact with white and black nodes, respectively.

3.5 Violation of liveness

In this section, we show how the liveness of the Ripple consensus protocol may be violated, even when all nodes have the same UNL and only one node is Byzantine. One can bring the protocol to a state where it cannot produce a correct ledger and stops making progress.

Consider a system with $2n$ correct nodes and one single Byzantine node. All nodes are assumed to trust each other, i.e., there is one common UNL containing all $2n + 1$ nodes. Observe that in this system, the fraction of Byzantine nodes can be made arbitrarily small by increasing n.

As illustrated in Figure 3.5, node $n + 1$, which is Byzantine exhibits a split-brain behavior and follows the protocol for an input transaction $tx$when interacting with nodes $1, \ldots, n$, and operates with a different input transaction tx' when interacting with nodes $n + 2, \ldots, 2n + 1$. This implies that the first half of the correct nodes, denoted $1, \ldots, n$, will propose a transaction tx and the other half, nodes $n+2, \ldots, 2n+1$, will propose transaction tx'. Similar to the execution shown in Section 3.4.1, the nodes start the consensus protocol roughly at the same time, and

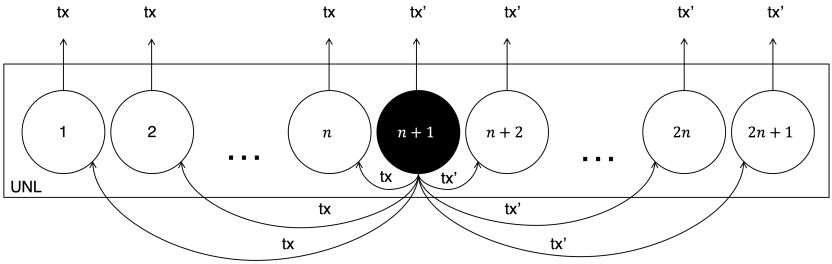


Figure 3.5: Setup in which liveness is violated in the Ripple network. The network consists of $2n+1$ nodes with one single UNL and 1 Byzantine (black). The n first nodes propose transaction tx while the last n propose transaction tx'. The Byzantine proposes a transaction tx to the n first nodes and transaction tx' to the last n.

they do not switch the preferred ledger. They proceed like this:

- Byzantine node $n+1$ sends two messages, [SUBMIT, tx] and [SUBMIT, tx'], using the gossip layer and causes $tx$to be received by the first n correct nodes and tx' to be received by the last n correct nodes.
- After some time has passed, the correct nodes start to close the ledger and move to the *establish* phase. Every correct node sends a PROPOSAL message, containing only the submitted transaction of which it knows (L81), namely $tx$for the first n correct nodes and tx' for the last n correct nodes.
- During *establish* phase, the correct nodes receive the PROPOSAL messages from all nodes (including the Byzantine node) and store them in *currPeerProposals* (L86). Since they all use the same UNL, all obtain the same PROPOSAL messages from the correct nodes.
- Each node creates disputes (L89) and updates them while more PROPOSAL messages arrive. Since the proposed transaction sets differ, each node creates a dispute for tx and for tx'.
- While the PROPOSAL messages are being processed, votes are counted in *updateVotes* (L124), using the *yays* and *nays* of each disputed transaction. For a correct node in $\{1, \ldots, n\}$, notice that the first n nodes and the Byzantine node vote *no* for tx' and the last n nodes

vote *yes*. Thus, the fraction of nodes voting *yes* for tx' is less than required threshold (50%), and so the first n nodes continue to vote *no* for tx'. Similarly, nodes $n+2$ to $2n+1$ never update their vote on *tx*and always vote *no* for *tx*.

- The *haveConsensus* function called periodically during the *establish* phase checks if at least 80% of the nodes in the UNL agree on the proposal of the node itself (L136). From the perspective of each one of the first n correct nodes, n other nodes agree, and n nodes disagree with its proposal, which contains *tx*. That is not enough support for achieving consensus, and the function will return FALSE. The same holds from the perspective of the last n correct nodes, which also continuously return FALSE.

- Finally, the correct nodes will continue trying to update votes and get enough support, but without being able to generate a correct ledger. No correct node proceeds to validating the ledger. In other words, the liveness of the protocol is not guaranteed.

3.6 Conclusion

Ripple is one of the oldest public blockchain platforms. For a long time, its native XRP token has been the third most valuable in terms of its total market capitalization. The Ripple network is implemented as a peer-to-peer network of validator nodes, which should reach consensus even in the presence of faulty or malicious nodes. Its consensus protocol is generally considered to be a Byzantine fault-tolerant protocol, but without global knowledge of all participating nodes and where a node only communicates with other nodes it knows from its UNL.

Previous work regarding the Ripple consensus protocol has already raised concerns about its liveness and safety. In order to better analyze the protocol, this work has presented an independent, abstract description derived directly from the implementation. Furthermore, this work has identified relatively simple cases in which the protocol may violate safety and/or liveness and have devastating effects on the network's health. Our analysis illustrates the need for very close synchronization, tight interconnection, and fault-free operations among the participating validators in the Ripple network.

Chapter 4

Quick Order Fairness: Definition and protocol

4.1 Introduction

The nascent field of *decentralized finance* (or simply *DeFi*) suffers from insider attacks: Malicious miners in permissionless blockchain networks or Byzantine leaders in permissioned atomic broadcast protocols have the power of selecting transactions that go into the ledger and determining their final order. Selfish participants may also insert their own, fraudulent transactions and thereby extract value from the network and its innocent users. For instance, a decentralized exchange can be exploited by *front-running*, where a genuine transaction tx carrying an exchange transaction is *sandwiched* between a transaction tx_{before} and a transaction tx_{after}. If tx buys a particular asset, the insider acquires it as well using tx_{before} and sells it again with tx_{after}, typically at a higher price. Such front-running and other price-manipulation attacks represent a serious threat. They are prohibited in traditional finance systems with centralized oversight but must be prevented technically in DeFi. Daian *et al.* [30] have coined the term *miner extractable value (MEV)* for the profit that can be gained from such arbitrage opportunities.

The traditional properties of *atomic broadcast*, often somewhat imprecisely called *consensus* as well, guarantee a total order: that all correct parties obtain the same sequence of transactions and that any transaction submitted to the network by a client is delivered in a reasonable

lapse of time. However, these properties do not further constrain *which* order is chosen, and malicious parties in the protocol may therefore manipulate the order or insert their own transactions to their benefit. Kelkar *et al.* [50] have recently introduced the new safety property of *order fairness* that addresses this in the Byzantine model. Kursawe [54] and Zhang *et al.* [91] have formalized this problem as well and found different ways to tackle it, relying on somewhat stronger assumptions.

Intuitively, *order fairness* aims at ensuring that transactions received by "many" parties are scheduled and delivered earlier than transactions received by "few" parties. The *Condorcet paradox* demonstrates, however, that such preference votes can easily lead to cycles even if the individual votes of majorities are not circular. The solution offered through *order fairness* [50] may therefore output multiple transactions *together as a set* (or batch), such that there is *no order* among all transactions in the same set. Kelkar *et al.* [50] name this property *block-order fairness* but calling such a set a "block" may easily lead to confusion with the low-level blocks in mining-based protocols.

In this work, we investigate order fairness in networks with n parties of which f are faulty for asynchronous and eventually synchronous atomic broadcast. This covers the vast majority of relevant applications since timed protocols that assume synchronous clocks and permanently bounded transaction delays have largely been abandoned in this space.

We first revisit the notion of block-order fairness [50]. In our interpretation, this requires that when n correct parties broadcast two transactions tx and tx', and $\overline{\gamma}n$ of them broadcast tx before tx' for some $\overline{\gamma} > \frac{1}{2}$, then tx' is not delivered by the protocol before tx, although both transactions may be output together. This guarantee is difficult to achieve in practice because Kelkar *et al.* [50] show that for the relevant values of $\overline{\gamma}$ approaching one half, the resilience of any protocol decreases. Tolerating only a small number of faulty parties seems prohibitive in realistic settings.

More importantly, we show that $\overline{\gamma}$ cannot be too close to $\frac{1}{2}$ because $\overline{\gamma} \geq \frac{1}{2} + \frac{f}{n-f}$ is necessary for any protocol. This result follows from establishing a link to the differential validity notion of consensus, formalized by Fitzi and Garay [38]. Notice that block-order fairness is a relative measure. We are convinced that a differential notion is better suited to address the problem. We, therefore, overcome this inherent limitation of relative order fairness by introducing *differential order fairness*: When the number of correct parties that broadcast a transaction tx before a

transaction tx' exceeds the number that broadcast tx' before tx by more than $2f + \kappa$, for some $\kappa \geq 0$, then the protocol must *not* deliver tx' before tx (but they may be delivered together). This notion takes into account existing results on differential validity for consensus [38]. In particular, when the difference between how many parties prefer one of tx and tx' over the other is smaller than $2f$, then *no protocol exists* to deliver them in fair order.

Last but not least, we introduce a new protocol, called *quick order-fair atomic broadcast*, that implements differential order fairness and is much more efficient than the previously existing algorithms. In particular, it works with optimal resilience $n > 3f$, requires $O(n^2)$ messages to deliver one transaction on average and needs $O(n^2L + n^3\lambda)$ bits of communication, with transactions of up to L bits and cryptographic λ-bit signatures. This holds for *any* order-fairness parameter κ. For comparison, the asynchronous Aequitas protocol [50] has resilience $n > 4f$ or worse, depending on its order-fairness parameter, and needs $O(n^4)$ messages.

To summarize, the contributions of this work are as follows:

- It illustrates some *limitations* that are inherent in the notion of block-order fairness (Section 4.4.1).
- It introduces *differential order fairness* as a measure for defining fair order in atomic broadcast protocols (Section 4.4.2).
- It presents the *quick order-fair atomic broadcast protocol* for differentially order-fair Byzantine atomic broadcast with optimal resilience $n > 3f$ (Section 4.5).
- It demonstrates that the quick order-fairness protocol has quadratic amortized transaction complexity, which is an n^2-fold improvement compared to the most efficient previous protocol for the same task (Section 4.5.3).

The work starts with a review of previous work (Section 4.2) and by describing our system model (Section 4.3).

4.2 Related work

Over the last decades, extensive research efforts have explored the state-machine replication problem. A large number of papers refer to this

problem, but only a few of them consider fairness in the order of delivered transactions. In this section, we review the related work on fairness.

Kelkar *et al.* [50] introduce a new property called *transaction order-fairness* which prevents adversarial manipulation of the ordering of transactions. They investigate assumptions needed for achieving this property in a permissioned setting and formulate a new class of consensus protocols, called Aequitas, that satisfy order fairness. A subsequent paper by Kelkar *et al.* [47] extends this approach to a permissionless setting. Recently, Kelkar *et al.* [49] presented another permissioned Byzantine atomic-broadcast protocol called Themis. It introduces a new technique called *deferred ordering*, which overcomes a liveness problem of the Aequitas protocols.

Kursawe [54] and Zhang *et al.* [91] have independently postulated alternative definitions of order fairness, called *timed relative fairness* and *ordering linearizability*, respectively. Both notions are strictly weaker than order fairness of transactions, however [47]. Timed relative fairness assumes that all parties have access to synchronized local clocks; it can ensure that if all correct parties saw transaction tx to be ordered before tx', then tx is scheduled and delivered before tx'. Similarly, ordering linearizability says that if the highest timestamp provided by any correct party for a transaction tx is lower than the lowest timestamp provided by any correct party for a transaction tx', then tx will appear before tx' in the output sequence. The implementation of ordering linearizability [91] uses a median computation, which can easily be manipulated by faulty parties [47].

The Hashgraph [9] consensus protocol also claims to achieve fairness. It uses gossip internally and all parties build a *hash graph* reflecting all of the gossip events. However, there is no formal definition of fairness and the presentation fails to recognize the impossibility of fair transaction-order resulting from the *Condorcet paradox*. Kelkar *et al.* [50] also show an attack that allows a malicious party to control the order of the transactions delivered by Hashgraph.

A complementary measure to prevent transaction-reordering attacks relies on threshold cryptography [19], [32], [80]: clients encrypt their input (transaction) transactions under a key shared by the group of parties running the atomic broadcast protocol. They initially order the encrypted transactions and subsequently collaborate for decrypting them. Hence, their contents become known only *after* the transaction order has been decided. For instance, the Helix protocol [8] implements this approach and additionally exploits in-protocol randomness for two ad-

ditional goals: to elect the parties running the protocol from a larger group and to determine which transactions among all available ones must be included by a party when proposing a block. This method provides resistance to censorship but still permits some order-manipulation attacks.

4.3 System model and preliminaries

4.3.1 System model

Processes. We model our system as a set of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$, also called *parties*, that communicate with each other. Parties interact with each other by exchanging transactions reliably in a network. A protocol for $\mathcal{P}$ consists of a collection of programs with instructions for all parties. Parties are computationally bounded and protocols may use cryptographic primitives, in particular, digital signature schemes.

Failures. In our model, we distinguish two types of parties. Parties that follow the protocol as expected are called *correct*. Contrary, the parties that deviate from the protocol specification or may crash are called *Byzantine*.

Communication. We assume that there exists a low-level mechanism for sending messages over reliable and authenticated point-to-point links between parties. In our protocol implementation, we describe this as "sending a message" and "receiving a message". Additionally, we assume *first-in first-out (FIFO) ordering* (Def. 2.3) for the links. This ensures that transactions broadcast by the same correct party are delivered in the order in which they were sent by a correct recipient.

Timing. This work considers two models, *asynchrony* and *partial synchrony*. Together they cover most scenarios used today in the context of secure distributed computing. In an *asynchronous* network, no physical clock is available to any party and the delivery of transactions may be delayed arbitrarily. In such networks, it is only guaranteed that a transaction sent by a correct party will *eventually* arrive at its destination. One can define asynchronous time based on logical clocks. A *partially synchronous* network [33] operates asynchronously until some point in time (not known to the parties), after which it becomes stable.

This means that processing times and transaction delays are bounded afterwards, but the maximal delays are not known to the protocol.

4.3.2 Byzantine FIFO consistent broadcast channel

We are using a Byzantine FIFO consistent broadcast channel (Def. 2.5) that allows the parties to deliver multiple transactions and ensures a notion of consistency despite Byzantine senders. The interface of such a channel provides two events involving transactions from a domain $\mathcal{T}$:

- A party invokes *bcch-broadcast*(tx) to broadcast a transaction $tx \in \mathcal{T}$ to all parties.
- An event *bcch-deliver*(p_j, l, tx) delivers a transaction $tx \in \mathcal{T}$ with label $l \in \{0,1\}^*$ from a party p_j.

The label that comes with every delivered transaction is an arbitrary bit string generated by the channel. Intuitively, the channel ensures that if a transaction is delivered with some label, then the transaction itself is the same at all correct parties that deliver this label. We recall the definition below.

Definition 2.5 (Byzantine FIFO Consistent Broadcast Channel). A Byzantine FIFO consistent broadcast channel satisfies the following properties:

Validity: If a correct party broadcasts a transaction tx, then every correct party eventually delivers tx.

No duplication: For every party p_j and label l, every correct party delivers at most one transaction with label l and sender p_j.

Integrity: If some correct party delivers a transaction tx with sender p_j and party p_j is correct, then tx was previously broadcast by p_j.

Consistency: If some correct party delivers a transaction tx with label l and sender p_j, and another correct party delivers a transaction tx' with label l and sender p_j, then $tx = tx'$.

FIFO delivery: If a correct party broadcasts some transaction tx before it broadcasts a transaction tx', then no correct party delivers tx' unless it has already delivered tx.

This primitive can be implemented by running, for every sender p_i, a sequence of standard consistent Byzantine broadcast instances (Def. 2.4) such that exactly one instance in each sequence is active at every moment. Each consistent broadcast instance is identified by a per-sender sequence number. When an instance delivers a transaction, the protocol advances the sequence number and initializes the next instance. The sequence number serves as the label. Details of this protocol are described by Cachin *et al.* [18, Sec. 3.12.2]; notice that their protocol also ensures FIFO delivery, although this is not explicitly mentioned there.

In addition to the *bcch-broadcast* and *bcch-deliver* events, in our protocol we use the following methods to access the BCCH primitive: *bcch-create-proof* and *bcch-verify-proof*. Those methods ensure that missing transactions can be transferred in a verifiable way, and they are implemented as in the protocol for verifiable consistent broadcast by Cachin *et al.* [19]. The input of *bcch-create-proof* is a list of transactions and it outputs a string s that contains a proof along with the list of transactions to be sent. A party that receives a transaction providing s can input this in *bcch-verify-proof* to verify the proof contained in s such that it is impossible to forge a proof for a transaction that was not *bcch-delivered*.

Another two methods, *bcch-get-length* and *bcch-get-transactions*, are used to get the number of sent transactions and to extract them.

4.3.3 Validated Byzantine consensus

Validated Byzantine consensus (Def. 2.6) defines an *external validity* condition. It requires that the consensus value is legal according to a global, efficiently computable predicate P, known to all parties. This allows the protocol to recognize proposed values that are acceptable to an external application. Note that it is not required that the decision value was proposed by a correct party, but all parties must be able to verify the validity. A consensus primitive is accessed through the events *vbc-propose*(v) and *vbc-decide*(v), where $v \in \mathcal{V}$ has a potentially large domain $\mathcal{V}$ and may contain a proof, which allows parties to verify the validity of v. We recall the definition below.

Definition 2.6 (Validated Byzantine Consensus). A protocol solves validated Byzantine consensus with validity predicate P if it satisfies the following conditions:

Termination: Every correct party eventually decides some value.

Integrity: No correct party decides twice.

Agreement: No two correct parties decide differently.

External validity: Every correct party only decides a value v such that $P(v) = \text{TRUE}$. Moreover, if all parties are correct and propose v, then no correct party decides a value different from v.

We intend this notion to cover asynchronous protocols, which actually only terminate probabilistically, as well as eventually synchronous protocols. The difference is not essential to our use of them.

Originally, *external validity* has been defined for *asynchronous* multi-valued Byzantine consensus, which requires randomized implementations [19]. But the property applies equally to consensus protocols with *partial synchrony*.

Among the asynchronous protocols, recent work by Abraham *et al.* [1] improves the expected communication (bit) complexity to $O(Ln^2)$ from $O(Ln^3)$ in the earlier work [19], where L is the maximal length of a proposed value.

In Dumbo-MVBA [64] the communication complexity of this primitive is further reduced to $O(Ln)$ through erasure coding, where the input of each party is split into coded fragments, distributed to every party, and recovered later.

Byzantine consensus protocols in the partial-synchrony model can easily be enhanced to provide external validity, when each party verifies P for every proposed value. For instance, the single-decision versions of PBFT [24] and of HotStuff [90] achieve best-case complexities $O(Ln^2)$ and $O(Ln)$, respectively; these values increase by a factor of n in the worst case.

4.3.4 Atomic broadcast

Atomic broadcast (Def. 2.7) ensures that all parties deliver the same transactions and that all transactions are output in the same order. This is equivalent to the parties agreeing on one sequence of transactions that they deliver. Atomic broadcast is also called "total-order broadcast" or simply "consensus" in the context of blockchains because it is equivalent to running a sequence of consensus instances. Parties may broadcast a transaction tx by invoking *a-broadcast*(tx), and the protocol outputs transactions through *a-deliver*(tx) events. We recall the definition below.

Definition 2.7 (Atomic Broadcast). A protocol for atomic broadcast satisfies the following properties:

Validity: If a correct party *a-broadcasts* a transaction tx, then every correct party eventually *a-delivers* tx.

No duplication: No transaction is *a-delivered* more than once.

Agreement: If a transaction tx is *a-delivered* by some correct party, then tx is eventually *a-delivered* by every correct party.

Total order: Let tx and tx' be two transactions such that p_i and p_j are correct parties that *a-deliver* tx and tx'. If p_i *a-delivers* tx before tx', then p_j also *a-delivers* tx before tx'.

4.4 Revisiting order fairness

In this section, we discuss the challenges of defining order fairness and highlight limitations of order fairness notions from previous works. We then introduce our refined notion of differential order-fair atomic broadcast.

4.4.1 Limitations

Defining a fair order for atomic broadcast in asynchronous networks is not straightforward since the parties might locally receive transactions for broadcasting in different orders. We assume here that a correct party receives a transaction to be broadcast (e.g., from a client) at the same time when it *a-broadcasts* it. If a party broadcasts a transaction tx before a transaction tx', according to its local order, we denote this by $tx \prec tx'$. Furthermore, we abandon the *validity* property above in the context of atomic broadcast with order fairness and assume now that every transaction is a-broadcast by all correct parties. This corresponds to the implicit assumption made for deploying order-fair broadcast.

Even if all parties are correct, it can be impossible to define a fair order among all transactions. This is shown by a result from social science, known as the *Condorcet paradox*, which states that there exist situations that lead to non-transitive collective voting preferences even if the individual preferences are transitive. Kelkar *et al.* [50] apply this to atomic broadcast and show that delivering transactions in a fair order is not always possible. Their example considers three correct parties p_1,

p_2, and p_3 that receive three transactions tx_a, tx_b, and tx_c. While p_1 receives these transactions in the order $tx_a \prec tx_b \prec tx_c$, party p_2 receives them as $tx_b \prec tx_c \prec tx_a$ and p_3 in the order $tx_c \prec tx_a \prec tx_b$. Obviously, a majority of the parties received tx_a before tx_b, tx_b before tx_c, but also tx_c before tx_a, leading to a cyclic order. Consequently, a fair order cannot be specified even with only correct parties.

One way to handle situations with such cycles in the order is presented by Kelkar *et al.* [50] with *block-order fairness*: their protocol delivers a "block" of transactions at once. Typically, a block will contain those transactions that are involved in a cyclic order. Their notion requires that if sufficiently many parties receive a transaction tx before another transaction tx', then no correct party delivers tx after tx', but they may both appear in the same block. Even though the order among the transactions within a block remains unspecified, the notion of block-order fairness respects a fair order up to this limit.

Kelkar *et al.* [50] specify "sufficiently many" as a γ-fraction of *all* parties, where γ represents an order-fairness parameter such that $\frac{1}{2} < \gamma \leq 1$. More precisely, block-order fairness considers a number of parties η that all receive (and broadcast) two transactions tx and tx'. Block-order fairness for atomic broadcast requires that whenever there are at least $\gamma\eta$ parties that receive tx before tx', then no correct party delivers tx after tx' (but they may deliver tx and tx' in the same block).

Kelkar *et al.* [50] explicitly count faulty parties for their definition. Notice that this immediately leads to problems: If $\gamma\eta < 2f$, for instance, the notion relies on a majority of faulty parties, but no guarantees are possible in this case. Therefore, we only count on events occurring at correct parties here and define a block-order fairness parameter $\overline{\gamma}$ to denote the fraction of *correct* parties that receive one transaction before the other.

Moreover, we assume w.l.o.g. that all correct parties eventually broadcast every transaction, even if this is initially input by a single party only. This simplifies the treatment compared to original block-order fairness, which considers only parties that broadcast *both* transactions, tx and tx' [50]. Our simplification means that a correct party that has received only one transaction will receive the other transaction as well later. This party should eventually include also the second transaction for establishing a fair order. It corresponds to how atomic broadcast is used in practice; hence, we set $\eta = n - f$. In asynchronous networks, furthermore, one has to respect f additional correct parties that may be delayed. Their absence reduces the strength of the formal notion of

block-order fairness in asynchronous networks even more.

In the following, we discuss the range of achievable values for $\overline{\gamma}$. Since we focus on models that allow asynchrony, we assume $n > 3f$ throughout this work. Fundamental results on validity notions for Byzantine consensus in asynchronous networks have been obtained by Fitzi and Garay [38]. Recall that a consensus protocol satisfies *termination*, *integrity*, and *agreement* according to Definition 2.6. *Standard consensus* additionally satisfies:

Validity: If all correct parties propose v, then all correct parties decide v.

Notice that this leaves the decision value completely open if only one correct party proposes something different. In their notion of *strong consensus*, however, the values proposed by correct parties must be better respected, under more circumstances:

Strong validity: If a correct party decides v, then some correct party has proposed v.

Unfortunately, strong consensus is not suitable for practical purposes because Fitzi and Garay [38, Thm. 8] also show that if the proposal values are taken from a domain $\mathcal{V}$, then the resilience depends on $|\mathcal{V}|$. In particular, strong consensus is only possible if $n > |\mathcal{V}|f$.

Related to this, they also introduce *δ-differential consensus*, which respects how many times a value is proposed by the correct parties. This notion ensures, in short, that the decision value has been proposed by "sufficiently many" correct parties compared to how many parties proposed some different value. More precisely, for an execution of consensus and any value $v \in \mathcal{V}$, let $c(v)$ denote the number of correct parties that propose v:

δ-differential validity: If a correct party decides v, then every other value w proposed by some correct party satisfies $c(w) \leq c(v) + \delta$.

To summarize, whereas the standard notion of Byzantine consensus requires that *all* correct parties start with the same value in order to decide on one of the correct parties' input, strong consensus achieves this in any case. It requires that the decision value has been proposed by *some* correct party. However, it does not connect the decision value to how many correct parties have proposed it. Consequently, strong consensus may decide a value proposed by just one correct party. Differential consensus, finally, makes the initial plurality of the decision

value explicit. For $\delta = 0$, in particular, the decision value must be one of the proposed values that is most common among the correct parties. More importantly, differential validity can be achieved under the usual assumption that $n > 3f$.

We now give another characterization of δ-differential validity. For a particular execution of some (asynchronous) Byzantine consensus protocol, let v^* be (one of) the value(s) proposed most often by correct parties, i.e.,

$$v^* = \arg\max_v c(v).$$

Lemma 4.1. *A Byzantine consensus protocol satisfies δ-differential validity if and only if in every one of its executions, it never decides a value w with $c(w) < c(v^*) - \delta$.*

Proof. Assume first that the protocol satisfies δ-differential validity and a correct party decides any value v in the domain. Then every other value w proposed by a correct party satisfies $c(w) \leq c(v) + \delta$. In particular, this implies $c(v^*) \leq c(v)+\delta$, which is equivalent to, $c(v) \geq c(v^*)-\delta$. Hence, the protocol *never* decides a value x with $c(x) < c(v^*) - \delta$.

To show the reverse direction, suppose x is such that $c(x) < c(v^*)-\delta$ and a correct party decides x. This does not satisfy δ-differential validity because also v^* has been proposed by a correct party but $c(v^*) > c(x)+\delta$. □

For consensus with a *binary* domain $\mathcal{V} = \{0, 1\}$, this means that a consensus protocol satisfies δ-differential validity if and only if in every one of its executions with, say, $c(0) > c(1) + \delta$, every correct party decides 0.

No asynchronous consensus algorithm for agreeing on the value proposed by a simple majority of correct parties exists, however. Fitzi and Garay [38, Thm. 11] prove that δ-differential consensus in asynchronous networks is *not possible* for $\delta < 2f$:

Theorem 4.2 ([38]). *In an asynchronous network, δ-differential consensus is achievable only if $\delta \geq 2f$.*

The above discussion already hints at issues with achieving fair order in asynchronous systems. Recall that Kelkar *et al.* [50] present atomic broadcast protocols with block-order fairness for the asynchronous setting with order-fairness parameter γ (whose definition includes faulty

parties). The corruption bound is stated as

$$n > \frac{4f}{2\gamma - 1}. \tag{4.1}$$

For $\gamma = 1$, which ensures fairness only in the most clear cases, there are $n > 4f$ parties required. For values of γ close to $\frac{1}{2}$, the condition becomes prohibitive for practical solutions.

In fact, even when using our interpretation, $\overline{\gamma}$ cannot be too close to $\frac{1}{2}$, as the following result shows. It rules out the existence of $\overline{\gamma}$-block-order-fair atomic broadcast in asynchronous or eventually synchronous networks for $\overline{\gamma} < \frac{1}{2} + \frac{f}{n-f}$.

Theorem 4.3. *In an asynchronous network with n parties and f faults, implementing atomic broadcast with $\overline{\gamma}$-fair block order is not possible unless $\overline{\gamma} \geq \frac{1}{2} + \frac{f}{n-f}$.*

Proof. Towards a contradiction, suppose there is an atomic broadcast protocol ensuring $\overline{\gamma}$-fair block order with $\frac{1}{2} < \overline{\gamma} < \frac{1}{2} + \frac{f}{n-f}$. We will transform this into a differential consensus protocol that violates Theorem 4.2.

The consensus protocol works like this. All parties initialize the atomic broadcast protocol. Upon *propose*(v) with some value v, a party simply *a-broadcasts* v. When the first value v' is *a-delivered* by atomic broadcast to a party, the party executes *decide*(v') and terminates.

Consider any execution of this protocol such that all correct parties propose one of two values, tx or tx'. Suppose w.l.o.g. that $c(tx) = \overline{\gamma}(n-f)$ and $c(tx') = (1-\overline{\gamma})(n-f)$, i.e., tx is proposed $c(tx)$ times by correct parties and more often than tx', since $\overline{\gamma} > \frac{1}{2}$. It follows that $\overline{\gamma}(n-f)$ correct parties *a-broadcast* tx before tx' and $(1-\overline{\gamma})(n-f)$ correct parties *a-broadcast* tx' before tx.

According to the properties of atomic broadcast all correct parties *a-deliver* the same value first in every execution. Moreover, the atomic broadcast protocol *a-delivers* tx before tx' by the $\overline{\gamma}$-fair block order property. This implies that the consensus protocol decides tx in every execution and never tx'. Since no further restrictions are placed on tx and on tx', this consensus protocol actually ensures δ-differential validity for some $\delta < c(tx) - c(tx')$ by Lemma 4.1.

However, the $c(tx)$ and $c(tx')$ satisfy, respectively,

$$\begin{array}{rcccccl} c(tx) & = & \overline{\gamma}(n-f) & < & \left(\frac{1}{2} + \frac{f}{n-f}\right)(n-f) & = & \frac{n+f}{2} \\ c(tx') & = & (1-\overline{\gamma})(n-f) & > & \left(1 - \frac{1}{2} - \frac{f}{n-f}\right)(n-f) & = & \frac{n-3f}{2} \end{array}$$

and, therefore, $\delta < c(tx) - c(tx') < \frac{n+f}{2} - \frac{n-3f}{2} = 2f$. But δ-differential asynchronous consensus is only possible when $\delta \geq 2f$, a contradiction. □

4.4.2 Differential order-fairness

The limitations discussed above have an influence on order fairness. The condition on δ to achieve *δ-differential consensus* directly impacts any measure of fairness. It becomes clear that a *relative* notion for block-order fairness, defined through a fraction like $\overline{\gamma}$, may not be expressive enough.

We now start to define our notion of *order-fair atomic broadcast*; it has almost the same interface as regular atomic broadcast. The primitive is accessed with *of-broadcast*(tx) for broadcasting a transaction tx and it outputs transactions through *of-deliver*(T) events, where T is a *set* of transactions delivered at the same time; T corresponds the block of block-order fairness. We want to count the number of correct parties that *of-broadcast* a transaction tx before another transaction tx' and introduce a function

$$b : \mathcal{T} \times \mathcal{T} \to \mathbb{N}$$

for all tx and tx' that were ever *of-broadcast* by correct parties. The value $b(tx, tx')$ denotes the *number of correct parties* that *of-broadcast* tx before tx' in an execution. As above we assume w.l.o.g. that a correct party will *of-broadcast* tx and tx' eventually and that, therefore, $b(tx, tx') + b(tx', tx) = n - f$.

Can we achieve that if $b(tx, tx') > b(tx', tx)$, i.e., when there are more correct parties that *of-broadcast* transaction tx before tx' than correct parties that *of-broadcast* tx' before tx, then no correct party will *of-deliver* tx' before tx? Using a reduction from δ-differential consensus, as in the previous result, we can show that this condition is too weak.

Theorem 4.4. *Consider an atomic broadcast protocol that satisfies the following notion of order fairness for some $\mu \geq 0$:*

Weak differential order fairness: *For any tx and tx', if $b(tx, tx') > b(tx', tx) + \mu$, then no correct party* a-delivers *tx' before tx.*

Then it must hold $\mu \geq 2f$.

Proof. Towards a contradiction, suppose there is an atomic broadcast protocol, which ensures that for all transactions tx and tx' with $b(tx, tx') >$

$b(tx', tx) + \mu$ and $\mu \geq 0$, no correct party *a-delivers* tx' before tx and that $\mu < 2f$. We will transform this into a differential consensus protocol that violates Theorem 4.2.

The consensus protocol works like this. All parties initialize the order-fair atomic broadcast protocol. Upon *propose*(v) with some value v, a party simply *of-broadcasts* v. When the first value v' is *of-delivered* to a party, the party executes *decide*(v') and terminates.

Consider any execution of this protocol such that all correct parties propose one of two values, tx or tx'. Suppose w.l.o.g. that tx is proposed $c(tx)$ times by correct parties and more often than tx', which is proposed $c(tx')$ times, with $c(tx) + c(tx') = n - f$ and that $c(tx) > c(tx') + \mu$. It follows that $b(tx, tx') = c(tx)$ correct parties *of-broadcast* tx before tx' and $b(tx', tx) = c(tx')$ correct parties *of-broadcast* tx' before tx, hence, $b(tx, tx') > b(tx', tx) + \mu$.

According to the properties of atomic broadcast, all parties *of-deliver* the same value first in every execution. Moreover, the protocol *of-delivers* tx before tx' because $b(tx, tx') > b(tx', tx) + \mu$. This implies that the consensus protocol decides tx in every such execution. Since no further restrictions are placed on tx and tx' and since $c(tx) - c(tx') > \mu$, this consensus protocol actually implements μ-differential consensus by Lemma 4.1. However, achieving μ-differential asynchronous consensus requires that $\mu \geq 2f$ according to Theorem 4.2. But $\mu < 2f$ by the above assumption. This is a contradiction. □

On the basis of this result, we now formulate our notion of κ-*differentially order-fair atomic broadcast*, using a fairness parameter $\kappa \geq 0$ to express the strength of the fairness. Smaller values of κ ensure stronger fairness in the sense of how large the majority of parties that *of-broadcast* some tx before tx' must be to ensure that tx will be *of-delivered* before tx' and in a fair order.

Recall that throughout this work, we assume that if one correct party *of-broadcasts* some transaction tx, then every correct party eventually also *of-broadcasts* tx. For reasons that are discussed later (in the remarks after the protocol description in Section 4.5.2), we use a weaker formal notion of validity, which considers executions with only correct parties.

Definition 4.5 (κ-Differentially Order-Fair Atomic Broadcast). A protocol for κ-*differentially order-fair atomic broadcast* satisfies the properties *no duplication, agreement* and *total order* of *atomic broadcast* and additionally:

Weak validity: If all parties are correct and *of-broadcast* a finite number of transactions, then every correct party eventually *of-delivers* all of these *of-broadcast* transactions.

κ-differential order fairness: If $b(tx, tx') > b(tx', tx) + 2f + \kappa$, then no correct party *of-delivers* tx' before tx.

Compared to the above notion of weak differential order fairness, we have $\kappa = \mu - 2f$. We show in the next section how to implement κ-differentially order-fair atomic broadcast.

4.5 Quick order-fair atomic broadcast protocol

This section presents first an overview of our *quick order-fair atomic broadcast* algorithm in Section 4.5.1. A detailed description of the implementation follows in Section 4.5.2, along with the pseudocode in Algorithm 5–6. Finally, the complexity of the algorithm is discussed in Section 4.5.3.

4.5.1 Overview

The protocol concurrently runs a Byzantine FIFO consistent broadcast channel (BCCH) and proceeds in rounds of consensus. BCCH allows parties to deliver multiple transactions consistently. An incoming *of-broadcast* event with a transaction tx triggers BCCH and *bcch-broadcasts* tx to the network. Additionally, every party keeps a local vector clock that counts the transactions that have been *bcch-delivered* from each sending party. Every party also maintains an array of lists *msgs* such that *msgs*$[i]$ records all *bcch-delivered* transactions from p_i.

When a party *bcch-delivers* the transaction tx, it increments the corresponding vector-clock entry and appends tx to the appropriate list in *msgs*. As soon as sufficiently many new transactions are found in *msgs*, a new round starts. Each party signs its vector clock and sends it to all others. The received vector clocks are collected in a matrix, and once $n - f$ valid vector clocks are recorded, a new validated Byzantine consensus (VBC) instance is triggered. The party proposes the matrix and the signatures for consensus, and VBC decides on a common matrix with valid signatures. This matrix defines a *cut*, which is a vector of indices, with one index per party, such that the index for p_j determines an

entry in $msgs[j]$ up to which transactions are considered for creating the fair order in the round. It may be that the index points to transactions that a party p_i does not store in $msgs[j]$ because they have not been *bcch-delivered* yet. When the party detects such a missing transaction, it asks all other parties to send the missing transaction directly and in a verifiable way, such that every party will store all transactions up to the cut in *msgs*.

Once all parties received the transactions up to the cut, the algorithm starts to build a graph that represents the dependencies among transactions that must be respected for a fair order. This graph resembles the one used in Aequitas [50], but its semantics and implementation differ. The vertices in the graph here are all *new* transactions defined by the cut and an edge (tx, tx') indicates that tx should at most be *of-delivered* before tx'.

The graph results from two steps. In the first step, the party creates a vertex for every transaction that appears in a distinct lists in *msgs* and it is not yet *of-delivered*. In the second step, the algorithm builds a matrix M such that $M[tx][tx']$ counts how many times tx appears before tx' in *msgs* (up to the cut). $M[tx][tx']$ can be interpreted as *votes*, counting how many parties want to order tx before tx'. Notice that entries of M exist only for tx and tx' where at least one of $M[tx][tx']$ and $M[tx'][tx]$ is non-zero.

If the *difference* between entries $M[tx][tx']$ and $M[tx'][tx]$ is large enough, then the protocol adds a directed edge (tx, tx') to the graph. The edge indicates that tx' must not be *of-delivered* before tx. More precisely, assuming that transactions tx and tx' have been observed by at least $n - f$ parties, such an edge is added for all tx and tx' with $M[tx][tx'] > M[tx'][tx] - f + \kappa$. The condition is explained through the following result.

Lemma 4.6. *If* $b(tx, tx') > b(tx', tx)+2f+\kappa$, *then* $M[tx][tx'] > M[tx'][tx]-f+\kappa$.

Proof. At least $M[tx][tx'] - f$ correct parties have *of-broadcast* tx before tx' because $M[tx][tx']$ may include reports about tx and tx' in *msgs* from up to f incorrect parties. In other words,

$$b(tx, tx') \geq M[tx][tx'] - f \quad \iff \quad M[tx][tx'] \leq b(tx, tx') + f$$

At most $M[tx][tx'] + 2f$ correct parties have *of-broadcast* tx before tx' because $M[tx][tx']$ may include reports about tx or tx' in *msgs* from up

to f incorrect parties, and there may be up to $2f$ correct parties whose arrays were not considered in this number. That is,

$$b(tx, tx') \leq M[tx][tx'] + 2f \quad \Longleftrightarrow \quad M[tx][tx'] \geq b(tx, tx') - 2f$$

Suppose $b(tx, tx') > b(tx', tx) + 2f + \kappa$. The above implies

$$\begin{aligned} M[tx][tx'] &\geq b(tx, tx') - 2f \\ &> b(tx', tx) + 2f + \kappa - 2f \\ &= b(tx', tx) + \kappa \\ &\geq M[tx'][tx] - f + \kappa. \end{aligned}$$

Thus, whenever $M[tx][tx'] > M[tx'][tx] - f + \kappa$, we need to prevent that the protocol *of-delivers* tx' before tx. We do this by adding an edge from tx to tx' to the graph; as shown later, this ensures that tx' is not *of-delivered* before tx. □

In the discussion so far, we have assumed that the two transactions tx and tx' were received by at least $n - f$ parties. Observe that every party can only contribute with 1 to either $M[tx][tx']$ or to $M[tx'][tx]$, but not to both. However, it may occur that only a few parties receive tx and tx' before the cut, which implies that $M[tx][tx']$ may be very small, for example. But that count might actually grow later and take on values up to $n - f - M[tx'][tx]$. For this reason, we extend the condition derived from Lemma 4.6 in the algorithm as follows: if $n-f-M[tx'][tx] > M[tx'][tx]-f+\kappa$ (which implies that $M[tx'][tx]$ is small, i.e., $M[tx'][tx] < \frac{n-\kappa}{2}$), we also add add an edge between tx and tx'. In summary, then, the algorithm adds an edge from tx to tx' whenever

$$\max\left\{M[tx][tx'], n - f - M[tx'][tx]\right\} > M[tx'][tx] - f + \kappa.$$

Creating the graph in this manner leads to a directed graph that represents constraints to be respected by a fair order. Notice that two transactions may be connected by edges in both directions when the difference is small and $\kappa < f$, i.e., there may be a cycle (tx, tx') and (tx', tx). This means that the difference between the number of parties voting for one or the other order is too small to decide on a fair order. Longer cycles may also exist. All transactions with circular dependencies among them will be *of-delivered* together as a set. For deriving this information, the algorithm repeatedly detects all strongly connected components in the graph and collapses them to a vertex. In other words, any two vertices

tx and *tx*′ are merged when there exists a path from *tx* to *tx*′ and a path from *tx*′ to *tx*. This technique also handles cases like those derived from the *Condorcet paradox*.

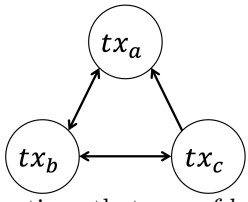
Finally, with the help of the collapsed graph, all transactions defined by the cut are *of-delivered* in a fair order: First, all vertices without any incoming edges are selected. Secondly, these vertices are sorted in a deterministic way and the corresponding transactions are *of-delivered* one after the other. Then the partyed vertices are removed from the graph and another iteration through the graph starts. As soon as there are no vertices left, i.e., all transactions are *of-delivered*, the protocol proceeds to the next round.

Note that cycles may also extend beyond the cut, as shown by Kelkar *et al.* [49]. Therefore, the algorithm holds back transactions and does not *of-deliver* them while they may still become part of a longer cycle. This is ensured by counting how many times a transaction appears in *msgs* up to the cut. In particular, let $C[tx]$ count this number for a transaction *tx*. We require that any transaction is only *of-delivered* when $C[tx] \geq \frac{n+f-\kappa}{2}$, i.e., after *tx* appears in *msgs* often enough such that it cannot become part of a cycle later or already be in a cycle that will grow later, e.g., through transactions that arrive after the cut.

Example 4.7. Let us consider a system of $n = 4$ parties, of which three (p_1, p_2, and p_3) are correct and one (p_4) is faulty ($f = 1$). We fix the order-fairness parameter $\kappa = 0$, the notion is trivially satisfied for higher values of κ. Every correct party *of-broadcasts* three transactions tx_a, tx_b, and tx_c, in an order that forms a Condorcet cycle. The Byzantine party p_4 does not *of-broadcast*. Suppose all transactions have been *bcch-delivered* in round r to all correct parties, as shown in Figure 4.1. Then the protocol obtains the cut $c = \begin{bmatrix}3 & 2 & 1 & 0\end{bmatrix}$.

From Algorithm 5-6 (L219), the matrix M and the corresponding graph (L222) are

$$M = \begin{bmatrix}0 & 0 & 0\\ 1 & 0 & 1\\ 2 & 0 & 0\end{bmatrix}$$



Notice that arbitrarily many transactions that are *of-broadcast* immediately after tx_a by p_1–p_3 might follow and arrive only in a future round, after cut *c*. The protocol cannot know this yet and must there-

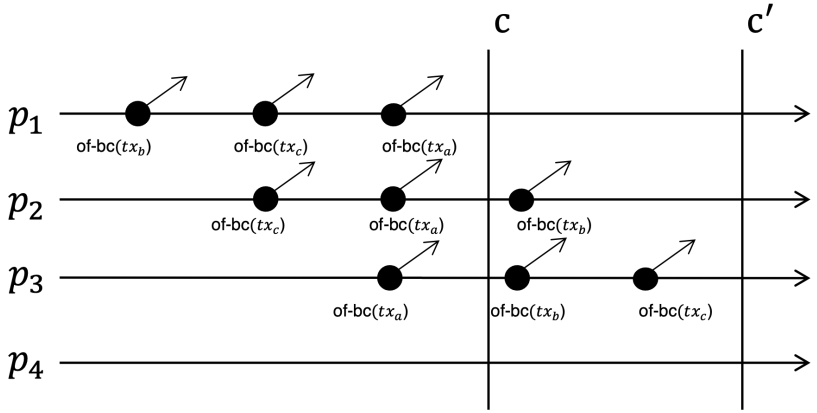
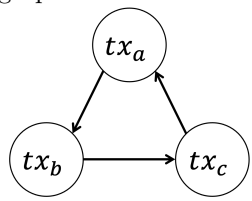


Figure 4.1: The execution of Example 4.7, in which three correct parties p_1, p_2, p_3 *of-broadcast* transactions that form a cycle, which makes it impossible to sort them in a fair order. After tx_a, and after the protocol has computed cut c, an unbounded number of additional transactions might follow (see text).

fore postpone *of-delivery* of tx_a. As captured by the condition that $C[tx_b] = 1 < \frac{n+f-\kappa}{2}$, no transaction is *of-delivered* in this round. The protocol continues with another round r' obtaining a cut c', cf. Figure 4.1. Then the matrix M and the graph become

$$M = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{bmatrix}$$



At this point, the protocol *of-delivers* $\{tx_a, tx_b, tx_c\}$ together, from a collapsed vertex, because now $C[tx_i] = 3 \geq \frac{n+f-\kappa}{2}$ for $i \in \{a, b, c\}$.

4.5.2 Implementation

Algorithm 5–6 shows the *quick order-fair atomic broadcast protocol* for a party p_i. The protocol proceeds in rounds, maintains a round counter r (L173), and uses a boolean variable *inround*, which indicates whether the consensus phase of a round is executing (L174).

Every party maintains two hash maps: *msgs* (L175) and *vc* (L176), in which party identifiers serve as keys. Hash map *msgs* contains ordered lists of *bcch-delivered* transaction from each party in the system. Variable *vc* is a vector clock counting how many transactions were *bcch-delivered* from each party.

Rounds. In each round, a matrix L (L177) and a list Σ (L178) are constructed as inputs for consensus. The matrix L will consist of vector clocks from the parties and Σ will contain the signatures of the parties. Additionally, every party maintains a list of integers called *cut* (L179) that are calculated in every round. This cut represents an index for every list in *msgs* to determine the transaction to be used for creating the fair order. Initially, all values are zero. Finally, all *of-delivered* transactions are included in a set *delivered* (L180), to prevent a repeated delivery in future rounds.

The protocol starts when a client submits a transaction *tx* using an *of-broadcast*(*tx*) event. BCCH then broadcasts *tx* to all parties in the network (L183). When *tx* with label l from party p_j is *bcch-delivered* (L184), the vector clock *vc* for party p_j is incremented. The attached label l is not used by the algorithm and only serves to define that all correct parties *bcch-deliver* the same transaction following Definition 2.5. Additionally, transaction *tx* is appended to the list *msgs*[j] using an operation *append*(*tx*) (L186).

When the length of p_j's list in *msgs* exceeds the *cut* value for p_j, new transactions may have arrived that should be ordered (L187). This tells the protocol to initiate a new round. A new round could also be triggered later, as described in the remarks at the end of this section.

The first step of round r is to set the flag *inround*. Secondly, the protocol digitally signs the vector clock *vc* and obtains a signature σ. The values r, σ, and *vc* are then sent in a STATUS message to all parties (L188–L190). When party p_i receives a STATUS message from p_j, it validates the contained signature σ' using *verify*(j, vc', σ') (L192). An additional security check is made by comparing the locally stored round number r with the round number r' from the message. If both conditions hold, the vector clock vc' is stored as row j in matrix L (L193) and σ' is stored in list Σ at index j (L194).

Defining a cut. As soon as p_i has received $n-f$ valid STATUS-message (L195), it invokes consensus (VBC, L196) for the round through *vbc-*

propose with proposal (L, Σ). The predicate of VBC checks that a proposal consists of a matrix L and a vector Σ such that for at least $n - f$ values j, the entry $\Sigma[j]$ is a valid signature on row j of L. When the VBC protocol subsequently decides, it outputs a common matrix L' of vector clocks and a list Σ' of signatures (L198). The party then uses L' to calculate the cut, where $cut[j]$ is the largest value s such that at least $f + 1$ elements in column j in L' are bigger or equal than s (L200). In other words, $cut[j]$ represents how many transactions from p_j were *bcch-delivered* by enough parties. This value is used as index into $msgs[j]$ to determine the transactions that will be considered for creating the order in this round.

The algorithm then makes sure that all parties will hold at least all those transactions in *msgs* that are defined by *cut*. Each party detects missing transactions from sender p_j from any difference between $vc[j]$ and $cut[j]$ (L202); if there are any, the party broadcasts a MISSING-message to all others. When another party receives such a request from p_j and already has the requested transactions in *msgs*, it extracts them into a variable *resend* (L206). More precisely, it extracts a proof from the BCCH primitive with which any other party can verify that the transaction from this particular sender is genuine. This is done by invoking *bcch-create-proof*(*resend*) (L207); the transactions and the proof are then sent in a RESEND-message to the requesting party p_j (L208).

When party p_i receives a RESEND-message with a missing transaction from p_k, it first verifies the provided proof s' from the message by invoking a *bcch-verify-proof*(s') function (L211). If the proof is valid, p_i extracts (L213) the transactions through *bcch-get-transactions*(s'), appends them to $msgs[k]$, and increments $vc[k]$ accordingly. The party repeats this until *msgs* contains all transactions included in the cut.

Ordering transactions. At this point, every party stores all transactions *msgs* that have been *bcch-delivered* up to the cut. The remaining operations of the round are deterministic and executed by all parties independently. The next step is to construct the directed *dependency graph* G that expresses the constraints on the fair order of the transactions. Vertices (V) in G represent transactions that may be *of-delivered* and edges (E) in G express constraints on the order among these transactions. First, all transactions within the cut that are not yet delivered are added as vertices to the set V (L 215).

Then, for each pair of transactions tx and tx' in V, the algorithm con-

Algorithm 5 Quick order-fair atomic broadcast (code for p_i).

State

173: $r \leftarrow 1$: current round
174: $inround \leftarrow$ FALSE
175: $msgs \leftarrow [\,] : \text{HashMap}\big[\{1, ..., n\} \rightarrow [\,]\big]$: array of *bcch-delivered* txs
176: $vc \leftarrow [\,] : \text{HashMap}\big[\{1, ..., n\} \rightarrow \mathbb{N}\big]$: vector clocks for *bcch*-delivered txs
177: $L \leftarrow [0]^{n \times n}$: matrix of logical timestamps, constructed from n v. clock
178: $\Sigma \leftarrow [\,]^n$: list of signatures from STATUS messages
179: $cut \leftarrow [0]^n$: the cut decided for the round
180: $delivered \leftarrow \emptyset$: set of delivered transactions

Initialization

181: Byzantine FIFO consistent broadcast channel (*bcch*)

182: **upon** *of-broadcast*(tx) **do**
183: *bcch-broadcast*(tx)

184: **upon** *bcch-deliver*(p_j, l, tx) **do**
185: $vc[j] \leftarrow vc[j] + 1$
186: $msgs[j].append(tx)$

// perhaps waiting longer
187: **upon** exists j **such that** $len(msgs[j]) > cut[j] \wedge \neg inround$ **do**
188: $inround \leftarrow$ TRUE
189: $\sigma \leftarrow sign(i, vc)$
190: send message $[\text{STATUS}, r, vc, \sigma]$ to all $p_j \in \mathcal{P}$

191: **upon** receiving message $[\text{STATUS}, r', vc', \sigma']$ from p_j
192: **such that** $r' = r \wedge verify(j, vc', \sigma')$ **do**
193: $L[j] \leftarrow vc'$
194: $\Sigma[j] \leftarrow \sigma'$

195: **upon** $\big|\{p_j \in \mathcal{P} \mid \Sigma[j] \neq \perp\}\big| \geq n - f$ **do**
196: *vbc-propose*$\big((L, \Sigma)\big)$ for validated Byzantine consensus in round r
197: $\Sigma \leftarrow [\,]^n$

198: **upon** *vbc-decide*$\big((L', \Sigma')\big)$ in round r **do** // calculate the cut
199: **for** $j \in \{1, \ldots, n\}$ **do** // for each row in L'
200: $cut[j] \leftarrow max\big\{s \mid \{k \mid \big|\{L'[k][j] \geq s\}\big| > f\}\big\}$
201: **for** $j \in \{1, \ldots, n\}$ **do** // check for missing transactions
202: **if** $vc[j] < cut[j]$ **then**
203: send message $[\text{MISSING}, r, j, vc[j]]$ to all $p_k \in \mathcal{P}$

Algorithm 6 Quick order-fair atomic broadcast (code for p_i).

204:**upon** receiving message $[\text{MISSING}, r', k, index]$ from p_j **such that** $r' = r$ **do**
205: **if** $vc[k] \geq cut[k]$ **then**
206: $resend \leftarrow msgs[k][index \dots cut[k]]$ // copy transactions from p_k
207: $s \leftarrow$ *bcch-create-proof*$(resend)$
208: send message $[\text{RESEND}, r, k, s]$ to p_j // send missing transactions

209:**upon** receiving message $[\text{RESEND}, r', k', s']$ from p_j
210: **such that** $r' = r \wedge len(msgs[k]) < cut[k]$ **do**
211: **if** *bcch-verify-proof*(s') **then**
212: $vc[k] \leftarrow vc[k] +$ *bcch-get-length*(s')
213: $msgs[k]$.*append*(*bcch-get-transactions*(s'))

214:**upon** $len(msgs[j]) \geq cut[j]$ for all $j \in \{1, \dots, n\}$ **do**
215: $V \leftarrow \left(\bigcup_{j \in \{1,\dots,n\}} msgs[j]\,[1 \dots cut[k]]\right) \setminus delivered$
216: $M \leftarrow [\,] : \text{HashMap}[\mathcal{T} \times \mathcal{T} \to \mathbb{N}]$
217: $C \leftarrow [\,] : \text{HashMap}[\mathcal{T} \to \mathbb{N}]$
218: **for** $tx, tx' \in V$ **do**
219: $M[tx][tx'] \leftarrow \Big|\{j \in \{1, \dots, n\} \mid$
220: tx appears before tx' in $msgs[j]\,[1 \dots cut[k]]\}\Big|$
221: $C[tx] \leftarrow \Big|\{p_j \mid tx \in msgs[j]\,[1 \dots cut[k]]\}\Big|$
222: $E \leftarrow \Big\{(tx, tx') \;\Big|\; \max\{M[tx][tx'], n - f - M[tx'][tx]\} > M[tx'][tx] - f + \kappa\Big\}$
223: $H \leftarrow (V, E)$ // $(V, E) = G$
224: **while** H contains some strongly connected subgraph
225: $\overline{H} = (\overline{W}, \overline{F}) \subseteq H$ **do**
226: $H \leftarrow H/\overline{F}$ // collapse vertices
227: **while** $\exists w \in sort(W) : indegree(w) = 0 \wedge stable(w)$ **do**
228: *of-deliver*$(flatten(w))$
229: $delivered \leftarrow delivered \cup flatten(w)$
230: $W \leftarrow W \setminus \{w\}$
231: $L \leftarrow [0]^{n \times n}$
232: $inround \leftarrow$ FALSE
233: $r \leftarrow r + 1$ // move to the next round

234:**function** $stable(w)$
235: **return** $\left(w \in \mathcal{T} \wedge C[w] \geq \frac{n+f-\kappa}{2}\right) \vee \bigwedge_{w' \in w : w' \notin \mathcal{T}} stable(w')$

236:**function** $flatten(w)$
237: **return** $\{tx \in w \mid tx \in \mathcal{T}\} \cup \bigcup_{w' \in w : w' \notin \mathcal{T}} flatten(w')$

structs M (L219) such that $M[tx][tx']$ counts how many times a transaction tx appears before transaction tx' in the cut. In the same loop, the algorithm counts how many times transaction tx appears within the cut and stores this result in array C (L221). Finally, all entries $M[tx][tx']$ and $M[tx'][tx]$ are compared and if condition $max\{M[tx][tx'], n - f - M[tx'][tx]\} > M[tx'][tx] - f + \kappa$ holds, then a directed edge from tx to tx' is added (L222). This edge indicates that tx must *not* be ordered *after* tx', i.e., that tx is *of-delivered* before tx' or together with tx'.

Any transactions that cannot be ordered with respect to each other now correspond to strongly connected components of G. A strongly connected component is a subgraph, which for each pair of vertices tx and tx' contains a path from tx to tx' and one from tx' to tx. In the next step, a graph $H = (W, F)$ is created and all strongly connected components in H are repeatedly collapsed until H contains no more cycles. This is done by contracting the edges in each connected component and merging all its vertices (L223–L226).

The algorithm further considers all vertices w without incoming edges and which satisfy condition $C[tx] \geq \frac{n+f-\kappa}{2}$, checked in function *stable*(w) (L 234). All such w will be sorted in a deterministic way (L 227). Notice that w may correspond to a transaction from $\mathcal{T}$ or a recursive set of sets of transactions. Therefore function *flatten*(w) (L 236) is used to extract transactions and *of-deliver* them (L 228). All *of-delivered* transactions are added to *delivered* (L229 to prevent a repeated partying. Finally, w is removed from H (L230), and a next pass of extracting vertices with no incoming edge follows. This is repeated until all vertices have been partyed and *of-delivered*.

The algorithm then initializes L, sets *inround* to FALSE, increments the round number r, and starts the next round (L231-L233).

Remarks. The condition for starting a round in L187 only waits until *one* single transaction exists in *msgs* that was not considered before. This is necessary for liveness but not very efficient. This number can be increased such that a new round starts only after $K = \Theta(n)$ new transactions have arrived. Note that this threshold affects the amortized transaction and bit complexities that are considered in Section 4.5.3.

Recall that our model assumes that every correct party *of-broadcasts* all transactions. For simplicity, though, our validity property has been formulated only for executions without faulty parties. It could be strengthened so that it holds for all executions, in which the parties do not *of-*

broadcast an unbounded number of them that form a Condorcet cycle.

The protocol can also be changed to satisfy the even stronger liveness property of Kelkar *et al.* [49], which the Themis protocol satisfies. To deal with Condorcet cycles of unbounded length, one would modify the interface of order-fair broadcast so that it additionally outputs *of-startblock* and *of-endblock* events that carry no parameters. Furthermore, *of-deliver* would only output single transactions from $\mathcal{T}$. An output "block" then consists of all transactions that are *of-delivered* between a *of-startblock* event and the subsequent *of-endblock* event. However, long cycles occur very infrequently in realistic scenarios, as shown by Kelkar *et al.* [49].

If consensus is not "black box" and treated in a modular way, more efficient variations of this protocol become possible. In particular, the ordering rounds may be integrated with a leader-based Byzantine consensus protocol [18]. This implies that multiple leaders in successive consensus rounds (or "epochs") may be needed to agree on the cut of one ordering round. The Themis protocol [49] adopts this pattern.

The protocol satisfies another natural property, which has not been made explicit before in the literature, but is achieved by several existing protocols [49], [50], [91], not only by quick order-fair broadcast. Consider an execution in which the correct parties *of-broadcast* transactions $tx_1, \ldots, tx_l$ such that $b(tx_i, tx_j) > 2f + \kappa$ for $i = 1, \ldots, l$ and $j = i + 1, \ldots, l$ and there are no further transactions *of-broadcast* that might include $tx_1, \ldots tx_l$ in a cycle: Then tx_i is actually *of-delivered* before tx_j. Note that differential order fairness is a safety condition and would not prevent that $tx_1, \ldots, tx_l$ are *of-delivered* jointly in one set.

4.5.3 Complexity

In this section, we analyze the complexity of the *quick order-fair atomic broadcast protocol*. We use two measures: message complexity and communication (bit) complexity. Moreover, we compare our results with existing algorithms from the literature.

Message complexity. If the Byzantine FIFO consistent broadcast channel (BCCH) is implemented using "echo broadcast" [79], it takes $O(n)$ protocol messages per transaction. Since more than f parties *of-broadcast* each transaction and f is proportional to n, the overall message complexity of BCCH is $O(n^2)$. Under high load, batching could be used to reduce the number of messages incurred by BCCH. In the

protocol itself, every party sends $O(n)$ STATUS, MISSING, and RESEND messages, which also amounts to $O(n^2)$ messages.

The cost of validated Byzantine consensus (VBC) depends on the assumptions used for implementing it. In the asynchronous model, optimal protocols [1], [64] achieve $O(n^2)$ messages on average. Assuming that K new transactions are delivered in each round, this becomes $O(\frac{n^2}{K})$ per transaction. Choosing $K = \Omega(n)$ reduces the amortized cost of consensus to $O(n)$ messages per transaction. Note that when using an implementation of VBC with complexity $O(n^3)$, as the algorithm of Cachin *et al.* [19], we can choose K proportional to n and may again obtain expected amortized message complexity $O(n^2)$.

With a partially synchronous consensus protocol according to Section 4.3.3, VBC uses $O(n)$ messages in the best case and $O(n^2)$ messages in the worst case. The total amortized cost of quick order-fair atomic broadcast per transaction, therefore, is also $O(n^2)$ messages in this implementation.

Communication (bit) complexity. If digital signatures are of length λ and transactions are at most L bits, the bit complexity of BCCH for one sender is $O(nL+n^2\lambda)$, and since we assume that $O(n)$ parties broadcast each message, this becomes $O(n^2L + n^3\lambda)$. Optimal asynchronous VBC protocols [1], [64] have $O(nL+n^2\lambda)$ expected communication cost, for their transaction length L. Since the proposals for VBC are $n \times n$ matrices, the bit complexity of this phase is $O(n^3 + n^2\lambda)$. Assuming that K is proportional to n, the amortized bit complexity of VBC per transaction is $O(n^2 + n\lambda)$. From this, it follows that the amortized bit complexity of the algorithm per transaction is $O(n^2L + n^3\lambda)$.

Discussion. Table 4.1 gives an overview of message complexities of algorithms with different notions for fair transaction ordering. We compare our *quick order-fair atomic broadcast* with the algorithms introduced by Kelkar *et al.* [50] and Zhang *et al.* [91]. We leave out from the overview the protocol by Kursawe [54] since it has a completely different approach for solving fair transaction ordering.

The asynchronous Aequitas protocol [50, Sec. 7] provides fair order using a *FIFO Broadcast primitive*, implemented by *OARcast* of Ho *et al.* [45]. The implementation of OARcast described there uses n *ARcasts* [45] for each transaction, where one ARcast causes $\Theta(n^2)$ network messages. Since Aequitas requires that every correct party broadcasts

each transaction, the total complexity increases by another factor of n. Thus, each transaction incurs a cost of $\Theta(n^4)$ messages in the gossip phase. Moreover, one instance of *set agreement* is executed for each transaction, and each one of them calls n binary consensus protocols. Therefore Aequitas uses $\Omega(n^4)$ messages for delivering one transaction, which exceeds the cost of quick order-fair broadcast at least by the factor n^2.

Ordering linearizability [91] is defined using a logical order of events observed on each party. Its implementation in the Pompē protocol, however, appears to require synchronized clocks in the sense of knowing bounds on differences between local clocks. Hence, the complexity of Pompē cannot be compared to that of asynchronous protocols for order fairness. Irrespective of this difference, its cost is $O(n^2)$ messages and one instance of Byzantine consensus per transaction. The communication complexity of this protocol is $O(n^3L)$ since each party broadcasts a SEQUENCE-message to all others with contents of length $O(nL)$.

Themis [49] relies strongly on a leader p_ℓ to construct a fair order. If p_ℓ does not perform its task timely, the protocol may switch to another leader, similarly to existing leader-based protocols. For assessing the complexity of Themis here, we consider the optimistic case, but note that the complexities stated for some other protocols, in particular for the quick order-fair broadcast, do not depend on timely leaders.

Themis lets all parties send their local orderings to p_ℓ first. Suppose these consist of approximately $K = \Theta(n)$ transactions each. Then p_ℓ constructs a graph G on these and sends G and some justification information to all parties. They maintain local graphs, update them in response, and potentially output some transactions. This incurs a cost of $O(n)$ messages. Since G contains K nodes and, in general, $O(K^2)$ edges, the average communication complexity is $O(n^2 + nL)$ in the best case.

4.6 Analysis

In this section, we show that the *quick order-fair atomic broadcast protocol* in Algorithm 5–6 implements κ-differentially order-fair atomic broadcast. The properties to be satisfied are (Definition 4.5): *no duplication, agreement, total order, strong validity and κ-differential order fairness.*

Lemma 4.8. *No transaction is of-delivered more than once in Algorithm 5–6.*

Notion	Algorithm	Avg. messages	Avg. communication
Block-Order-Fairness [50]	Async. Aequitas [50]	$O(n^4)$	$O(n^4 L)$
Ordering Linearizability [91]	Pompē* [91]	$O(n^2)$	$O(n^3 L)$
Block-Order-Fairness [49]	Themis [49]	$O(n)$	$O(n^2 + nL)$
Differential Order Fairness	Quick o.-f. broadcast	$O(n^2)$	$O(n^2 L + n^3 \lambda)$

Table 4.1: Overview of different notions for fair transaction ordering and corresponding algorithms, with their expected message and communication complexities. The summary assumes $L \geq \lambda$. (* The Pompē protocol requires synchronized clocks.)

Proof. The check in L215 of the protocol implementation ensures that no transaction is *of-delivered* more than once. In the step when the protocol creates graph vertices, transactions that are already contained in variable *delivered* are filtered out. Those transactions will not be included in the graph and cannot be *of-delivered* again. Note that even in the case when a transaction transaction *tx* is *bcch-delivered* multiple times, because of filtering in L215, it is not possible that *tx* is *of-delivered* more than once. □

Lemma 4.9. *In Algorithm 5–6, if a transaction tx is of-delivered by some correct party, then tx is eventually of-delivered by every correct party.*

Proof. Suppose that a transaction *tx* is *of-delivered* by some correct party p_i in round r. Following the protocol steps, in round r all correct parties decide on the same L' (L198). This is guaranteed by the *agreement* property of the validated Byzantine consensus because no two correct parties decide differently. Since the matrix L' is used to construct the cut deterministically, all correct parties construct the same *cut* (L200).

We can then distinguish two cases: In the first case, all correct parties have already *bcch-delivered* *tx* and store it in *msgs*. In the second case, there are some correct parties that have never heard of *tx* simply because of some delays in the network. Then these correct parties send a MISSING-message to all parties, requesting the delivery of their missing transactions. Since every transaction included in the cut was announced by $f + 1$ parties, and therefore also by at least one correct party, some party will respond with a RESEND-message containing *tx*. Once all these transactions are delivered, all correct parties store *tx* in *msgs*.

In the next step, every correct party builds graph G. Each vertex in the graph is constructed deterministically from the same information by every party, concretely, from the transactions in *msgs* and excluding those that are already in the *delivered* set (L215). Following the protocol, every correct party will eventually construct the same G and output the same sequence of transactions, also including *tx*. □

Lemma 4.10. *Let tx and tx′ be two transactions such that p_i and p_j are correct parties that of-deliver tx and tx′. In Algorithm 5–6, if p_i of-delivers tx before tx′, then p_j also of-delivers tx before tx′.*

Proof. Consider two distinct transactions *tx* and *tx′* and let p_i and p_j be any two correct parties that *of-deliver* both transactions. Assume that p_i *of-delivers* *tx* before *tx′*. If p_i *of-delivers* *tx* and *tx′* in round r, then both transactions were included in the cut for p_i. Due to the argument used to establish the *agreement* property in Lemma 4.9, it must be that *tx* and *tx′* were also included in the cut for party p_j in round r. The rest of the protocol, i.e., building a graph and *of-delivering* transactions is deterministic. Therefore, p_j delivers these two transactions in round r and also *of-delivers* *tx* before *tx′*. Extending this argument over all rounds of the protocol, it follows that every correct party *of-delivers* the same sequence of transactions. □

Lemma 4.11. *If all parties are correct and of-broadcast a finite number of transactions in Algorithm 5–6, then every correct party eventually of-delivers these transactions.*

Proof. Let p_i be some correct party that *of-broadcasts* a transaction transaction *tx*. Due to the *validity* property of the underlying Byzantine FIFO consistent broadcast channel, every correct party eventually *bcch-delivers* *tx*. According to the algorithm, in every round r a party p_i waits for $n - f$ parties to receive signed vector clocks to proposes a matrix of logical timestamps L for validated Byzantine consensus (L196).

The *termination* property of validated Byzantine consensus guarantees that every correct party eventually decides some value and according to the *agreement* property, no two correct parties decide differently. The resulting common L' allows then each party to determine if tx is considered in the current round r. A transaction tx is considered if at least $f+1$ parties have *bcch-delivered* tx and reported it in their vector clock (L200). Additionally, if tx is considered in the current round but some party p_i has not *bcch-delivered* tx yet, p_i will request that other parties send it the missing transaction (L203). Further, all transactions in *msgs* are added as vertices to the graph G (L215). Moreover, because every party *of-broadcasts* a finite number of transactions, every possible graph that is created will be finite. Since tx was *of-broadcast* by a correct party p_i, all parties are correct and *of-broadcast* a finite number of transactions, tx will eventually be *of-delivered*.

□

Lemma 4.12. *In Algorithm 5–6, if* $b(tx, tx') > b(tx', tx) + 2f + \kappa$*, then no correct party of-delivers* tx' *before* tx*.*

Proof. Recall that $b(tx, tx')$ is the number of correct parties that receive and *of-broadcast* tx before tx'. Consider any two transactions tx and tx' such that $b(tx, tx') > b(tx', tx) + 2f + \kappa$.

Suppose tx and tx' are both included in the cut of some round and none of them has been *of-delivered* yet. The protocol defines a threshold based on M for creating an edge between two vertices. Lemma 4.6 shows that the condition for differential order fairness ensures that $M[tx][tx'] > M[tx'][tx] - f + \kappa$ in the protocol, where $M[tx][tx']$ counts how many times tx appears before tx' in *msgs*. Moreover, as explained in connection with Lemma 4.6, the algorithm extends this condition for adding an edge (m, m') to

$$\max\left\{M[tx][tx'], n - f - M[tx'][tx]\right\} > M[tx'][tx] - f + \kappa, \tag{4.2}$$

in order to cope with particularly small values of $M[tx'][tx]$. This may be the case when the full relative ordering information about tx and tx', in the sense that $M[tx][tx'] + M[tx'][tx] \geq n - f$, is not yet available with the cut. The implementation then adds an edge from tx to tx' to the graph (L222). This implies that tx' will not be *of-delivered* before tx because the algorithm respects this order by traversing the graph starting with vertices that have no incoming edges. Therefore, tx is either *of-delivered* before tx' or both transactions are delivered together,

within the same set. Moreover, observe that (4.2) ensures that graph generated by the protocol is connected.

Consider now the case that tx' is not included at all in the cut of the current round r. We want to show that for all $m \in V$ of the graph G, if tx is *of-delivered* in round r, there cannot be such an tx', for which an edge (tx', tx) would be added at a later round and which might therefore violate κ-order fairness. Recall that an edge (tx', tx) is added to G in a round whenever (4.2) holds.

To be more precise, we show that the condition in L227 and the properties of *stable*() ensure κ-differential order fairness for such tx and tx'. Let $\bar{w}$ be a node of the graph as in L227. If every tx in $\mathit{flatten}(\bar{w})$ appears at least $\frac{n+f-\kappa}{2}$ times in *msgs* up to the cut, i.e., satisfies $\mathit{stable}(tx)$, it means that no tx' (not in the cut) can be ordered before the transactions in G of subsequent rounds. In fact, let $m \in \mathit{flatten}(\bar{w})$ be such that $\mathit{stable}(tx) = \text{TRUE}$ and tx' not be in the cut. Since $\mathit{stable}(tx)$ holds, $C[tx] \geq \frac{n+f-\kappa}{2}$ also means that $M[tx][tx'] \geq \frac{n+f-\kappa}{2}$. Thus,

$$M[tx'][tx] \leq n - M[tx][tx'] \leq n - \frac{n+f-\kappa}{2} = \frac{n-f+\kappa}{2}$$

in any future round as well. But this implies $M[tx'][tx] - M[tx][tx'] \leq -f + \kappa$, and thus, no edge (tx', tx) is added according to (4.2). The argument given earlier then shows that order fairness is maintained. Notice that this takes care of scenarios as in Example 4.7 that include some transaction $\bar{tx} \in \mathit{flatten}(\bar{w})$ with $\neg\mathit{stable}(\bar{tx})$. There may exist a further transaction tx' not included in the cut such that tx' must be ordered not after $\bar{tx}$. □

Lemmas 4.8–4.12 directly imply the following theorem, which concludes the analysis of the protocol.

Theorem 4.13. *Algorithm 5–6 implements κ-differentially order-fair atomic broadcast.*

4.7 Conclusion

The quick order-fair atomic broadcast protocol guarantees transaction delivery in a differentially fair order. It works both for asynchronous and eventually synchronous networks with optimal resilience, tolerating corruptions of up to one third of the parties. Compared to existing order-fair atomic broadcast protocols, our protocol is considerably more

efficient and incurs only quadratic cost in terms of amortized message complexity per delivered transaction.

Chapter 5

Quick Order Fairness: Implementation and Evaluation

5.1 Introduction

Decentralized finance (DeFi) describes a financial system built on blockchain technology that aims to recreate and enhance traditional financial services without relying on centralized authorities, such as banks or brokers. In the DeFi ecosystem, users can engage in various financial activities, including lending, borrowing, trading, and earning interest. DeFi mechanisms are implemented by smart contracts running on a decentralized blockchain network. Many parties jointly power the network through a consensus protocol robust against attacks by malicious actors.

As described in Section 4.1 DeFi, is not (yet) immune against certain kinds of fraud such as *front-running*. Such attacks exploit the decentralized and transparent nature of the consensus and transaction execution in a blockchain, highlighting the need for a protocol that imposes fairness in DeFi. Moreover, front-running may occur also on non-programmable blockchains like the XRP Ledger [87].

Research has proposed several solutions to prevent such attacks. One is to enforce a *causal order* using distributed cryptography. A second approach called *receive-order fairness*, considers in which order transactions were received by the parties running the consensus protocol and

enforces corresponding constraints on the order resulting from consensus. A third approach called *randomized order* removes the influence on transaction order by the parties running consensus. Finally, the last approach, called *architectural separation*, splits the task of ordering transactions and delegates it to a separate service. Chapter 6 provides a detailed overview of these approaches.

Quick Order-Fair Atomic Broadcast (QOF) [21] is a representative of the group of receive-order fairness protocols. As described in Chapter 4, it adds a new property called *differential order fairness* to the existing properties of atomic broadcast. The protocol works for asynchronous and eventually synchronous networks with optimal resilience and tolerates faults of up to one-third of the total number of parties. The resilience to faults does not depend on the fairness notion. Compared to similar solutions, QOF is more efficient. It requires, on average, $O(n^2)$ messages to deliver one transaction. For comparison, the asynchronous *Aequitas* protocol [50, Sec. 7] needs $O(n^4)$ messages and has resilience $n > 4f$ or worse. Protocol *Themis* [49] achieves the same resilience and also takes $O(n^2)$ messages to deliver one transaction, though a SNARK-based variant reduces this further in the best case. However, the number of faulty parties tolerated by Aequitas and Themis depends on the quality of the fairness that is achieved.

This chapter elaborates on the order-fair atomic broadcast and the QOF protocol. Our primary motivation is to describe an implementation of the QOF protocol in detail and to measure its cost. Implementing a prototype is essential for empirically validating the theoretical base of the proposed solution. More precisely, we describe a modular implementation of the QOF protocol on top of an existing library for atomic (i.e., total-order) broadcast called *bamboo* [39], which realizes the HotStuff [90] consensus protocol (note that *consensus* and *atomic broadcast* are synonyms here). It uses three components: Byzantine consistent broadcast, validated Byzantine consensus, and a graph module. The first module provides consistency for transactions sent by potentially faulty parties [21]. The validated Byzantine consensus module is built directly on bamboo and supports *external validity*. The graph module maintains a directed acyclic graph and provides functions for capturing potential dependencies among the transaction. Connecting all modules and implementing the logic of extracting the fair order of transactions from the graph structure completes the implementation of the QOF protocol.

The implementation allows us to evaluate the performance of the

QOF protocol and assess its efficiency. We measure throughput and latency and compare it to the baseline HotStuff implementation in bamboo [39]. Our experiments indicate that (with four servers) compared to the HotStuff protocol, QOF reduces the throughput by at most 5% and increases latency by about 50ms, reflecting the impact of the QOF protocol's increased complexity in an ideal, emulated network. We also draw a connection between our results and the performance of similar protocols for imposing a fair order, including Themis [49] and Pompē [91].

Furthermore, we outline how the Quick Order-Fair protocol may be deployed in practice, offering two integration approaches. The first approach involves implementing it as a separate service, where clients submit transactions to ordering nodes. Ordering nodes use the QOF protocol to determine a fair order, which validators execute through a smart contract. The second approach directly integrates QOF into validators, with clients submitting transactions to validators running the algorithm and executing transactions on the ledger.

To summarize, this work presents three contributions:

- It describes a practical implementation of the QOF protocol, illustrating many aspects left out in earlier work and providing a coherent representation of the protocol and its components.
- It examines the protocol's integration into real-world systems, explaining possible designs and providing a smart contract blueprint.
- It conducts an empirical evaluation in several dimensions: scalability, throughput, and latency. This evaluation affirms the efficacy of the QOF protocol and provides valuable guidance for its practical deployment in decentralized systems.

The chapter starts with a review of the existing techniques for preventing MEV (Section 5.2). Then, we describe the Quick Order Fairness protocol (Section 5.3). Section 5.4 describes the prototype's building blocks and implementation. Section 5.5 describes how the protocol can be integrated into practical systems. Section 5.6 presents the evaluation results, and finally, Section 5.7 concludes the chapter.

5.2 Related work

Recall from Section 4.2 that several lines of research have been developed in the past years to address the front-running problem at the consensus

level of blockchain networks, both in academia and in practice. A detailed treatment is included in Chapter 6. On a high level, we can group the proposed defense methods into four categories. Methods of the first kind are explained in Section 6.2, aim to prevent side information leaks to malicious insiders by using distributed cryptography, which enforces a *causal order* on the transaction sequence produced by consensus. The second kind of defense, known as *receive-order fairness* (Section 6.3), considers how the transactions were received by the individual parties running the consensus protocol and enforces corresponding constraints on the order resulting from consensus. The third category *randomized order* (Section 6.4) removes the influence on transaction order by the parties running consensus. The last category, *architectural separation* (Section 6.5), splits the task of ordering transactions and delegates it to a separate service.

Heimbach and Wattenhofer [44] give an overview of state-of-the-art techniques for preventing manipulation of the transaction order. They present a taxonomy of the techniques and compare them regarding decentralization, security, scope, and other properties. Another survey of knowledge given by Baum *et al.* [10] describes common front-running attacks and assesses three mitigation categories. Moreover, they introduce a sandwich attack on input batching techniques that can be mitigated with private user balances and secret input stores. Both works conclude that despite the growing number of approaches to address transaction reordering manipulations on blockchains, an effective technique to mitigate the front-running problem still needs to be discovered. The current state of research indicates that no existing approach can fully meet the requirements posed by a decentralized blockchain environment.

5.3 Quick Order Fairness protocol

This section reviews the Quick Order Fairness protocol (QOF) (Chapter 4). The protocol functions effectively in both asynchronous and eventually synchronous networks, demonstrating resilience by tolerating corruptions in up to a third of the parties. It is accessed with *of-broadast(tx)* for broadcasting a transaction tx and outputs transactions through *of-deliver(T)*, where T is a set of transactions delivered at the same time. If one correct party *of-broadcasts* some transaction tx, then every correct party eventually also *of-broadcasts* tx. We recall the definition of κ-*differentially order-fair atomic broadcast* below.

Definition 4.5 (κ-Differentially Order-Fair Atomic Broadcast). A protocol for *κ-differentially order-fair atomic broadcast* satisfies the properties *no duplication, agreement* and *total order* of *atomic broadcast* and additionally:

Weak validity: If all parties are correct and *of-broadcast* a finite number of transactions, then every correct party eventually *of-delivers* all of these *of-broadcast* transactions.

κ-differential order fairness: If $b(tx, tx') > b(tx', tx) + 2f + \kappa$, then no correct party *of-delivers* tx' before tx.

The value $b(tx, tx')$ denotes the *number of correct parties* that *of-broadcast* tx before tx' in an execution. Parameter f denotes the number of faulty parties, and fairness parameter $\kappa \geq 0$ expresses the strength of the fairness. Smaller values of κ ensure stronger fairness in the sense of how large the majority of parties that *of-broadcast* some tx before tx' must be to ensure that tx will be *of-delivered* before tx' and in a fair order.

The protocol consists of three budling blocks: FIFO consistent broadcast channel, validated Byzantine consensus, and graph module. Since the FIFO consistent broadcast channel and validated Byzantine consensus will be described in Section 5.4, we will focus on the graph-building phase in this section. Concretely, we will describe how the protocol builds a graph and determines the order of transactions using a toy example.

5.3.1 Broadcast and consensus

The quick order fairness protocol proceeds in rounds and concurrently uses a FIFO consistent broadcast channel (bcch) to deliver transactions [21]. Figure 5.1 depicts an example of the first phase of the protocol. The system consists of a three correct parties, p_i, p_j, p_k, where parameter κ, respectively, round r is zero. To keep the example simple, we do not include Byzantine parties.

The protocol starts once the client submits a transaction tx_1 (1). Over time, the client will submit three more transactions: tx_2, tx_3, and tx_4 (not necessary in this order). The submitted transaction is *bcch-broadcasted* (2) to other parties. Every party keeps a local vector clock vc that counts the transactions that have been *bcch-delivered* from each sending party. Every party also maintains an array of lists *msgs* such

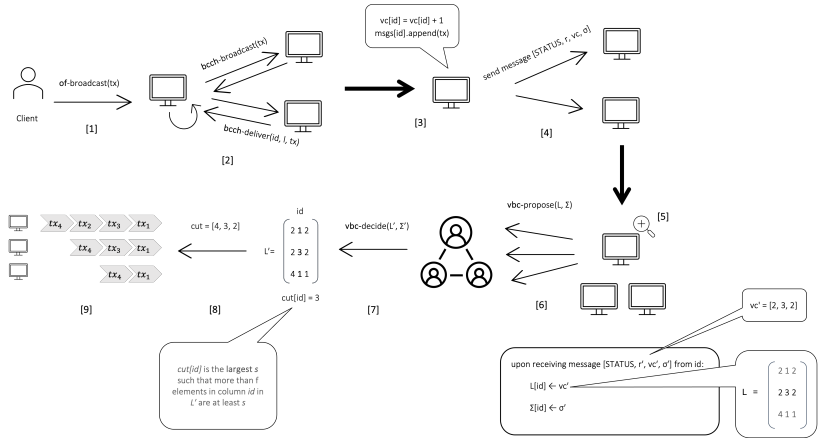


Figure 5.1: Execution of the protocol shows the first (consensus) phase of the protocol.

that $msgs[i]$ records all *bcch-delivered* transactions from p_i. Upon *bcch-delivering tx*, every party increments vc counter and appends the transaction to local array $msgs$ (3). The new round starts when sufficiently many new transactions are found in $msgs$. In the next step, each party signs its vc and sends it to all other parties as STATUS message (4). All received vector clocks and signatures are stored in matrices L and Σ (5). A row of matrix L maps to received vc from the party id. Once $n - f$ STATUS messages are collected, a party proposes L and Σ to the consensus module (6).

The protocol then runs a validated Byzantine consensus (vbc) protocol to agree on a matrix L' and a list Σ' of signatures that validate L' (7). In this example, the protocol decides on matrix L' and uses it to determine the cut (8). Each matrix column calculates the largest value such that more than f elements in the column are at least that value. The cut is then defined as the vector of these values corresponding to each party. The cut is $[4, 3, 2]$ in this example. The cut determines an entry in $msgs$ array up to which transactions are considered for creating the fair order in the round (9). A party may be missing some transactions in the cut. In that case, the protocol will ask other parties to send the missing transactions. Once the message exchange is completed, the protocol proceeds to the next phase.

5.3.2 Building a graph

The next phase is building a graph. Figure 5.2 shows steps of building a directed dependency graph representing a fair transaction order. Previous execution steps produced the cut [4, 3, 2]. This means that in the current round (10), party p_i observes $tx_4 \prec tx_2 \prec tx_3 \prec tx_1$, party p_j observes $tx_4 \prec tx_3 \prec tx_1$ and p_k observes $tx_4 \prec tx_1$. The first step is to create vertices of the graph by selecting all unique transactions within the cut that have not yet been delivered (11). In this example, this will produce four vertices. The next step is constructing matrix M (12), i.e., calculating for every party how many times transaction tx is delivered before tx', within the cut. The next step adds edges (13) to the graph by checking the following condition:

$$max\{M[tx][tx'], n - f - M[tx'][tx]\} > M[tx'][tx] - f + \kappa \tag{5.1}$$

where $M[tx][tx']$ is the number how many times is tx delivered before tx', f is the number of faulty parties, n is the total number of parties and κ is fairness parameter. Note that this condition check is done for each pair of vertices so that edges might be added in both directions. To avoid the problem of Condorcet cycles, the next step tries to collapse the graph (14). In this example, there is a path from every vertex to another, so the whole graph is collapsed into a single vertex. Starting from the vertex with zero incoming edges, the protocol extracts transactions from the vertex and checks *stable* condition, i.e., if every transaction appears more or equal to $\frac{n+f-\kappa}{2}$ times within the cut. In this example, transaction tx_2 appears only once in the cut but should appear at least twice to be stable (15). Therefore, the protocol cannot deliver anything and moves to the next round (16).

Figure 5.3 shows a continuation of the execution shown in Figure 5.2. Meanwhile, more transactions arrived at parties (17), extending the cut to [4, 4, 4]. As in the previous figure, we add vertices (18), calculate again matrix M (19), and add edges (20) to the graph following the same steps. The difference from the last round is that this time, we see fewer edges in the graph because the protocol has more information about the order, making it more confident about transaction ordering. After the collapsing stage (21), we create two vertices (tx_4) and (tx_1, tx_2, tx_3). The protocol starts from vertex (tx_4) (with zero incoming edges) and checks *stable* condition (22). That is more than enough since the transaction tx_4 appears four times in the cut. It is *of-delivered* first. Then, we remove vertex tx_4 from the graph (23), and the protocol chooses the next vertex

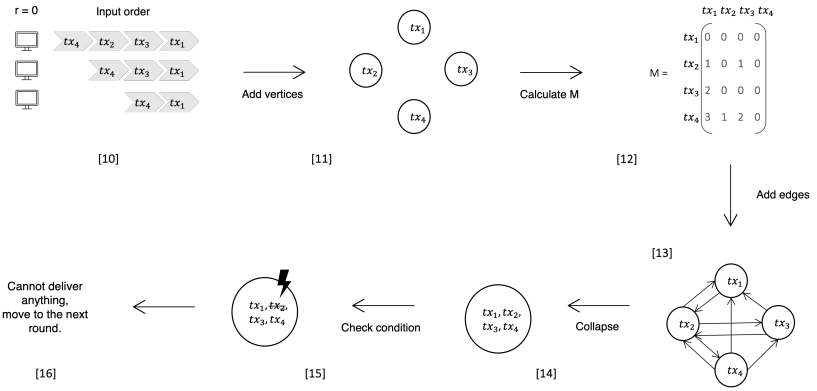


Figure 5.2: Example execution of building a graph starting from the first round.

(tx_1, tx_2, tx_3). Again, because the next vertex has multiple transactions inside, the protocol checks if each satisfies *stable* condition. We deliver all three transactions together this time since they fulfill the condition (24). We then remove the corresponding vertex, leaving nothing else to deliver.

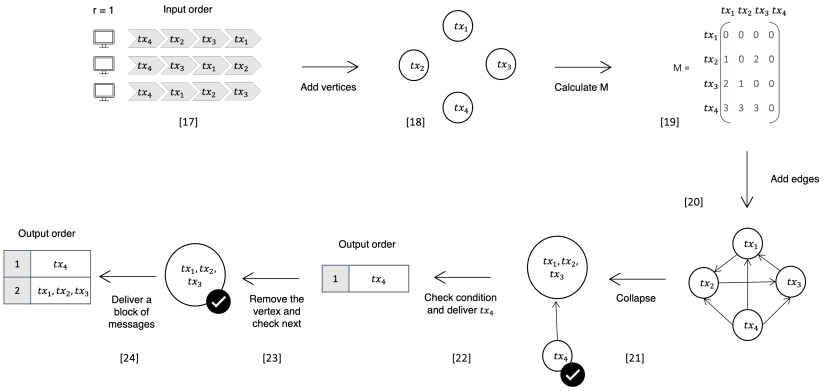


Figure 5.3: Continuation of the execution from Figure 5.2.

5.4 Implementation

Implementing the quick order-fair protocol builds on top of three modules: a Byzantine consistent broadcast module, a validated Byzantine consensus module, and a graph module. In the following, we describe the implementation of these modules. The code is written in Go version 1.15.7.

5.4.1 Byzantine consistent broadcast channel

We modularly implement the Byzantine consistent broadcast channel (bcch) abstraction. For every sender, bcch invokes a sequence of broadcast primitives (bcb) so that only one is active at any moment. BCB relies on authenticated perfect links (al) that communicate with Transmission Control Protocol (TCP).

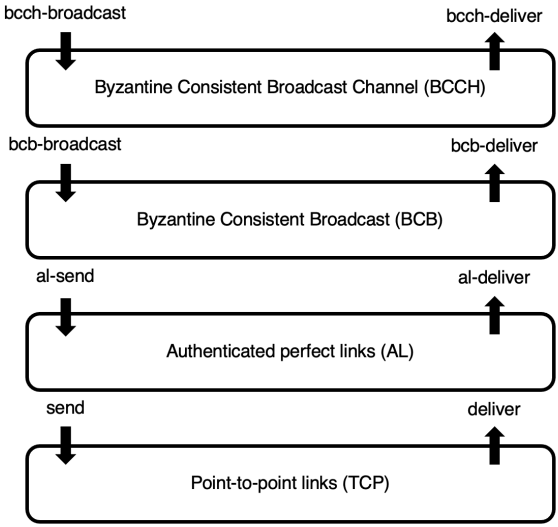


Figure 5.4: A stack of modules for implementing Byzantine consistent broadcast channel in Quick Order Fairness protocol.

Implementation details

The Byzantine consistent broadcast channel implementation follows Algorithm 3.19 of Cachin *et al.* [18]. Initially, each party creates an instance of bcch, automatically creating an instance of Byzantine consistent broadcast (*bcb*). Then protocol waits for *of-broadcast(tx)* event to happen. Every time this event is triggered, *bcch-broadcasts tx* using underlying *bcb-broadcast(tx)* primitive. This primitive implements Signed Echo Broadcast presented in Algorithm 3.17 [18]. This protocol uses an authenticated perfect links abstraction and a cryptographic digital signature scheme. Authenticated perfect links (*al*) primitive implements Authenticate and Filter shown in Algorithm 2.4 [18]. It uses a Hash-based message authentication[1] (HMAC) over a TCP network communication. Specifically, in our implementation, we implement HMAC_256 in the file *hmac.go*, package *hotstuff-impl/crypto*.

5.4.2 Validated Byzantine consensus

This module implements validated Byzantine consensus (vbc) introduced by Cachin *et al.* [19] using the HotStuff [90] implementation in the project *bamboo*[2]. Observe that HotStuff does not provide validation. We must modify the implementation to cope with the *external validity* property. Moreover, HotStuff is implemented as an atomic broadcast instance: the output for every correct party is a sequence of ordered transactions. To make this into a consensus protocol, we agree on considering the first message output by a correct party proposed by the leader for round r.

Figure 5.5 shows a high-level architecture for implementing validated Byzantine consensus. A client sends a *of-broadcast* transaction to the Quick Order Fairness module. This module communicates with the validated Byzantine consensus module through a *vbc-propose*.

A validated Byzantine consensus protocol is activated by a *vbc-propose* message carrying a value v with a proof π that validates v, i.e., π should satisfy a predicate P for v. A correct party only decides for values validated by a proof π. In our quick order-fair protocol, a correct party proposes for consensus a matrix L of vector clocks (counting how many transactions were *bcch-delivered* from each party) together with a list Σ containing the signatures of the parties on the vector clocks,

[1] HMAC is a cryptographic technique with a hash function and a secret key.
[2] `https://github.com/gitferry/bamboo`

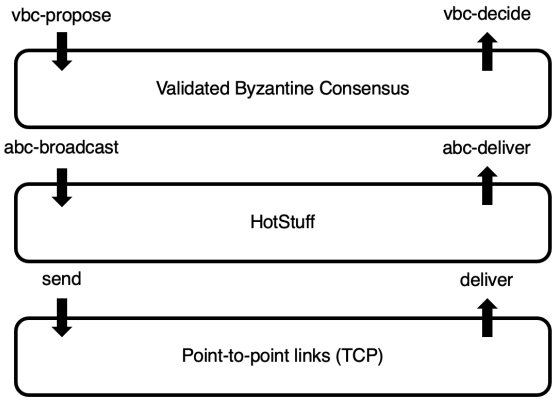


Figure 5.5: A stack of modules for implementing validated Byzantine consensus in Quick Order Fairness protocol.

which represents the proof π for the validation. We define predicate P as true whenever Σ correctly verifies L for round r. In particular,

$$P_r(L, \Sigma) = \text{TRUE iff } \forall vc' \in L,\ \forall \sigma' \in \Sigma : \mathit{verify}(j, vc', \sigma').$$

In HotStuff, whenever the leader for round r broadcasts a proposal block containing a list of (possibly different) matrices L or a correct replica delivers the block proposed by the leader, then verify that $P_r(L, \Sigma)$ holds; proceed only if $P_r(L, \Sigma) = \text{TRUE}$ and halt otherwise. HotStuff uses underlying point-to-point links for internal communication. The first matrix in the delivered block is decided and used to determine the *cut* for round r. More precisely, the validation party occurs in the validated Byzantine consensus module, which *vbc-decides* on a matrix L' with a proof π that validates L'.

Implementation details

An abstract vbc module (Def. 2.6) has the following events: *vbc-propose(v)* and *vbc-decide(v)*. A correct party first proposes a value and then must

wait for a decision before it proposes a new value. We recall the definition below.

Definition 2.6 (Validated Byzantine Consensus). A protocol solves validated Byzantine consensus with validity predicate P if it satisfies the following conditions:

Termination: Every correct process eventually decides some value.

Integrity: No correct process decides twice.

Agreement: No two correct processes decide differently.

External validity: Every correct process only decides a value v such that $P(v) = \text{TRUE}$. Moreover, if all processes are correct and propose v, then no correct process decides a value different from v.

Atomic broadcast (Def. 2.7) abstraction is accessible via two events: *abc-broadcast*(v) that broadcasts a value v and *abc-deliver*(v) that delivers a decided value v. We recall the definition below.

Definition 2.7 (Atomic Broadcast). A protocol for atomic broadcast satisfies the following properties:

Validity: If a correct process *a-broadcasts* a message m, then every correct process eventually *a-delivers* m.

No duplication: No message is *a-delivered* more than once.

Agreement: If a message m is *a-delivered* by some correct process, then m is eventually *a-delivered* by every correct process.

Total order: Let m and m' be two messages such that p_i and p_j are correct processes that *a-deliver* m and m'. If p_i *a-delivers* m before m', then p_j also *a-delivers* m before m'.

Algorithm 7 shows an abstract implementation of vbc using atomic broadcast (ABC) running on p_i. A flag *inround* is, by default, set to false and will change only when a consensus round starts. Variables *roundp* and *roundd* count how many times a value is proposed, respectively, and delivered. A new vbc consensus round starts when *vbc-propose*(v) is triggered only when *inround* is false. Then, counter *roundp* increases by one, *inround* is set to true, and the transaction is broadcasted. Transaction structure is given in Table 5.1. Upon party p_i receiving transaction

w (containing value v) from another party, it checks if the round of received transaction is the same as its *roundp* and if *roundp* > *roundd*. If yes, then *roundd* is increased by one, *inround* set to false, and value v is *vbc-decided*. Otherwise, the proposal value v is discarded.

Attribute	Description
vbc	Message tag.
v	Proposed value.
roundp	Proposing round.

Table 5.1: Structure of a proposed transaction.

Algorithm 7 Abstract implementation of vbc using ABC (code for p_i).

```
    State:
238:    inround ← FALSE                          // flag if we can start consensus
239:    roundp ← 0        // counter for how many times proposed (QOF round)
240:    roundd ← 0                          // counter for how many times delivered

241:upon vbc-propose(v) such that not inround do
242:    roundp ← roundp + 1      //in QOF this happens after building the graph
243:    inround ← TRUE
244:    // abc-broadcast proposal of p_i for vbc instance roundp
245:    abc-broadcast([VBC, roundp, v])

246:upon abc-deliver(w) such that w = [VBC, r, v] do
247:    // first abc-delivered proposal for this round
248:    if r = roundp ∧ roundp > roundd then
249:        roundd ← roundd + 1
250:        inround ← FALSE
251:        vbc-decide(v)
252:    else
253:        // discard proposal value v
```

Algorithm 8 is a concrete implementation of vbc in QOF within HotStuff of the bamboo library. As in the previous algorithm, *roundp* and *roundd* keep consensus running correctly. Additionally, *view* keeps track of HotStuff round, and matrix *votes* stores votes for a specific *block.id*, given by *voter*. Upon *vbc-propose* with value v, the algorithm starts a new round of vbc consensus by increasing *roundp* by one and creates transaction t, which is a tuple of VBC tag, *roundp* and v. The created transaction is added to the local mempool of the bamboo library

of a party p_i using the *mempool.addNew*(t) function.

When party p_i becomes leader, it increases *view* by one and creates a block. Block is created from payload generated by the function *mempool.some*(). This function will get up to 20 transactions from the mempool and put them in *payload*. Then, the function *makeBlock* takes *payload* and *view* to build a block. The block is then sent to all parties in the message BLOCK.

When party p_i receives a block from some other party, it uses it to create a vote. First, using cryptographic function *sign*, party p_i signs *block.id* and produces signature σ, which is included in *vote*. The vote structure is given in Table 5.2. Then, the created vote is sent inside VOTE message to the next leader.

Attribute	**Description**
view	Round in which is the vote created.
voter	Party that created the vote.
block.id	Identifier of a voted block.
σ	Signature of the vote.

Table 5.2: Structure of a vote.

When a leader receives the vote, it first verifies it, using the *verify* function, and only then it adds the vote into matrix *votes*. If more than $\frac{2}{3}$ of a total number of parties voted for the same *block.id*, a quorum is reached, and a quorum certificate (*qc*) is generated. At the same time, *view* is updated. The block can be committed if the *qc* view is bigger or equal to three. As we know, HotStuff takes three rounds to commit a block, so the previous condition comes from this fact. Before committing the block, the last check is to check if the view of the grandparent's and parents' blocks is correct. Finally, the block is committed by calling the function *commitBlock* that takes the grandparent block and current view as arguments. This function will append a committed block to the channel called *cBlocks*.

Every time a new block is committed, the algorithm takes the last block from *cBlocks* and extracts the payload in transaction t. If an extracted tag is VBC, the proposal round of t is the same as the current proposal round, and *roundp* > *roundd* then *roundd* is increased by one, and the value stored in t is *vbc-decided*. Otherwise, the value of the proposed transaction, respectively, is discarded.

5.4.3 Graph building

The last phase of the quick order fairness protocol is building a graph that reflects the fair order of transactions. The graph is implemented as a directed acyclic graph (DAG), where every vertex represents a transaction (delivered by bcch), and an edge represents a dependency between two transactions. Package *graph* holds the implementation of the graph structure and utility functions.

Implementation details

Algorithm 9 depicts the structure of the implemented graph. The graph is defined as a map of vertices. The Vertex structure represents each vertex. The relations between vertices represent the graph's edges,i.e., an edge has no explicit structure. Each vertex keeps track of outbound edges to other vertices.

Implemented functions in the graph module are given in Table 5.3. These functions create a graph, add and remove vertices, add edges, collapse a graph, calculate strongly connected components, and implement other helper functions. This work will focus on the implementation of collapsing a graph functionality, given in Algorithm 9, since it is the most complex function in the graph module.

Function	Description
NewDirectedGraph	Creates a new directed graph.
AddVertex	Adds a vertex to a graph.
RemoveVertex	Removes a vertex from a graph.
AddEdge	Adds an edge between two vertices.
CollapseGraph	Collapses a graph into a single vertex.
SCC	Implements Strongly Connected Components.
DFS	Implements Depth First Search.
Transpose	Transposes a graph.
Visit	Loops through the graph using DFS and outputs SCC.
Indegree	Calculates the indegree of a vertex.

Table 5.3: List of implemented graph utility functions.

After a party constructs vertices and edges of the graph G, it will call function *CollapseGraph* that will try to collapse the graph G into

a single vertex (L307). The function *CollapseGraph* first checks if the graph has less or equal to one vertex. If yes, it returns the same graph since there is nothing to collapse. Otherwise, it creates a new graph H and calculates strongly connected components (SCC) of G (L316). Then, it loops through all components and adds them as vertices to the new graph H. Finally, it returns the new graph H.

The function $SCC(G)$ finds strongly connected components, i.e., in a directed graph, it checks if there is a directed path from every vertex in the component to every other vertex in the same component (L316). It loops through all vertices in G and checks if a vertex has been visited. If this is not the case, it calls *DFS* function (L331) that will traverse the graph and mark all visited vertices. Then, it transposes the graph G and loops through all vertices in the stack. For each vertex from the stack, it checks if it has already been visited. If not, it calls *Visit* function that will loop through the graph using DFS and fill *components* list. Finally, it returns the *components*.

Function $DFS(visited, stack, node)$ implements Depth First Search (DFS) algorithm (L331). It checks if a given *node* is already visited. If not, it marks it as visited and loops through all neighbors of the *node*. For each neighbor, it calls the recursively *DFS* function. Finally, it appends the vertex to the stack. We choose to implement DFS since it is a simple algorithm with efficient time complexity.

5.5 Integration

This section discusses how to deploy QOF in a blockchain network. One approach is implementing it as a separate service through which clients submit transactions. Alternatively, the network's validators can integrate it directly with the consensus protocol.

A separate service. In one approach, clients first send transactions to specific *ordering nodes* responsible for imposing order fairness (Figure 5.6). In this scenario, the QOF protocol provides a separate ordering mechanism, like those implemented by layer-two networks. In contrast to many layer-two solutions, the ordering nodes implement proper distributed consensus. The ordering nodes output a sequence of transactions that respects differential order fairness. This information is signed and sent to validators who run the smart contract that executes transactions on the ledger in the given order. With this approach, the ledger

protocol is agnostic to the fair ordering process, and a contract may opt to consume only transactions in a fair order. In this case, the smart contract should verify the digital signatures of the ordering nodes before executing transactions. Some pseudocode for a smart contract of this kind is shown in Algorithm 10.

This approach can readily be integrated with many layer-two networks that have recently become popular [55]. These networks increase scalability and efficiency by moving transaction execution off the main blockchain. Running off-chain, they enable faster and more cost-effective transactions while retaining the security benefits of the underlying blockchain by pushing only the effects of these transactions onto the main network. Today, layer-two systems typically use a centralized sequencer that gathers client transactions and organizes them. Subsequently, transactions are executed through the roll-up mechanism, and the resulting updated state is recorded on layer one, the main blockchain, through a roll-up contract deployed there. As most employ just one sequencer, this poses a single point of failure and restricts interoperability with other, distinct layer-two systems. It is readily possible to distribute the function of the sequencer across multiple nodes, even to serve multiple roll-up protocols from the same distributed layer-two sequencer. Such an approach was recently introduced by *Espresso* [35].

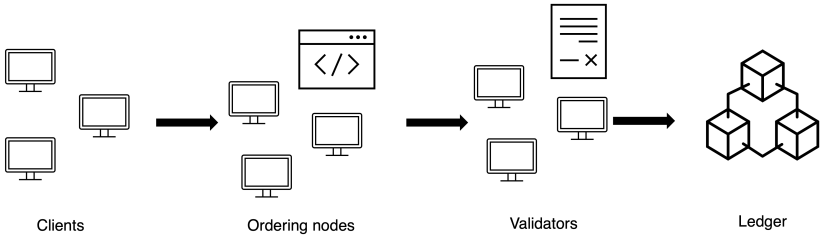


Figure 5.6: Clients submit transactions to ordering nodes that use the QOF algorithm as a separate service. The algorithm outputs transactions in fair order and sends them to the validators who run a smart contract. Smart contract executes transactions on the ledger in a fair order.

Integration with consensus. The second approach is to directly build the QOF protocol into the consensus protocol run by the val-

idators, as shown in Figure 5.7. In this case, clients submit transactions to the validators that extend their consensus protocol by the QOF algorithm and output transactions in fair order. Then, validators run a smart contract to execute transactions on the ledger. This approach implements the fairness notion natively and for all smart contracts and realizes the original vision of protecting the system against front-running. A notable disadvantage is the requirement to change the original protocol since the QOF algorithm needs to be integrated. Furthermore, the very notion of differential order-fairness relies on a known set of validators. Hence, such an integration is not feasible for truly permissionless consensus protocols.

In particular, layer-one protocols like *Tendermint Core*[3] are suitable for integrating QOF protocol based on their trust model. Tendermint is a Byzantine Fault Tolerant (BFT) consensus protocol that tolerates up to one-third of failures by stake. Tendermint forms the basis of all blockchain networks in the *Cosmos ecosystem*, a collection of interoperable networks that comes with tools for efficient and secure communication and coordination between the individual blockchains. There exist many other blockchain networks with stake-based consensus, into which the QOF protocol may be integrated (Algorand[4], Cardano[5], Internet Computer/DFINITY[6], Avalanche[7] and more).

Aptos[8] and *Sui*[9] [70] use related consensus models, but differ in a crucial aspect. Aptos runs *Block-STM* [41], an engine for parallel execution for smart contracts, and Sui works in a permissionless setting, where transactions are sent through a form of consistent broadcast that does not establish consensus. However, both also use quorums at their core, like the other protocols mentioned earlier. Since neither Aptos nor Sui imposes a total order on all transactions, they permit some amount of concurrent execution, which greatly improves scalability compared to traditional Byzantine agreement methods. This poses a challenge for integrating differential order fairness with their execution model, which remains open at this time.

[3] `https://tendermint.com`
[4] `https://algorandtechnologies.com`
[5] `https://cardano.org`
[6] `https://internetcomputer.org`
[7] `https://www.avax.network`
[8] `https://aptos.dev`
[9] `https://sui.io`

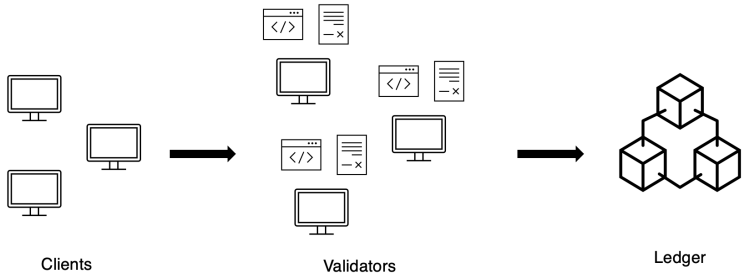


Figure 5.7: Clients submit transactions to the validators running the QOF algorithm. Validators agree on a fair order of transactions and run a smart contract to execute transactions on the ledger.

5.6 Evaluation

We evaluate the performance of the QOF protocol for estimating the cost of adding order fairness. Consensus is implemented by the HotStuff protocol in both cases. We first describe the experimental setup and then present the results.

5.6.1 Experimental setup

The starting point of our implementation of quick order fairness was the HotStuff implementation in the *bamboo* project [39]. The library[10] is implemented in the Go language and has several HotStuff flavors. Therefore, we modified the original code slightly to integrate it with the QOF protocol. Concretely, we modified the basic HotStuff protocol. The exact modifications are shown in the Section 5.4.

As the baseline for the evaluation, we use *bamboo*'s HotStuff implementation in the same benchmark setup as the QOF protocol. Therefore, we can compare the performance of both protocols in the same environment. We also compare and discuss the performance of QOF with other protocols such as Themis [49], Pompē [91], and unmodified bamboo HotStuff [39]. Although these protocols are tested in different

[10] `https://github.com/gitferry/bamboo`

benchmark setups, we can still get a rough idea of how QOF performs compared to other protocols.

Our benchmarks measure and analyze the protocol's latency (in milliseconds) and throughput (transactions per second). An important question that arises is how to measure these metrics. Should we measure time from when the client submits a transaction (client latency) or when a party receives it (server latency)? We chose server latency because it is more relevant to the protocol's performance. The client would additionally depend on the network delay between the client and the server.

All benchmarks are made on one Linux virtual machine running Ubuntu 22.04 within an OpenStack hypervisor, with 32 GB memory and 16 vCPUs of an AMD EPYC-Rome Processor at 2.3GHz and 4500 bogomips. In our benchmark, we vary the number of servers from 4 to 64 to see how the protocol scales. We generate transactions from a separate client party and send them to all servers.

5.6.2 Results

In this benchmark, we report the latency and throughput of the quick order fairness protocol and compare it to the baseline HotStuff protocol. We focus first on scalability and then evaluate how the payload size of a transaction and network delay affect the protocol's performance. Finally, we compare the performance of the QOF protocol to other protocols.

Scalability. A scalability benchmark is crucial for assessing a system's ability to handle increased workloads effectively. Figure 5.8 shows how the throughput in both protocols changes with the increase in the number of servers. With four servers, QOF reduces the throughput compared to HotStuff by about 300 transactions per second or approximately 5%. Throughput reduction shrinks as the number of servers increases; with 64 servers, the reduction is about 6%.

Figure 5.9 depicts the increase of latency with the increase of server numbers. Here, we observe that QOF increases latency by about 36 ms for four servers, which is around three times higher than HotStuff, and this difference continues to grow as the number of servers increases. The difference is primarily due to the increased computational load for processing the graph, which becomes more complex with more vertices and servers. Moreover, quick order fairness employs a Byzantine consistent

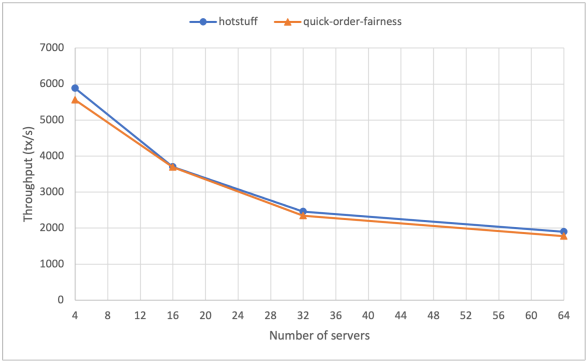


Figure 5.8: Performance scalability of HotStuff and Quick Order Fairness protocols, showing how throughput degrades with changing number of servers.

broadcast channel that adds latency and computational cost compared to the HotStuff implementation.

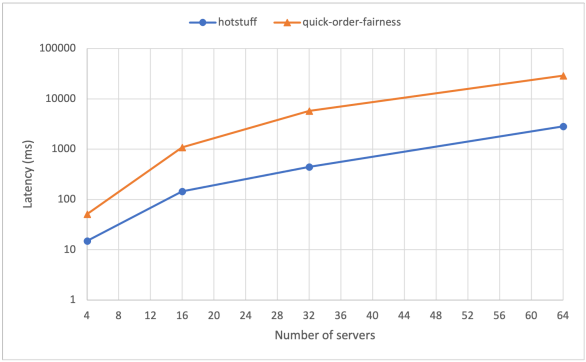


Figure 5.9: Performance scalability of HotStuff and Quick Order Fairness protocols, showing how latency increases with changing number of servers.

Transaction payload size. In this benchmark, we analyze how the payload size of a transaction affects the performance of the quick order fairness protocol and HotStuff. We vary the payload size of a transaction

from 256 bytes to 2048 bytes and show the results for the setup with four servers. Figure 5.10 shows the throughput of the protocol. The throughput data shows a gradual decrease as the payload size increases, with the throughput experiencing a reduction of approximately 14%. Figure 5.11 shows the protocol latency, and we see that the latency is constant for all payload sizes. We conclude that the payload size does not affect the performance of the quick order fairness and HotStuff protocol.

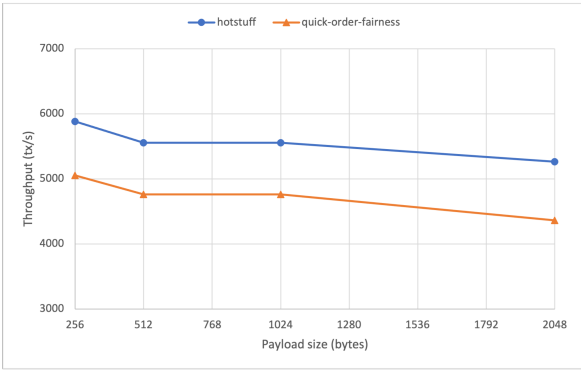


Figure 5.10: The figure shows how the throughput of the quick order fairness and HotStuff protocol changes for different payload sizes. We vary the payload size from 246 to 2048 bytes.

Network delay. In this test, we introduce additional network delay as it might happen in the real-world environment. We vary the network delay from 0 to 20 milliseconds. Specifically, the delay is determined within a range defined by the configuration settings, introducing variability that mirrors the unpredictability of actual network latencies. This approach enhances the realism of the test environment, enabling a more comprehensive evaluation of the system's performance under diverse network delay scenarios. Measurements taken in a network with four servers are shown in Figures 5.12 and 5.13, in terms of throughput and latency, respectively. The results indicate a substantial impact of network delay on system performance. As network delay increases from 0 to 20 ms, throughput decreases significantly while latency experiences a substantial increase. These findings underscore the sensitivity of throughput and latency to network delays, emphasizing the importance of minimiz-

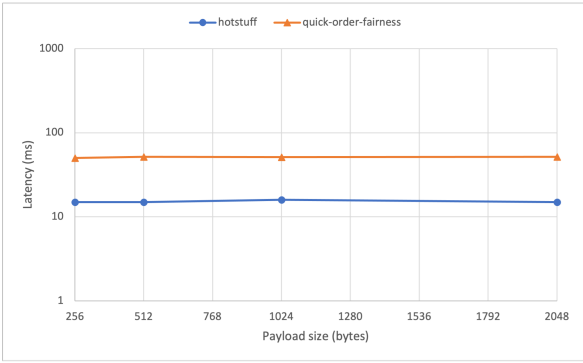


Figure 5.11: The figure shows how the latency of the quick order fairness and HotStuff protocol changes for different payload sizes. We vary the payload size from 246 to 2048 bytes.

ing network latency for optimal system performance.

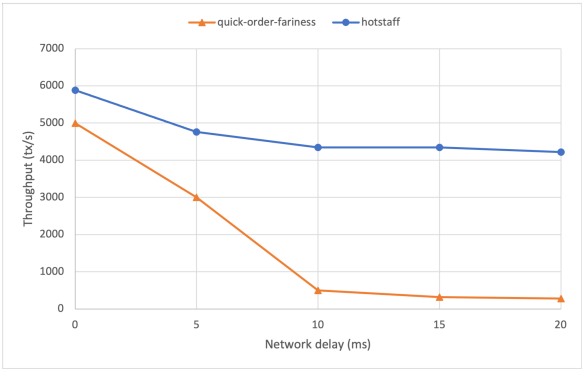


Figure 5.12: The figure shows how the throughput of the quick order fairness and HotStuff protocol changes for different network delays. The network delay changes from 0 to 20 milliseconds.

Comparison to other works. Comparing our benchmarks to other evaluations from the literature poses a challenge since the benchmark setups differ. Therefore, we can only get a rough idea of how the quick order fairness protocol performs compared to other protocols. In this

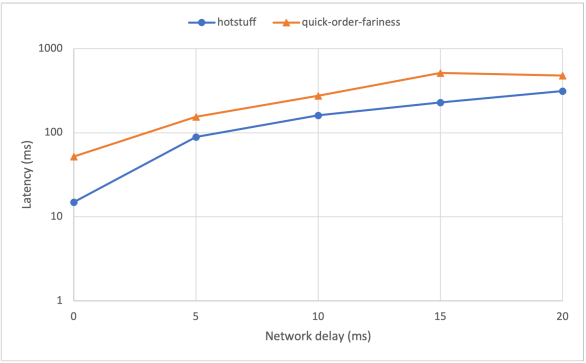


Figure 5.13: The figure shows how the latency of the quick order fairness and HotStuff protocol changes for different network delays. The network delay changes from 0 to 20 milliseconds.

comparison, we look at the performance of the Themis [49], Pompē [91] HotStuff variant, and unmodified HotStuff implemented in bamboo [39].

All benchmarks were conducted on diverse virtual machine providers, each instance equipped with 16 vCPUs and 32 GB of memory (except for Pompē, which utilized a machine with 64GB memory). We restrict our comparison to a common data point in all benchmark scenarios involving 32 servers executing the respective protocols.

Themis demonstrates remarkable performance, achieving a throughput 21 times higher than QOF. However, it is crucial to note that this substantial difference is influenced by the dedicated virtual machine per server in Themis, allowing much more parallelization. In contrast, QOF employs a setup where all 32 servers run within one VM, sharing the same vCPU.

The second-best performance is observed in the HotStuff variant of Pompē, surpassing QOF by approximately four times in terms of throughput, with a reduction in latency by a factor of three. This performance difference is presumably also influenced by the allocated dedicated VMs per server and by variations in the HotStuff implementation. Pompē leverages the original HotStuff implementation [60] written in C++.

Finally, we executed the HotStuff variant of bamboo within the same virtual machine as QOF. Specifically, in the scenario involving 32 servers, bamboo HotStuff achieves a throughput two times higher than QOF.

These performance results for bamboo HotStuff can be attributed to the complex graphs constructed by QOF. Furthermore, introducing a Byzantine-consistent broadcast channel incurred additional latency and computational costs compared to the HotStuff implementation.

5.7 Conclusion

Through the practical implementation of the QOF protocol, the work offers an executable representation, enhancing the protocol's accessibility and applicability. The systematic exploration of the protocol's integration into real-world systems, complete with smart contract pseudocode, lays a foundation to integrate the QOF protocol. The empirical evaluation, containing critical dimensions like scalability, throughput, and latency, not only confirms the protocol's efficacy but also provides invaluable insights for its practical deployment. Future work should optimize the codebase to enhance throughput and latency, ensuring the QOF protocol is ready for real-world deployment. Despite a slight reduction in throughput compared to the HotStuff protocol, the QOF protocol's complexity is justified by its resilience against front-running attacks. This work contributes practically and theoretically, extending the understanding and applying the quick order fairness protocol in decentralized systems.

Algorithm 8 Concrete implementation of vbc in QOF within HotStuff of bamboo (code for p_i).

```
    State:
254:    roundp ← 0: counter of how many times value is proposed (QOF round)
255:    roundd ← 0: counter of how many times value is decided
256:    view ← 0: counter of HotStuff protocol rounds
257:    votes ← [0]^{n×n}: matrix of votes for each block.id and voter

258:upon vbc-propose(v) do
259:    roundp ← roundp + 1    //in QOF this happens after building the graph
260:    // instead of abc-broadcast, add the proposal of p_i for the vbc instance
261:    // as a new transaction to the mempool
262:    t ← (VBC, roundp, v)
263:    mempool.addNew(t)

264:upon becoming leader do
265:    view ← view + 1
266:    payload ← mempool.some()                    // get up to 20 transactions
267:    block ← makeBlock(payload, view)
268:    send message [BLOCK, block] to all p ∈ P                  // vbc.go L348

269:upon receiving message [BLOCK, block] from p_j do
270:    σ ← sign(i, block.id)
271:    vote ← makeVote(view, i, block.id, σ)                  // hotstuff.go L105
272:    send message [VOTE, vote] to next view                   // to next leader

273:upon receiving message [VOTE, vote] from p_j
274:    such that verify(j, vote.block.id, vote.σ) do          // hotstuff.go L123
275:        votes[vote.block.id][vote.voter] ← vote
276:        if #(votes[vote.block.id]) > #(P) · 2/3 then       // quorum.go L66
277:            qc ← (vote.view, vote.block.id)
278:            view ← qc.view + 1
279:            if qc.view ≥ 3 then
280:                parentBlock ← getParentBlock(qc.block.id)
281:                grandparentBlock ← getParentBlock(parentBlock.block.id)
282:                if grandparentBlock.view + 1 = parentBlock.view
283:                    and parentBlock.view + 1 = qc.view then
284:                        // a block is appended to cBlocks
285:                        cBlocks ← commitBlock(grandparentBlock, view)

286:upon cBlocks is updated do
287:    commitedBlock ← cBlocks[0]                  // get latest committed block
288:    t ← commitedBlock.payload[0]
289:    (tag, round, val) ← t
290:    if tag = VBC ∧ round = roundp ∧ roundp > roundd then
291:        roundd ← roundd + 1
292:        vbc-decide(val)                                        // vbc.go L299
293:    else
294:        // discard proposed value
```

Algorithm 9 Abstract implementation of collapsing a graph.

```
295:Type:
296:     Graph(
297:          HashMap[string → Vertex] vertices)        // Collection of vertices.
298:     Vertex(
299:          string key,                                // Unique identifier of a vertex.
300:          bcchMessage data,                          // Transaction delivered by bcch.
301:          HashMap[string → Vertex] vertices)         // Connected vertices.
302:     bcchMessage(
303:          []byte message,                            // Transaction delivered by bcch.
304:          int round,                // Round in which the message was delivered.
305:          int fromProcess,                  // Party that delivered the message.
306:          string id)                                 // Unique identifier of a message.

307:upon CollapseGraph(G : Graph) do
308:     if #(G.vertices) ≤ 1 then
309:          return G
310:     else
311:          H ← NewDirectedGraph()
312:          components ← SCC(G)
313:          for c ∈ components do
314:               H.AddVertex(c)
315:          return H

316:function SCC(G : Graph)
317:     vertices ← G.vertices
318:     visited, components, stack ← [ ]
319:     for i ∈ {0, . . . , #(vertices)} do
320:          node ← vertices[i]
321:          if visited[node] = FALSE then
322:               DFS(visited, stack, node)
323:     transposed ← G.Transpose()
324:     visited ← [ ]
325:     for #(stack) ≠ 0 do
326:          // removes and returns the last element
327:          v ← stack[#(stack) − 1]
328:          if visited[v] = FALSE then
329:               transposed.Visit(visited, v, components)
330:     return components

331:function DFS(visited, stack, node)
332:     if visited[node] = FALSE then
333:          visited[node] ← TRUE
334:          for neighbour ∈ node.vertices do
335:               DFS(visited, stack, neighbour)
336:          stack.append(node)
```

Algorithm 10 Pseudocode of a smart contract.

```
337:// T is a set of ordered transactions; σ is a signature
338:upon receiving message [TRANSACTIONS, T, σ] do
339:     if verify(T, σ) = TRUE then
340:          for tx ∈ T do
341:               execute(tx)
```

Chapter 6

Defending Against Reordering in Decentralized Finance

6.1 Introduction

In this chapter, we focus again on the field of blockchain called decentralized finance (DeFi), which is very popular nowadays. DeFi is a promising field with many advantages over the traditional financial system. For example, it is censorship-resistant, permissionless, transparent, and open-source. Censorship resistance means DeFi systems are resilient to external attempts to control or restrict transactions. A permissionless system implies that individuals can participate in DeFi without prior approval or intermediaries. Transparency in DeFi ensures that all transactions are publicly recorded on a blockchain. Open-source denotes that the code and protocols governing DeFi platforms are publicly accessible, allowing collaborative development and innovation by a diverse community of contributors.

However, it also has some disadvantages. One of them is the *front-running* problem as we saw in Chapters 4 and 5. Let us say that Alice (a victim) wants to buy some amount of ETH. Eve (an attacker) observes mempool[1] by analyzing the pending transactions queued for con-

[1]The mempool, or memory pool, is a temporary storage area for unconfirmed

firmation, extracting relevant details, and strategically predicting market movements based on the anticipated impact of specific transactions, such as Alice's purchase of ETH. Eve uses this chance to gain profit by performing a so-called *sandwich attack*. She will create two transactions: one is a buying transaction with a higher gas fee than Alice's transaction, and the second one is a selling transaction that will be executed after the price goes up. Therefore, Eve will earn profit from selling the coin.

The profit that can be gained from this attack has been named *maximal* or *miner extractable value* (MEV) by Daian *et al.* [30]. According to the *Flashbots explorer*[2], the total MEV extracted from September 2020 until September 2022 is around 675 million USD. This number has been increasing until this day. Therefore, it is crucial to find a solution to this problem. The previous example was a simple example of a front-running attack. Many other types of front-running attacks are more complicated and more profitable [93]. This kind of attack is forbidden in the traditional financial system but must be prevented technically in DeFi.

The proposed defense methods can be categorized into four groups. The first category employs distributed cryptography to prevent side information leaks to malicious insiders, ensuring a *causal order* (Section 6.2) on the consensus-generated transaction sequence. The second category, known as *receive-order fairness* (Section 6.3), analyzes how individual parties participating in the consensus protocol receive transactions, imposing corresponding constraints on the resultant order. Chapters 4 and 5 addressed this category. The third category, called *randomized order* (Section 6.4), seeks to eliminate the influence of consensus validators on transaction order. The final category, *architectural separation* (Section 6.5), proposes to separate the task of ordering transactions and delegate it to a separate service. There is a significant interest in the research community to explore protection mechanisms against front-running attacks. The rest of this chapter describes some existing solutions and their advantages and disadvantages.

transactions in a blockchain network, holding transactions until they are selected by miners to be included in the next block.

[2] `https://explore.flashbots.net/`

6.2 Causal-order fairness

One of the first notions of fairness (in the context of this work) was introduced back in 1994 by Reiter and Birman [80]. The *causal* definition requires from the protocol that if some transaction tx caused or may have caused some other transaction tx', then tx should always be committed first to preserve the causal ordering defined by Lamport [56]. They define *causality* as following: "if *deliver*(m') is executed at a correct or honest server s when *deliver*(m) has not yet been executed at s, then m' was issued before m was decrypted anywhere." [80]. Later works by Cachin *et al.* [19] and Duan *et al.* [32] redefined the causal definition. All of these solutions require that transaction proposals are encrypted so malicious parties cannot learn the content of the transaction before it is committed.

6.2.1 Threshold cryptography

The foundations of blockchain systems can be traced back to distributed systems and cryptography. The problem of replicating services in a distributed system is well known. The main challenge is to ensure that all replicas agree on the same state. Reiter and Birman presented in their work [80] the first solution that satisfies *availability* and *integrity* together while maintaining a *causal order* in the distributed system. An innovative aspect of their research is eliminating the need for client authentication towards servers. Authors constructed a protocol that combines an atomic broadcast protocol and a public-key threshold cryptosystem.

A (k, n)-threshold cryptosystem creates a public key and n shares of the corresponding private key in a way that any message m that is encrypted using the public key, each share can be used to generate a partial result of m, where any k of these partial results can be combined to decrypt m. The system is secure if possessing $k - 1$ or less shares does not lead to decrypting a new ciphertext and is resistant to known plaintext attacks.

The main idea of their protocol, which uses the RSA threshold cryptosystem, is that each client encrypts every message m using a public key of the server and attempts to force k servers to cooperate to decrypt it. Each correct server abstains from broadcasting its partial result of m until the delivery sequence up to m is fixed locally. Therefore, even if a malicious server collects k partial results, the delivery sequence is

already fixed, and the malicious server cannot change it. The protocol achieves causality only if each server requires k partial results to decrypt a message.

This research has generated numerous possibilities for subsequent work and has motivated other researchers to delve deeper into this field of computer science. One of the critical questions at the time was how to make their protocol more practical, given that it relied on atomic broadcast protocols, and how to make it secure against *Chosen-Ciphertext Attack (CCA)* and use more standard encryption.

Cachin *et al.* [19] extended previous work by presenting *secure causal atomic broadcast* (SC-ABC). They defined new notions: message integrity, message consistency, and message secrecy, which, together with safety and liveness, guarantee causal order.

The proposed protocol employs the same encryption to guarantee a causal order among delivered messages and has a two-step delivery process (*schedule* and *reveal*). Protocol requires that only after a party schedules a ciphertext, it can reveal and broadcast its decryption share. It is not possible to have two consecutive schedule or reveal events.

Specifically, the SC-ABC protocol uses an $(f + 1, n)$-threshold cryptosystem for which parties share the decryption key. First, a trusted dealer generates a public key for the cryptosystem and distributes the corresponding private key shares to the parties. Then, it creates a unique label ID for each message by applying encryption using the public key. Since all instances of SC-ABC share the same public key, using labeled ciphertexts is necessary to distinguish different instances. Then, a ciphertext c is broadcasted to all parties. Upon delivery of c, the party computes a decryption share and sends it to all other parties together with c. It waits for $f + 1$ messages relevant to c. When it receives $f + 1$ messages, it decrypts c and delivers the message. After receiving the acknowledgment, the party processes the next delivery event and generates the corresponding acknowledgment.

The secure causal atomic broadcast protocol is secure against adaptive CCA attacks and works for $n > 3f$. However, the problem of expensive public-key threshold cryptosystems still left space for improvement.

6.2.2 Commit-reveal protocols

The previous techniques employ public-key encryption as a critical component, which may not be efficient for some scenarios. Duan *et al.* [32] proposed three novel secure causal Byzantine Fault Tolerant (BFT) pro-

tocols without using public-key cryptography. They introduced a protocol that can use any order-fair BFT protocol and any *non-malleable commitment with associated data*. The other two new protocols employ *asynchronous robust secret sharing*. This section will focus on the first construction (in the paper called CP1) since this solution is very prominent and has been discovered many times anew. It can thus be considered folklore.

The main idea of the CP1 protocol is to use a *non-malleable commitment with associated-data* scheme (NM-CAD). First, let us explain what a commitment scheme is. A commitment scheme is a cryptographic technique used to allow one to commit to a chosen value, which is revealed only after the committer decides to open the commitment. The NM-CAD scheme has three operations: *Cgen*, *Commmit*, and *Open*. *Cgen* generates a commitment key *ck*. *Commmit* takes as input commitment key *ck*, a message m and a header h and produces a pair (c, d) where c is a commitment and d is a decommitment (opening). *Open* takes as input *ck*, h, c, m and d and produces a decision bit. The associated data (header) of the commitment resembles the label concept from threshold cryptosystems.

The key concept behind CP1 is as follows: during the scheduling phase, a commitment is made to a specific value m along with an identifier (*ID*) using the underlying BFT protocol. In the revealing phase, both the value m and the corresponding opening d are processed through the same BFT protocol using the same identifier *ID*. In more detail, a system fixes a commitment key *ck*. A client picks an *ID* as the commitment header h for a message m and computes (c, d) using *Commit*. Then it sends (*ID*, schedule, c) to parties. Parties then verify the message and header, run BFT protocol to schedule the commitment and notify a client that they have delivered the message m. Once the client receives $f + 1$ such notification messages, it starts the reveal process by sending a message $(ID, \text{reveal}, (m, d))$ to the parties. Again, parties verify the correctness of m and d, use the *Open* method from NM-CAD and run the BFT protocol to deliver the message m. Parties then can process the m and respond to the client.

The main advantage of CP1 compared to the previous solution is that it does not rely on a trusted setup or interactive setup and can be realized using only symmetric cryptography, which is much more efficient. A disadvantage of this protocol might be that clients may fail to send messages and openings on time, or replicas could delay or drop them. To avoid this, authors propose an optimization such that in the

reveal process, once a correct party verifies m and d, it forwards them to the rest of the replicas. However, this does not resolve the issue of a faulty client that deliberately holds back the opening.

6.2.3 Time-lock encryption

Although the previous solution reveals the transactions after the order is fixed, it still has open issues. For example, an adversary may choose not to reveal the transaction, so the final ordering may not be optimal. To overcome this problem, Khalil *et al.* introduced the TEX protocol [51] that solves the problem as mentioned earlier by using a verifiable delay function (VDF) [13]. For instance, if a client does not reveal a transaction promptly, other parties in the system can decrypt it by solving a somewhat hard cryptographic puzzle.

The concept of *timed-release cryptography* was introduced by Rivest *et al.* in 1996 [84]. The authors' initial idea was to encrypt a message so no one can decrypt it until some defined upfront time has passed. In their work, they propose two ways to implement timed-release crypto. One is the use of *time-lock puzzles*, and the other one is the use of trusted agents. In this section, we will focus on the first approach.

The idea of time-lock puzzles requires creating a puzzle that reveals a decryption key after a certain amount of time. The challenge is designing it so that using large parallel computers only solves the puzzle after the dedicated time passes. The puzzle should be automatically solved at a given time, but a computer needs to work continuously on solving it before the key is revealed.

However, the previous solution does not scale to many participants, and TEX protocol [51] assumes synchrony at the commitment phase. Therefore, Doweck and Eyal [31] defined a new problem called *Multi-Party Timed Commitments (MPTC)* and presented two solutions to it. The first solution, called *Timed Capsule*, proves its correctness without relying on trust, incentives, or synchrony. The second solution, called *Capsule Chain*, is a frontrunning-resistant blockchain protocol that adapts Time Capsule. Nevertheless, this protocol is not perfect. Some open questions still tackle the problem of mining advantage, ensured output, transaction fees, and aborted blocks.

A more recent work from 2022 by Chiang *et al.* introduces the *Fair-PoS* [27] protocol, which is the first consensus protocol that achieves input fairness in the permissionless setting proposed. It is secure against

an adaptive adversary[3] and works in semi-synchronous networks. In this paper, authors introduce a new notion called *input fairness*. The input fairness holds for *"all blocks if (1) the adversary cannot decrypt an honestly encrypted input in block B before B is in the k-common-prefix and (2) encrypted inputs in B are eventually decrypted by all honest parties"*.

The input fairness guarantees that an adversary cannot observe the plain text before finalizing the transaction. This is achieved by encrypting inputs using *delay encryption*. Classic time-lock puzzles require a dedicated extraction process for each client input, which could be more problematic at higher throughputs. On the contrary, delay encryption allows encryption of all inputs in a block under a single unknown key. This key can be extracted as time elapses. Moreover, only one key extraction is needed for each block. However, the extraction procedure requires access to specialized hardware to ensure timely execution. In practice, clients are light and have limited resources, so this solution is impractical and would give an advantage to the clients with more resources. Therefore, the authors propose a novel *key extraction protocol*. This protocol requires staking parties to insert the extracted keys from previous blocks into child blocks of the same chain within a fixed schedule. The parties can only extend a chain if past key extractions are completed on time.

To conclude this section, we observe some disadvantages of the above-described approaches. For instance, transactions may be executed much later than when submitted to the network. In the time-lock puzzle approach, delay must adapt to network delay and the adversary's computational power.

6.3 Receive-order fairness

Leader-based protocols for consensus, i.e., atomic broadcast, allow some processes to unilaterally affect the final order of transactions, as we have already seen in Chapters 4 and 5. The concept of receive-order fairness is based on a committee in which every party has its local view of received transactions. The committee then agrees on the order of transactions, using consensus, and outputs them in the agreed fair order. The common challenge for all of those solutions is the *Condorcet paradox* [40]. In

[3] An adaptive adversary controls the network delay and may corrupt parties during the protocol execution.

the following part of this section, we will describe some of the existing solutions. Note that [50], [54], [91] were published around the same time.

6.3.1 Wendy

Kursawe [54] investigated the concept of relative order fairness, exploring methods to ensure fairness in the sequencing of transactions. The author also shows the impracticality of achieving fairness according to one of the more intuitive definitions. Subsequently, Kursawe introduces Wendy, a suite of low-overhead protocols designed to implement various fairness concepts. Introduced protocols have optimal resiliency in the asynchronous model ($n > 3f$) and optimal latency. The presented protocols can be added to any existing consensus protocol with a known set of validators.

For instance, one of the new proposed fairness definitions is *timed relative fairness*. This definition has slightly weaker fairness but allows much stronger liveness guarantees. The definition assumes that all parties have access to a local clock, so if there is a time $\mathcal{T}$ such that all hones parties saw transaction tx before time $\mathcal{T}$, and transaction tx' after time $\mathcal{T}$, then tx must be scheduled before tx'. The author claims that parties do not have to have synchronized local clocks, but the clocks must count forward, and there are no two identical timestamps. Using GPS as a time source is sufficient to make this approach practical.

Wendy's protocol must ensure that if tx needs to be scheduled before tx', then tx must be scheduled before tx' in an earlier or the same block. Since timestamps are incorporated into the block, the local ordering of requests within a block can be carried out after the block is delivered.

6.3.2 Pompē

Zhang *et al.* [91] introduced a new primitive called Byzantine ordered consensus. This primitive extends the Byzantine fault tolerant (BFT) protocol specification to express guarantees about the total order it generates. Assuming that parties use as their ordering preference the time they first see a transaction, authors proposed a new definition of order fairness, called *ordering linearizability*. This property ensures that if the highest timestamp from all correct parties for transaction tx is lower than the lowest timestamp from all correct parties for transaction tx', then tx is ordered before tx'.

The authors also proposed a new protocol called Pompē that implements Byzantine ordered consensus. Besides satisfying the standard safety and liveness properties of BFT, Pompē introduces an ordering phase that enforces ordering linearizability. In the ordering phase, the protocol utilizes timestamps to establish a total transaction order. To lock a position for a transaction in this total order, a party p_i initiates a two-step process. In the first step, p_i, having a transaction tx, gathers signed timestamps on tx from a quorum of $2f + 1$ parties, and the median timestamp among these signatures becomes the assigned timestamp for tx, determining its position in the total order. This step ensures that the assigned timestamp is upper- and lower-bounded by timestamps from the correct parties, a crucial aspect for achieving linearizability. In the second step, p_i broadcasts tx and its assigned timestamp, waiting for acceptance by a quorum of $2f + 1$ parties. A transaction tx accepted by this quorum is guaranteed inclusion in the totally ordered ledgers of correct parties and has its position in the ledgers determined by the assigned timestamp. However, the implementation of ordered linearizability is based on a median calculation, which malicious processes can easily manipulate, as shown in later work by Kelkar *et al.* [47].

6.3.3 Aequitas and Themis

Around the same time as the previous two works, Kelkar *et al.* [50] introduced a new property called *transaction order-fairness* which prevents adversarial manipulation of the ordering of transactions. They first introduce *receive-order-fairness* [50, Definition 1.1] informally as: "if sufficiently many (at least γ–fraction) parties receive a transaction tx before another transaction tx', then all honest nodes must output tx before tx'". However, the authors show the impossibility result of achieving this definition and propose a new relaxed definition called *block-order-fairness* [50, Definition 1.3]. In this definition, instead of delivering one transaction before another, both transactions can be delivered simultaneously in the same block. This definition also solves the problem of the Condorcet paradox, presented in Subsection 6.3.6.

Kelkar *et al.* presented a new class of consensus protocols, Aequitas, that satisfy block-order-fairness. These protocols can be realized as leader-based and leaderless protocols and work in synchronous and asynchronous settings, resulting in four protocols. Aequitas protocols use two primitives: FIFO broadcast and set Byzantine agreement. The protocols proceed in three stages:

1. Gossip stage: each party gossips its local transaction ordering to all other parties.

2. Agreement stage: parties reach a consensus on the group of parties whose local orderings are to be considered when determining the global ordering of a specific transaction.

3. Finalization stage: parties finalize the global ordering of a transaction by utilizing the set of local orderings determined in the agreement stage.

Aequitas protocol achieves impractically high $\mathcal{O}(n^3)$ communication complexity and provides only a weaker liveness property. In Section 4.4.1, we further discuss the limitations of Aequitas protocol.

A subsequent paper by Kelkar *et al.* [47] extends this approach to a permissionless setting. Kelkar *et al.* [49] presented another permissioned Byzantine consensus protocol, called Themis, that works in partially synchronous setting and tolerates faults of up to one-fourth of the total number of parties. It achieves the same fairness property as Aequitas but provides stronger liveness.

Themis achieves the *batch-order-fairness* property, with the parameter $\frac{1}{2} < \gamma \leq 1$, specifies that if γ fraction of honest parties receive a transaction tx before tx' from the client, then tx should be ordered no later than tx'. Batches emerge due to non-transitive Condorcet cycles in the message receipt times among parties. Nevertheless, the authors observe that cycles can persist for an arbitrary length of time, posing a liveness challenge for the Aequitas protocol. Therefore, instead of waiting to see all transactions to order them (like in Aequitas), Themis uses the technique of unspooling. This technique allows the protocol to output a batch in parts while still ensuring that all transactions in the same batch are output in an uninterrupted sequence.

Themis implements batch-order-fairness using a new primitive called *deferred ordering*. Using deferred ordering, blocks generated by a leader contain transactions that are fully ordered alongside only partially ordered transactions. Partially ordered transactions await total ordering by a subsequent honest leader. The finalization of these partially ordered transactions occurs within the network delay and does not necessitate waiting indefinitely for the ordering of future transactions, such as the occurrence of Condorcet cycles.

Compared to Aequitas communication complexity, Themis achieves a much lower communication complexity of $\mathcal{O}(n^2)$. Moreover, Kelkar *et*

al. proposed in the full version [48], another more theoretical design of Themis called SNARK-Themis, which uses SNARKs to achieve a communication complexity of $\mathcal{O}(n)$. SNARKs (Succinct Non-Interactive Arguments of Knowledge) are cryptographic proofs used in blockchain to verify computations without revealing sensitive information. However, this protocol requires a trusted setup.

6.3.4 Quick Order Fairness

Cachin *et al.* [21] relate order fairness to the fstandard validity notions for consensus protocols and highlight some limitations with the existing formalization. Based on this, authors introduce a new *differential order fairness* property that fixes these issues. They also present the *quick order-fair atomic broadcast protocol* that guarantees payload message delivery in a differentially fair order and is much more efficient than existing order-fair consensus protocols. It works for asynchronous and for eventually synchronous networks with optimal resilience, tolerating corruptions of up to one-third of the processes. Previous solutions required there to be less than one-fourth of faults. Furthermore, the protocol incurs only quadratic cost regarding amortized message complexity per delivered payload. The complete protocol definition is given in Chapter 4 and implementation in Chapter 5.

6.3.5 Fino

Another line of research commits to transaction ordering without seeing the content of a transaction, i.e., transaction content is hidden in a way that malicious parties cannot analyze and exploit it. This property is called *blind order-fairness*. An orthogonal line of research has shown a very good performance for BFT protocols designed around Directed Acyclic Graphs (DAG). DAGs provide high throughput, decouple transactions from metadata ordering, and guarantee a causal ordering of broadcasted messages. Malkhi and Szalachowski show in their work [66] how these two promising lines of research can be combined to mitigate the MEV problem.

The paper introduces a new protocol called Fino, a DAG-based BFT protocol that implements the Order-then-Reveal technique to achieve blind order fairness. In this protocol, clients encrypt transactions and send them to BFT parties. In this way, consensus protocol commits to an ordering of transactions blindly. The authors propose two techniques for

implementing Order-Then-Reveal; one is with threshold cryptography, and the other one is with verifiable secret sharing.

Fino integrates protection against MEV into a BFT protocol designed for the partial synchrony model, utilizing a DAG transport. BFT participants regularly aggregate pending encrypted transactions into a batch and employ the DAG transport to disseminate them. The crucial point for executing Order-then-Reveal on a DAG is that each view needs to wait until the reconstruction and verification of transactions from the previous view are successfully completed. In this manner, each view serves a dual purpose: initially, it blindly commits a new batch of transactions to the total ordering, and subsequently, it causally follows sufficient shares to reveal all previously committed transactions, establishing a deterministic opening.

6.3.6 Bounded unfairness

A new class of receive-order fairness definitions called *bounded unfairness*, introduced by Kiayias *et al.* [52], aims to minimize the distance between any pair of unfairly ordered transactions in the output of a distributed ledger. Their research shows that finding an optimal solution is connected to the graph properties, especially to *bandwidth* metric of strongly connected components in the transaction dependency graph. Therefore, the authors introduce a new property *directed bandwidth order-fairness* and a new protocol called *Taxis* that works in the permissionless setting. The paper presents two variants: one that matches the property but lacks in performance and liveness and a second that achieves liveness and better complexity but uses a relaxed version of the directed bandwidth called *timed directed bandwidth*.

The $\prec$ symbol describes the "order before" relation between transactions. This relation is irreflexive, asymmetric, and transitive. To achieve order fairness, we must know how sufficiently many parties received a transaction before the other party. This parameter is called the order fairness parameter and is denoted by $\varphi \in \mathbb{R}^+$. For two transactions tx and tx', definition of (φ, B)-fairness (where B is a bound in $\mathbb{N}$) says that $tx \prec^{\varphi} tx'$ if φ-fraction of input profiles rank tx before tx'.

Let $\mathbb{T}$ denote the set of all possible transactions with elements tx. A transaction profile is a bijection $R : \mathbb{T} \to [m]$ where $m = |\mathbb{T}|$. Considering the set of n parties, we denote $\mathcal{R}$ as a list of all transaction profiles. Output ordering σ is defined as $\sigma = F(\mathcal{R})$, where F is a serialization function that takes an indefinite number of transaction profiles $\mathcal{R}$ as

input.

The input profile is defined for each protocol party and ranks all received transactions in the order they were received. The goal is that tx cannot be serialized more than B positions later than tx'. This is achieved when the output ordering σ satisfies the following condition:

$$\sigma(tx) - \sigma(tx') \leq B. \tag{6.1}$$

The standard order fairness is satisfied for $B = 0$ and relaxed for $B > 0$. The goal is to minimize B, i.e., the number of unfair states in a state machine replication (SMR) context. Therefore, B is a function of the parties' input profiles and given pair of transactions. If and only if $tx \prec^{\varphi} tx'$ holds, the input profile defines a transaction dependency graph G, which includes an edge from tx to tx'. The authors observe that (φ, B)-fairness problem relates to graph bandwidth over graph G. The bandwidth problem tries to find a permutation of vertices in a graph to minimize the maximum distance between any pair of adjacent vertices. The *directed bandwidth* tries to minimize the length of edges that violate fairness. Therefore, bound B is defined as a function of the directed bandwidth of the strongly connected component that has two transactions or 0 if there is no such component.

6.3.7 Condorcet attack

Even though there exist several receive-order fairness solutions, they are not perfect. Each solution must deal with the so-called *Condorcet paradox*. The Condorcet paradox is a situation in which protocol cannot decide on a fair order of transactions. Let us assume there are three processes and three transactions are broadcasted. Each process receives transactions in a different order. The first process receives transactions as $[tx_1, tx_2, tx_3]$, the second process receives transactions as $[tx_2, tx_3, tx_1]$, and the third process receives transactions as $[tx_3, tx_1, tx_2]$. This is the case when a cycle in a graph appears. Some solutions deal with such cases in a way that they output them together. For instance, Aequitas [50] uses term *block-order fairness* because they output cycles within the same "block" [4].

However, this is not a perfect solution and a recent research paper by Vafadar and Khabbazian describes *Condorcet attack* [88]. In this attack, an adversary tries to break the fair ordering by injecting Condorcet

[4]Term block can be confused with low-level blocks in mining-based protocols.

cycles and trapping honest transactions inside the cycle. The authors show how the attack works and propose three methods to mitigate the attack.

A malicious client executes the attack in three steps:

- In the *initialization* phase, the malicious client assigns a subset of transactions to each party from one partition. Different parties receive different sets of transactions. The client then determines an order for each subset and sends it to the corresponding parties in the partition.
- In the *pause* phase, the client waits a specific amount of time to ensure that honest transactions arrive so they will be trapped between malicious transactions.
- In the *finalization* phase, the client closes the Condorcet cycle by sending a new set of transactions to each party in a determined order.

Performing such an attack on the Aequitas protocol results in outputting the whole cycle as a block of transactions and executing them alphabetically (as defined in the protocol), which might break the order in which honest transactions arrived. On the other hand, Themis [49] attempts to break links in the Hamiltonian cycle to prevent cycles.

Therefore, the authors propose three techniques to prevent the Condorcet attack:

- Ranked pairs batch-ordering choose a maximal subset of E' edges of the input graph $G = (V, E)$, with high weights such that $G' = (V, E')$ is a directed acyclic graph (DAG). The limitation of this strategy is that it works only when all parties are honest.
- Post-decryption resolution is a technique that allows parties to resolve the order of transactions after decryption and partition them into independent groups. The limitation of this strategy is a significant computational overhead.
- Broadcast technique attempts to break the attack pattern by immediately broadcasting received messages to all parties. Therefore, transactions arriving in the attack's last phase are irrelevant because parties have already received them. The limitation of this strategy is that it is not applicable if the adversary has strong control over the network and can delay broadcasted transactions.

6.4 Randomized order

The blockchain community widely knows about the concept of randomizing transaction order within a block. Yanai [2] and others have explored this concept and implemented it in practical applications. For example, Randomspam [78] recognizes the potential for spamming attacks linked to randomized transactions. In these situations, attackers strategically insert numerous low-cost transactions to enhance the chances of precisely placing some of these transactions at a lucrative transaction point. This section will describe some existing solutions that use randomization to prevent front-running attacks.

6.4.1 Partitioned and permuted protocol

Alpos *et al.* [4] presented a novel decentralized solution Π^3 called "Partitioned and Permuted Protocol." The main idea is that the final order of transactions in the block is determined by a permutation Σ, which the miners generate. However, the problem is that miners can still influence the order of transactions most beneficially if Σ is known before creating a block. The authors solve this problem by revealing Σ only after the block to be permuted has been mined. If Σ is revealed after creating the block, leaders could collude, try different permutations, and choose the most beneficial. Therefore, leaders commit to their contributions to the permutation before the block is mined. In this way, the bias introduced by the miners is bounded. The described protocol implements a delayed reward release mechanism to incentivize leaders to behave honestly. If they diverge from the execution of the protocol, their reward is lost.

Leaders may still behave dishonestly because running a sandwich attack is sometimes more profitable than getting the reward. To increase the protection, authors implemented transaction *chunking*. Each transaction is split into m chunks, so the probability of profitable permutation is decreased. Moreover, with the increase in the number of chunks, the probability of profitable permutation approaches zero.

The Π^3 protocol requires no external resources. It can be implemented on top of any blockchain protocol, and the construction occurs with minimal overhead except for a slight latency increase. The standard security properties of an underlying blockchain remain unchanged.

6.4.2 Frontrunning in the XRP ledger

Front-running attacks are not exclusive to programmable blockchains; even non-programmable blockchains, such as Ripple, can fall victim [87]. In the initial XRP Ledger version, attackers exploited lower transaction IDs to ensure execution precedence over victims. Ripple's developers addressed this issue by introducing a new transaction ordering strategy that employs a pseudo-random shuffle. This strategy incorporates a random salt derived from accepted transactions, ensuring consistent ordering computation across all participants.

The shuffling algorithm operates as follows: it first computes an account key for each transaction by combining its account ID and salt through the XOR operation. The algorithm then arranges the transactions through pairwise comparisons based on certain criteria. If the account keys differ, the transactions are ordered in ascending order. If the transaction sequence numbers differ, they are arranged in ascending order. If all else is equal, the transactions are ordered by their transaction hash. The result is a pseudo-randomly shuffled list of transactions, with transactions from the same account grouped in their original submission order. In the XRP Ledger, transactions are processed sequentially by parties. Even if a transaction fails, it still appears in the ledger with a tec-class result code placed at the end of the canonical list. After processing all transactions, failed ones are retried but remain at the end of the canonical list.

Tumas *et al.* [87] identified two attack strategies, indicating that the shuffling algorithm enhances security against naive attacks but does not entirely prevent them. The authors investigated historical main net data to assess the potential upper limit of profit, approximating 1.4 million USD, that attackers could have gained by consistently front-running opportunities on the XRP Ledger over two months.

6.4.3 Uniform random execution

Randomizing the transactions order extends beyond security measures; for example, Chitra *et al.* [28] proposed the Uniform Random Execution mechanism for privacy enhancement in constant function market makers (CFMMs). This protocol randomly splits and permutes the ordering of large trades to provide user privacy.

CFMMs are widely adopted for decentralized trading, facilitating hundreds of billions of dollars. However, these mechanisms lack pri-

vacy features for users. Authors quantify the trade-off between pricing and privacy in CFMMs. They explore a straightforward privacy-enhancing approach called Uniform Random Execution that provides (ϵ, δ)-differential privacy. The privacy parameter ϵ relies on the curvature of the CFMM trading function and the volume of executed trades. This mechanism applies to any blockchain system, allowing smart contracts to access a verifiable random function.

6.5 Architectural separation

The separation of services (roles) in the architecture of a blockchain is yet another promising approach to prevent front-running. The main idea is to either have a separate service that orders transactions or to separate the task of ordering transactions into two roles: block builder and block proposer. In the following part of this section, we will describe some existing solutions starting with use of multiparty computation [2], [11], [12], [53], [63]. Then we will describe proposer-builder separation [17], [43]. At the end of this section, we will describe solutions that use trusted hardware [59] and transaction ordering policy for centralized sequencers [67].

6.5.1 Multiparty computation

Distributed computation across multiple parties, known as multiparty computation (MPC), allows computation on private data without revealing this data to other parties. MPC is an established technology that is native and close to practical deployment. MPC protocols and implementations have been developed with practical considerations in mind, making them suitable for integration into operational systems and applications. This technology can tackle the problem of transaction ordering as well.

Fairness in MPC is interpreted as a condition that either all parties get output or none. Kiayias *et al.* [53] introduced a new robust MPC protocol with compensation. In this protocol, compensation is used to guarantee fairness. Moreover, the protocol works in a constant number of rounds.

Lu *et al.* proposed a new asynchronous MPC implementation called *HoneyBadgerMPC* [63], arguing that the previous implementations do not address fairness in their solutions. However, HoneyBadgerMPC

guarantees fairness and output delivery in a malicious setting without depending on timing assumptions. *AsynchroMix* mixing service employs HoneyBadgerMPC and runs in asynchronous epochs, where in each epoch, the system selects a subset of clients and mixes their inputs before publishing them. HoneyBadgerMPC uses an asynchronous broadcast protocol to receive inputs and initiate mixing epochs in a distributed way.

Blinder, anonymous committed broadcast has been proposed by Abraham *et al.* [2]. Blinder guarantees security (anonymity) and robustness and is censorship resistant (honest client cannot be blocked from participating). The authors agree that one of this protocol's applications is preventing front-running in decentralized exchanges. Blinder overcomes this problem as all transactions are committed and opened only after all trade orders are submitted. Moreover, since the protocol has anonymity, it can be used as a communication medium for future anonymized solutions for distributed exchanges.

Insured MPC is yet another construction, presented by Baum *et al.* [11]. The protocol results in the output being fairly delivered to the parties, or there is proof that a set of parties misbehaved while running MPC. The resulting proof is used for financial punishments of malicious parties and can be verified by third parties.

Baum *et al.* [12] extended the previously mentioned Insured MPC and created an efficient, universally composable privacy-preserving decentralized exchange immune to front-running. A set of servers runs a private cross-chain exchange order-matching protocol in this exchange. When parties are correct, the on-chain complexity is similar to the complexity of a centralized exchange. In case of an active adversary, clients get a refund and do not lose their funds. This work involves an experiment and shows that the results are efficient enough to be used in practice. The main building blocks are MPC and threshold signatures with identifiable abort.

The main drawback of all previously listed approaches is the same – the validity of a transaction can be checked only after it is revealed.

6.5.2 Proposer-builder separation

According to one of the Ethereum[5] founders, Vitalik Buterin, the best-known solution for preventing MEV is proposer-builder separation (PBS)

[5] `https://ethereum.org/`

paradigm [17]. Initially, Ethereum was designed in a way that a block proposer builds and proposes a block in the network. However, the block proposer has too much power, can censor transactions, and mount front-running attacks. Therefore, the proposer-builder separation is proposed.

Buterin proposed a new concept that separates the task into block building and block proposing by introducing three roles: builder, proposer, and relays. The role of the block builder is to select and order transactions in a block with the most value for the proposer. The proposer then chooses a block with the highest value and proposes it to the Ethereum network. The builder and proposer are connected via a relayer, which receives blocks from builders and forwards them to proposers. PBS draft specification aims to achieve two main goals: censorship resistance and decentralization of block validation. The proposed paradigm got attention with *the merge* in September 2022 when Ethereum was transformed from Proof-of-Work to Proof-of-Stake.

A few months after *the merge*, Heimbach *et al.* [43] published an analysis on whether PBS achieves its promises and how it would work in practice. One of the design goals of PBS is decentralizing block validation by not giving large parties an advantage in block building. The reality is different. The professionalized builder has an advantage in building profitable blocks. Another finding shows a high degree of centralization for relays and builders.

Moreover, Heimbach *et al.* show that PBS does not achieve the promised censorship resistance in practice. They found that a significant proportion of blocks are built by censoring relays. Another question arises from trust in relays since they are responsible for holding blocks from builders in escrow for validators. They keep the block's content private until the validator proposes it to the network. They show that only one among all relayers delivers the promised value. According to the current roadmap [77], PBS is still in the research phase with essential design questions to be solved.

6.5.3 Fairness using trusted hardware

Trusted hardware refers to specialized hardware components, such as secure enclaves, that provide a secure and isolated environment for sensitive computations or data storage. These components enhance the overall security of systems by safeguarding critical operations against unauthorized access or tampering, offering a higher level of confidence in the integrity and confidentiality of computing processes.

The concept of using trusted hardware to guarantee a fair order is introduced by Li *et al.* [59]. Authors use a trusted execution environment (TEE) [29], [58], [76] to operate the mempool. The benefits are the following:

1. Sequencing algorithms are coded into TEEs, ensuring compatibility with arbitrary rules.

2. Once the transaction enters the TEE, it is faithfully ordered according to predefined rules.

3. Transactions are encrypted and remain undisclosed until proposed in a block, mitigating front-running attacks during transaction submission.

Li *et al.* provide a generic fairness definition called *verifiable fairness*. This definition relaxes the sequencing rules from the previous fairness definitions, and by generalizing these rules to a publicly known predicate, the definition becomes more flexible and inclusive. Moreover, it allows the adoption of application-specific sequencing rules without necessitating the formulation of a new definition. The paper introduces *verifiable fairness with accountability* that guarantees that given an arbitrary rule, any client can verify according to the rule that a party faithfully sequences a list of transactions.

The work proposes a solution that satisfies the verifiable fairness definition and implement a functional prototype by using Go Ethereum [36] and OpenSGX [46]. At the high level, the proposed solution works as follows. Initially, the parties establish their TEEs and transfer the mempool data into these secure enclaves. To submit a transaction tx, the client acquires the public key pk of the TEE at party p_i. Subsequently, the client encrypts the transaction tx using pk and transmits the ciphertext to p_i.

Upon receiving the ciphertext, p_i forwards the encrypted transactions to its TEE-based mempool. The mempool decrypts the transaction using sk, corresponding to pk, and adds tx to its secure memory. Simultaneously, TEE generates a signature for tx using its secret key and transmits it (via party p_i) to the client, serving as a receipt confirming the inclusion of tx in the mempool. When party p_i requires a list of ordered transactions, the TEE retrieves and organizes a list of pending transactions l from its secure memory through the *Select* and *Order* algorithms. The *Order* algorithm outputs an ordered sequence

of transactions based on a defined ordering rule, which any party in the system knows. Subsequently, the TEE signs the ordered result r using the secret key and produces a signature σ_r. The result r and the signature σ_r are then returned to party p_i.

Now, p_i can engage in the consensus protocol. Upon receiving p_i's proposal, other parties verify the validity of the order by checking σ_r. If p_i behaves honestly, tx will ultimately be committed on-chain. After tx is committed, p_i initiates an operation to remove tx from the mempool.

Although TEE brings only benefits to their solution, there are still some disadvantages to using TEEs. The problem is an untrustworthy TEE host which can behave maliciously. Although the host cannot view the transaction content or manipulate their order, it can withhold encrypted transactions from entering the TEE or prevent that output from the TEE reaches the other nodes. Therefore, all participants have to trust the hardware vendor for security. Another problem is poor performance, which is confirmed by the evaluation of the prototype implementation. A benchmark shows that the prototype is slightly slower than the original Ethereum node. Finally, many attacks on SGX as TEE are known [37], [74], [92].

6.5.4 TimeBoost

Centralized sequencers use ordering policies to order transactions before they are executed. Some existing ordering policies used in blockchain are first-come, first-serve (FCFS), per-block transaction bidding, and block auctions. However, those policies have several disadvantages. Therefore, Mamageischvili *et al.* proposed a new ordering policy called *TimeBoost* [67].

TimeBoost calculates a scoring function that considers timestamps and bids and orders transactions based on this score. More specifically, the scoring function is defined as

$$S(tx) = \pi(b) - t, \tag{6.2}$$

where π is a bidding function, b is a bid, and t is a timestamp. The bidding function is

$$\pi(b) = \frac{gb}{b + c}, \tag{6.3}$$

where g is time and c is some constant. By increasing the bid, users decrease their effective timestamp, i.e., increase their score. The main

idea is that the transaction with the highest score is executed first. There is also a limit on how much time can be bought. Therefore, no transaction can outbid a transaction received earlier, ensuring a low transaction finalization time. Moreover, TimeBoost can work with encrypted transactions. This can be implemented using threshold decryption by a committee or trusted hardware.

Although this line of research is orthogonal to the research of decentralized protocols ([21], [49], [50]), they propose the use of decentralized protocols for agreeing on the scoring of transactions. In other words, they can be used as input for TimeBoost to agree on the timestamp and the bid of transactions.

6.5.5 Rationally binding commitments

As we saw in previous sections, there is extensive research on fair ordering in a data-independent fashion. However, all protocols have the same limitation: they work only under the assumption that enough parties in the protocol are honest.

To overcome this problem, Wadhwa *et al.* [89] constructed a novel concept of *rationally binding transactions* that assumes rationality instead of honesty, i.e., a rational party may take any action that maximizes its utility (profit). They also constructed an automated market maker (AMM) called *AnimaguSwap* that uses rationally binding transactions to prevent front-running attacks.

The paper first presents a framework that captures existing ordering protocols for data-independent ordering, such as fair ordering and content-oblivious protocols. Using this framework, authors prove the impossibility of constructing such protocols in a way that they are secure in the presence of rational parties. The authors proposed two solutions to overcome the impossibility. The first approach is to use TEEs to restrict the parties from reconstructing the transaction before it has been committed on the chain. The second approach is *AnimaguSwap*, which can achieve data-independent ordering in the presence of rational parties.

The protocol requires parties (stakers) to put down monetary investment, stake, that can be burned in case of misbehavior. One party works as a "flipper". The user samples a random bit $b \in \{0, 1\}$ to create a rationally binding commitment to transaction tx. Depending on the chosen bit, a user creates a transaction that is either the intended transaction tx or a related but different transaction tx^*. The flipper receives

b as a deniable message[6] from the user, sings the bit, and sends it back to the user as an acknowledgment. The user then creates a transaction and shares it with other parties. In order to open the commitment, parties reveal the shared transaction, and the flipper reveals b. After this, the transaction is executed. The flipper is incentivized to behave honestly because if it reveals a wrong b, it will get slashed since parties can use the acknowledgments as evidence. Concretely, in *AnimaguSwap*, if user transaction tx is selling an asset, the reverse transaction tx^* is buying the same asset. When parties decide to collude, they have to guess which transaction will be executed, so it is not profitable to execute the sandwich attack.

6.6 Practical systems

Earlier sections explore theoretical solutions, but certain systems also aim to defend against reordering attacks in practice. In this section, we will describe some of them. Starting with Chainlink's proposal for building a Fair Sequencing Services [14], we will continue with Espresso Systems and Offchain Labs' effort to create a decentralized sequencer [34]. Finally, we look at the Oasis network [73] and their approach to preventing MEV attacks.

6.6.1 Fair sequencing services

The research team of blockchain company Chainlink[7] proposed in their whitepaper a new concept called Fair Sequencing Services (FSS) [14]. They intend to build a fair DeFi ecosystem and to support developers in building DeFi contracts protected from market manipulations.

Chainlink proposes that FSS should have three components: *monitoring, sequencing* and *posting* of transactions. Monitoring means that oracle nodes both monitor the mempool of the mainchain and allow off-chain submissions. Sequencing means that oracle nodes order transactions according to an ordering policy. Finally, when transactions are ordered, they are posted to the main chain.

There are two possible ways to realize fair transaction sequencing in FSS. One is to implement order-fairness protocol, such as protocols described in Section 6.3. When writing the whitepaper, authors suggest

[6] This is needed because the flipper itself can generate a message.

[7] `https://chain.link/`

using some of the existing order-fairness algorithms, for instance, Aequitas [50]. However, this algorithm is not ideal for practical use due to significant communication overhead. The authors believe that in the future, a practical framework for order fairness could be implemented in FSS.

The second method is secure causality preservation, described in Section 6.2. However, this method requires additional cryptographic methods in FSS. It aims to hide the transaction data, wait until the order is final, and only then reveal the transaction data. However, these techniques do not hide metadata. Chainlink claims that metadata, such as an IP address or an Ethereum address, can still be used for performing front-running. Therefore, they propose, as a future work, an exploration of combining two described methods.

6.6.2 Espresso systems

Recall that Section 6.5.4 described a centralized sequencer called TimeBoost. However, centralized sequencers in Layer 2 (L2) roll-ups pose a problem, as they are a single point of failure, threatening the entire roll-up if they malfunction. Additionally, applications from different L2 ecosystems are more challenging to interoperate. To overcome these problems, a new concept of decentralized sequencer called Espresso sequencer is proposed by Espresso Systems [35]. Consisting of two fundamental components, the Espresso sequencer incorporates the HotShot Consensus and Tiramisu data availability, which are responsible for sequencing transactions and guaranteeing data availability.

In a recent document [34], Espresso Systems and Offchain Labs (authors of TimeBoost) announced a collaborative effort to create a decentralized version of the TimeBoost transaction ordering sequencer. The document details the proposed roadmap and development steps for integrating decentralized Timeboost with the Espresso Sequencer, with plans to start discussions, present design considerations, build integrations, conduct research and development, and eventually make the protocol available for roll-ups using the Espresso Sequencer. However, there is currently no further information available on the system's current operational status or deployment in practice.

6.6.3 Oasis network

The Oasis network [73] is a Layer 1 proof-of-stake smart-contract platform designed to offer scalability and privacy in blockchain applications. The platform features a modular design that facilitates the integration of different consensus mechanisms and allows independent parallel runtimes, known as *ParaTimes*, to employ various verifiable and confidential computing techniques simultaneously. ParaTimes is implemented using TEEs for private computation. TEEs act as secure and isolated environments where the execution of smart contracts takes place. This TEE-based ParaTime allows for cost-effective, confidential execution of smart contracts while maintaining the ability to verify the correctness of computations. The TEEs ensure the execution environment is secure, providing privacy for sensitive data and preventing unauthorized access or manipulation of the contract's logic. TEEs are crucial in achieving the Oasis network's goal of enabling privacy.

In terms of defending against reordering, Oasis employs a multifaceted approach to prevent MEV attacks in its ecosystem [72]. Utilizing an MEV Blocker, applications on the Oasis platform automatically ensure protection against MEV. Integration of the Oasis Privacy Layer (OPL) enhances user security by enabling confidential communication and asset transfers within applications. Using a confidential mempool, running in TEEs adds a layer of protection by resisting front-running and other attacks. By strategically separating verification processes on-chain and executing computations in a separate, confidential, and MEV-resistant environment, Oasis reduces the risk of MEV exploits for its users.

6.7 Conclusion

Finding a comprehensive solution to the front-running challenge in DeFi requires understanding the strengths and limitations inherent in various proposed approaches. We group these approaches into four categories: causal order, receive-order fairness, randomized order, and architectural separation.

For example, while time-lock puzzle encryption seems simple, it risks delayed transaction execution. MPC faces constraints, allowing validation only post-revelation, leaving room for exploitation. Despite promises, PBS encounters practical issues like censorship resistance and relay reliability. TEEs, while advantageous, pose challenges with un-

trustworthy hosts and concerns about energy consumption. DeFi lacks a one-size-fits-all solution; choosing the suitable defense method depends on specific needs. A collaborative effort in the research community is vital for developing adaptive, resilient solutions and contributing to establishing a robust DeFi ecosystem.

Chapter 7

Conclusion

This dissertation addressed two segments of blockchain technology: consensus protocols and decentralized finance. The first part of this dissertation focused on understanding the Ripple consensus protocol, analyzing its safety and liveness properties. The second part of this dissertation addressed front-running attacks in decentralized finance, proposing a theoretical and practical solution to the problem and giving an overview of other existing solutions.

Analyzing Ripple, one of the oldest public blockchain platforms reveals some issues despite its significant place in the world of cryptocurrencies. While its native XRP token has long held a prominent position in market capitalization, concerns about the liveness and safety of the Ripple consensus protocol have emerged. This work provided an independent, abstract protocol description, highlighting potential safety and liveness violations. Our analysis underscored the requirement for close synchronization, seamless interconnection, and fault-free operations among the validators actively participating in the Ripple network.

Another problem, often seen in decentralized finance, called a front-running attack, is also addressed in this dissertation. In a front-running attack, a malicious party tries to manipulate the order of transactions in a block to gain a financial advantage. The proposed solution, the quick order-fair atomic broadcast protocol, offers a promising solution to transaction reordering challenges. Operating efficiently in asynchronous and eventually synchronous networks, with optimal resilience against faulty parties, this protocol performs better than similar fair ordering protocols. The practical implementation of the QOF protocol enhances

its applicability, with empirical evaluations confirming its efficiency and providing insights for real-world deployment. Despite a reduction in throughput compared to similar protocols, the QOF protocol's resilience against front-running attacks justifies its complexity, practically and theoretically contributing to decentralized systems.

The last chapter of this dissertation addressed front-running challenges in DeFi, which involved the strengths and limitations of proposed approaches. The conclusion is that there is no one-size-fits-all solution to the front-running problem. Each defense method has its strengths and weaknesses. So, the key is to choose the right solution based on the specific needs of each situation. It is like picking different tools for different jobs — finding the best fit for each case in the DeFi landscape. The dynamic nature of DeFi and blockchain technology necessitates a collaborative effort within the research community to develop adaptive and resilient solutions. As the field continues to evolve, the journey towards enhancing security measures remains an ongoing process, encouraging innovation and contributing to establishing a robust DeFi ecosystem.

Bibliography

[1] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *PODC*, ACM, 2019, pp. 337–346.

[2] I. Abraham, B. Pinkas, and A. Yanai, "Blinder - scalable, robust anonymous committed broadcast," in *CCS*, ACM, 2020, pp. 1233–1252.

[3] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, 1985.

[4] O. Alpos, I. Amores-Sesar, C. Cachin, and M. Yeo, "Eating Sandwiches: Modular and Lightweight Elimination of Transaction Reordering Attacks," in *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 286, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 12:1–12:22.

[5] I. Amores-Sesar, C. Cachin, and J. Micic, "Security analysis of ripple consensus," in *OPODIS*, ser. LIPIcs, vol. 184, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 10:1–10:16.

[6] E. Androulaki, A. Barger, V. Bortnikov, *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *EuroSys*, ACM, 2018, 30:1–30:15.

[7] F. Armknecht, G. O. Karame, A. Mandal, F. Youssef, and E. Zenner, "Ripple: Overview and outlook," in *TRUST*, ser. Lecture Notes in Computer Science, vol. 9229, Springer, 2015, pp. 163–180.

[8] A. Asayag, G. Cohen, I. Grayevsky, *et al.*, "A fair consensus protocol for transaction ordering," in *ICNP*, IEEE Computer Society, 2018, pp. 55–65.

[9] L. Baird, *The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance*, Swirlds Tech Report, SWIRLDS-TR-2016-01, 2016. [Online]. Available: `https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf`.

[10] C. Baum, J. H. Chiang, B. David, T. K. Frederiksen, and L. Gentile, "Sok: Mitigation of front-running in decentralized finance," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 13412, Springer, 2022, pp. 250–271.

[11] C. Baum, B. David, and R. Dowsley, "Insured MPC: efficient secure computation with financial penalties," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 12059, Springer, 2020, pp. 404–420.

[12] C. Baum, B. David, and T. K. Frederiksen, "P2DEX: privacy-preserving decentralized cryptocurrency exchange," in *ACNS (1)*, ser. Lecture Notes in Computer Science, vol. 12726, Springer, 2021, pp. 163–194.

[13] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *CRYPTO (1)*, ser. Lecture Notes in Computer Science, vol. 10991, Springer, 2018, pp. 757–788.

[14] L. Breidenbach, C. Cachin, B. Chan, *et al.*, "Chainlink 2.0: Next steps in the evolution of decentralized oracle networks," *Chainlink Labs*, vol. 1, pp. 1–136, 2021.

[15] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.

[16] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: `https://ethereum.org/`.

[17] V. Buterin, *Proposer/block builder separation-friendly fee market designs*, Jun. 2021. [Online]. Available: `https://ethresear.ch/t/proposer-block-builder-separation-friendly-fee-market-designs/9725`.

[18] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011.

[19] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 2139, Springer, 2001, pp. 524–541.

[20] C. Cachin and J. Micic, "Quick order fairness: Implementation and evaluation," *arXiv preprint arXiv:2312.13107*, 2023.

[21] C. Cachin, J. Micic, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 13411, Springer, 2022, pp. 316–333.

[22] C. Cachin and B. Tackmann, "Asymmetric distributed trust," in *OPODIS*, ser. LIPIcs, vol. 153, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 7:1–7:16.

[23] C. Cachin and M. Vukolic, "Blockchain consensus protocols in the wild (keynote talk)," in *DISC*, ser. LIPIcs, vol. 91, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 1:1–1:16.

[24] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[25] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, vol. 5959, Lecture Notes in Computer Science, Springer, 2010.

[26] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," *CoRR*, vol. abs/1802.07242, 2018.

[27] J. H. Chiang, B. David, I. Eyal, and T. Gong, "Fairpos: Input fairness in permissionless consensus," in *AFT*, ser. LIPIcs, vol. 282, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 10:1–10:23.

[28] T. Chitra, G. Angeris, and A. Evans, "Differential privacy in constant function market makers," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 13411, Springer, 2022, pp. 149–178.

[29] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, p. 86, 2016.

[30] P. Daian, S. Goldfeder, T. Kell, *et al.*, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *SP*, IEEE, 2020, pp. 910–927.

[31] Y. Doweck and I. Eyal, "Multi-party timed commitments," *CoRR*, vol. abs/2005.04883, 2020.

[32] S. Duan, M. K. Reiter, and H. Zhang, "Secure causal atomic broadcast, revisited," in *DSN*, IEEE Computer Society, 2017, pp. 61–72.

[33] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[34] Espresso Systems, *Espresso Systems and Offchain Labs Release Roadmap for Decentralized Timeboost*, 2023. [Online]. Available: `https://medium.com/@espressosys/espresso-systems-and-offchain-labs-release-r-d-roadmap-for-decentralized-timeboost-5d0007dff66d`.

[35] Espresso Systems, *HotShot/docs/espresso-sequencer-paper.pdf at main · EspressoSystems/HotShot — github.com*, 2023. [Online]. Available: `https://github.com/EspressoSystems/HotShot/blob/main/docs/espresso-sequencer-paper.pdf`.

[36] G. Ethereum, 2022. [Online]. Available: `https://github.com/ethereum/go-ethereum`.

[37] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Comput. Surv.*, vol. 54, no. 6, 126:1–126:36, 2022.

[38] M. Fitzi and J. A. Garay, "Efficient player-optimal protocols for strong and differential consensus," in *PODC*, ACM, 2003, pp. 211–220.

[39] F. Gai, A. Farahbakhsh, J. Niu, C. Feng, I. Beschastnikh, and H. Duan, "Dissecting the performance of chained-bft," in *ICDCS*, IEEE, 2021, pp. 595–606.

[40] W. V. Gehrlein, "Condorcet's paradox," *Theory and Decision*, vol. 15, no. 2, pp. 161–197, 1983.

[41] R. Gelashvili, A. Spiegelman, Z. Xiang, *et al.*, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *PPoPP*, ACM, 2023, pp. 232–244.

[42] G. Golan-Gueta, I. Abraham, S. Grossman, *et al.*, "SBFT: A scalable and decentralized trust infrastructure," in *DSN*, IEEE, 2019, pp. 568–580.

[43] L. Heimbach, L. Kiffer, C. F. Torres, and R. Wattenhofer, "Ethereum's proposer-builder separation: Promises and realities," in *IMC*, ACM, 2023, pp. 406–420.

[44] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *AFT*, ACM, 2022, pp. 47–60.

[45] C. Ho, D. Dolev, and R. van Renesse, "Making distributed applications robust," in *OPODIS*, ser. Lecture Notes in Computer Science, vol. 4878, Springer, 2007, pp. 232–246.

[46] P. Jain, S. J. Desai, M. Shih, *et al.*, "Opensgx: An open platform for SGX research," in *NDSS*, The Internet Society, 2016.

[47] M. Kelkar, S. Deb, and S. Kannan, "Order-fair consensus in the permissionless setting," in *APKC@AsiaCCS*, ACM, 2022, pp. 3–14.

[48] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," *IACR Cryptol. ePrint Arch.*, p. 1465, 2021.

[49] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," in *CCS*, ACM, 2023, pp. 475–489.

[50] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *CRYPTO (3)*, ser. Lecture Notes in Computer Science, vol. 12172, Springer, 2020, pp. 451–480.

[51] R. Khalil, A. Gervais, and G. Felley, "TEX - A securely scalable trustless exchange," *IACR Cryptol. ePrint Arch.*, p. 265, 2019.

[52] A. Kiayias, N. Leonardos, and Y. Shen, "Ordering transactions with bounded unfairness: Definitions, complexity and constructions," *IACR Cryptol. ePrint Arch.*, p. 1253, 2023.

[53] A. Kiayias, H. Zhou, and V. Zikas, "Fair and robust multi-party computation using a global transaction ledger," in *EUROCRYPT (2)*, ser. Lecture Notes in Computer Science, vol. 9666, Springer, 2016, pp. 705–734.

[54] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *AFT*, ACM, 2020, pp. 25–36.

[55] L2BEAT, *Value locked: Show rollups only*, 2023. [Online]. Available: `https://l2beat.com/scaling/summary`.

[56] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[57] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[58] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys*, ACM, 2020, 38:1–38:16.

[59] R. Li, X. Hu, Q. Wang, S. Duan, and Q. Wang, "Transaction fairness in blockchains, revisited," *IACR Cryptol. ePrint Arch.*, p. 1034, 2023.

[60] *Libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity*. [Online]. Available: `https://github.com/hot-stuff/libhotstuff`.

[61] LibraBFT Team, *State machine replication in the Libra blockchain*, Technical report, 2020. [Online]. Available: `https://developers.libra.org/docs/state-machine-replication-paper`.

[62] G. Losa, E. Gafni, and D. Mazières, "Stellar consensus by instantiation," in *DISC*, ser. LIPIcs, vol. 146, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 27:1–27:15.

[63] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication," in *CCS*, ACM, 2019, pp. 887–903.

[64] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *PODC*, ACM, 2020, pp. 129–138.

[65] A. D. Luzio, A. Mei, and J. Stefa, "Consensus robustness and transaction de-anonymization in the ripple currency exchange system," in *ICDCS*, IEEE Computer Society, 2017, pp. 140–150.

[66] D. Malkhi and P. Szalachowski, "Maximal extractable value (MEV) protection on a DAG," in *Tokenomics*, ser. OASIcs, vol. 110, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 6:1–6:17.

[67] A. Mamageishvili, M. Kelkar, J. C. Schlegel, and E. W. Felten, "Buying time: Latency racing vs. bidding for transaction ordering," in *AFT*, ser. LIPIcs, vol. 282, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 23:1–23:22.

[68] L. Mauri, S. Cimato, and E. Damiani, "A formal approach for the analysis of the XRP ledger consensus protocol," in *ICISSP*, SCITEPRESS, 2020, pp. 52–63.

[69] P. Moreno-Sanchez, N. Modi, R. Songhela, A. Kate, and S. Fahmy, "Mind your credit: Assessing the health of the ripple credit network," in *WWW*, ACM, 2018, pp. 329–338.

[70] MystenLabs, *Sui/doc/paper/sui.pdf at main · MystenLabs/sui — github.com*, 2023. [Online]. Available: `https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf`.

[71] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Whitepaper, 2009. [Online]. Available: `http://bitcoin.org/bitcoin.pdf`.

[72] O. Network, *Oasis Web3 MEV Protection*. [Online]. Available: `https://oasisprotocol.org/blog/web3-mev-protection`.

[73] O. Network, *The Oasis Blockchain Platform*. [Online]. Available: `https://oasisprotocol.org`.

[74] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel SGX," *CoRR*, vol. abs/2006.13598, 2020.

[75] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[76] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, 130:1–130:36, 2019.

[77] *Proposer-builder separation roadmap*, Aug. 2023. [Online]. Available: `https://ethereum.org/nl/roadmap/pbs/`.

[78] *Random ordering of equally-priced transactions incentivises competitive spam*, 2023. [Online]. Available: `https://github.com/ethereum/go-ethereum/issues/21350`.

[79] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in *CCS*, ACM, 1994, pp. 68–80.

[80] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 986–1009, 1994.

[81] Ripple Labs, *Ripple 1.4.0*. [Online]. Available: `https://github.com/ripple/rippled/releases/tag/1.4.0`.

[82] Ripple Labs, *XRP Ledger Documentation > Concepts > Consensus Network > Consensus*, 2020. [Online]. Available: `https://xrpl.org/consensus.html`.

[83] Ripple Labs, *XRP Ledger Documentation > Concepts > Introduction > XRP Ledger Overview*. [Online]. Available: `https://xrpl.org/xrp-ledger-overview.html`.

[84] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," 1996.

[85] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[86] D. Schwartz, N. Youngs, and A. Britto, *The Ripple protocol consensus algorithm*, 2014. [Online]. Available: `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.

[87] V. Tumas, B. B. F. Pontiveros, C. F. Torres, and R. State, "A ripple for change: Analysis of frontrunning in the XRP ledger," in *ICBC*, IEEE, 2023, pp. 1–9.

[88] M. A. Vafadar and M. Khabbazian, "Condorcet attack against fair transaction ordering," in *AFT*, ser. LIPIcs, vol. 282, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:21.

[89] S. Wadhwa, L. Zanolini, F. D'Amato, A. Asgaonkar, F. Zhang, and K. Nayak, "Breaking the chains of rationality: Understanding the limitations to and obtaining order policy enforcement," *IACR Cryptol. ePrint Arch.*, p. 868, 2023.

[90] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *PODC*, ACM, 2019, pp. 347–356.

[91] Y. Zhang, S. T. V. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *OSDI*, USENIX Association, 2020, pp. 633–649.

[92] W. Zheng, Y. Wu, X. Wu, *et al.*, "A survey of intel SGX and its applications," *Frontiers Comput. Sci.*, vol. 15, no. 3, p. 153 808, 2021.

[93] L. Zhou, X. Xiong, J. Ernstberger, *et al.*, "Sok: Decentralized finance (defi) attacks," in *SP*, IEEE, 2023, pp. 2444–2461.

Declaration of consent

on the basis of Article 18 of the PromR Phil.-nat. 19

Name/First Name: Milojević Jovana

Registration Number: 17-106-402

Study program: Computer Science

Bachelor ☐ Master ☐ Dissertation ☑

Title of the thesis: Security and Fairness of Blockchain Consensus Protocols

Supervisor: Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of September 5th, 1996 and Article 69 of the University Statute of June 7th, 2011 is authorized to revoke the doctoral degree awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with theses submitted by others.

Bern, 22.01.2024

Place/Date

Signature