ORDERED CONSENSUS WITH EQUAL OPPORTUNITY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yunhao Zhang

May 2024

ORDERED CONSENSUS WITH EQUAL OPPORTUNITY

Yunhao Zhang, Ph.D.

Cornell University 2024

The specific total order of commands agreed upon when running state machine replication (SMR) is immaterial to fault tolerance: all that is required is for all correct nodes to follow the same order. In SMR-based blockchains, however, correct nodes should be responsible for being unbiased when choosing the total order, and for preventing malicious parties from manipulating the order to their advantage. The traditional specification of SMR correctness, however, has no language to express such responsibilities. This dissertation thus extends SMR as *ordered consensus* by introducing the language of ordering preferences and relevant features, which lead to the following contributions.

With ordering preferences, the two notions of Byzantine democracy and Byzantine oligarchy are specified, which capture the minimum and maximum degree of power that Byzantine (*e.g.,* malicious) nodes would have over the order. We prove that the minimum degree of Byzantine power is in general inevitable, but the maximum degree, a Byzantine oligarchy, can be avoided. To remove Byzantine oligarchies, this dissertation introduces Pompē, an SMR protocol that is guaranteed to order commands in a way analogous to linearizability. The evaluation shows that Pompē can achieve competitive performance compared to state-of-the-art SMR protocols in which Byzantine nodes dictate the total order.

With relevant features, two principles are specified capturing the notion of *equal opportunity*, *i.e.,* how correct nodes should treat clients equally instead of

being biased toward certain clients. These principles are inspired by social sciences and law, leading to the notion of a point system. In a point system, system designers specify a set of relevant features and a function that maps a vector of feature values into a score. The total order is thus decided by scores and ties are broken randomly. To achieve equal opportunity, this dissertation introduces a *secret random oracle* (SRO), a system component that generates random numbers in a fault-tolerant manner, and Pompē-SRO, an extension of Pompē that is guaranteed to order commands in a way analogous to a point system. The evaluation shows that Pompē-SRO could mitigate real-world concerns about ordering in blockchains, including front-running and sandwich attacks.

BIOGRAPHICAL SKETCH

Yunhao Zhang was born in Tianjin, China. Tianjin is a city famous for its food and Xiangsheng, a traditional performing art. In addition to enjoying food and Xiangsheng, Yunhao spent eight years studying algorithms and programming under the training of Wei Teng, a teacher from Nankai High School in Tianjin. Yunhao feels that he owes his career to Wei. [1]

Yunhao then left Tianjin and spent four years in Shanghai for his bachelor's degree at Shanghai Jiao Tong University. While keeping interests in theory and algorithms, his focus switched to computer systems and engineering under the influence of professors Binyu Zang and Yubin Xia. In his junior year, he started to do systems research with professors Rong Chen and Haibo Chen. Under the guidance of these four professors, Yunhao gained solid skills and published his first research paper at SOSP'17.

Yunhao first met professors Lorenzo Alvisi and Robbert van Renesse when they visited Jiao Tong in the summer of 2016. Yunhao was deeply attracted by the research possibilities combining theory and systems and thus followed them to Cornell University in 2017 as a PhD student. At Cornell, his research on BFT and consensus has been guided by Professor Lorenzo Alvisi while his teaching on operating systems, especially the work of EGOS-2000 [2], has been guided by Professor Robbert van Renesse.

[1] Quoting Martinus Veltman, Dutch physicist and Nobel laureate, *I have found out that many physicists owe their career to a good high school teacher*.

[2] EGOS-2000 is a teaching project envisioning a future where every student can read all the code of an operating system. See `https://github.com/yhzhang0128/egos-2000`

To my parents, who planted the seed of knowledge in my mind.

ACKNOWLEDGEMENTS

Most importantly, I am deeply grateful to my advisor Lorenzo Alvisi, from whom I gained a passion of exploring distributed computing principles and learned methodologies for the explorations. Lorenzo's passion for this field and rigorous attitude toward scientific research will always make him a role model in my life. Moreover, Lorenzo's encouragement played a fundamental role in my PhD endeavor. In the summer of 2018, Lorenzo recognized my good performance in various theory courses and encouraged me to explore the connection between distributed computing and game theory. This suggestion laid the foundation of my PhD exploration: my research has been inspired by fields in the economics literature that are closely related to game theory, as listed in the related work chapter. Lorenzo granted me a remarkable degree of freedom to explore the related fields and distill useful concepts. Without such freedom, this dissertation would not have been possible. Beyond my academic life, Lorenzo's encouragement has been important to my personal life as well. Lorenzo was always a great and caring listener. I sincerely appreciate the company that Lorenzo has offered over these years and, in return, I have also strived to be a good TA, student, and website hacker, among other roles.

This dissertation would also be impossible without the lessons I learned from Joe Halpern and Xingzhong Yu, two minor advisors on my committee. Joe taught me decision theory and social choice theory: social choice is the key inspiration for the first half of this dissertation and decision theory provides the mathematical definitions for the second half. In my A exam, Joe also suggested that randomness could play an important role in my research direction, which turned out to be very insightful. In addition to technical details, Joe taught me an important philosophy: a simple concept (*e.g.*, expected utility) may have

several underlying axioms and each axiom can be scrutinized empirically with real-world observations. This philosophy has heavily influenced the treatment of point system in this dissertation where related real-world observations are made in the legal context. The choice of the legal context was influenced by Professor Yu from the Cornell Law School. Specifically, I learned from Professor Yu that many legal problems in the Internet and technology context follow the same set of legal principles (*e.g.*, rights) as in traditional contexts like torts or traditional copyright. Following this philosophy, this dissertation attempts to connect the principles of equal opportunity between the context of blockchains and traditional legal contexts such as employment.

Robbert van Renesse is not on my committee, but I have always regarded him as a respectful mentor and advisor. Robbert started his Harmony project as a hobby which eventually evolved into an important part of the operating systems teaching curriculum. Robbert also started the EGOS project back in 2018, leading to several wonderful teaching and research projects for Cornell students. Over the years, I have been deeply influenced by his spirit of pursuing hobbies and turning hobbies into fun for students. I extended Robbert's EGOS project into the EGOS-2000 project, trying to enrich and spread the educational values of EGOS beyond Cornell. I have a feeling that EGOS-2000 would leave a big Cornell mark in the systems field in addition to the research written in this dissertation, which would not have been possible without Robbert.

Lidong Zhou, Srinath Setty, and Qi Chen from Microsoft Research have all provided invaluable support to my PhD research throughout the past five years. Lidong taught me the important philosophy of how to make a theory minimal: introducing a concept only when it is absolutely necessary. This philosophy has served as key guidance throughout this dissertation as well as my work on

EGOS-2000, contributing much to the elegance of these works. Around 2020, Lidong was among the first ones who saw a connection between distributed consensus and social choice theory. Srinath further provided great theoretical insights and he is the leading author of the first set of proofs in this dissertation. Qi, my third collaborator from Microsoft, provided much-needed assistance in the implementation and evaluation chapters. It has been my great pleasure to collaborate with these Microsoft friends.

In addition to research, student service (or department service) was also an important part of my PhD journey. I served as the president of the Computer Science Graduate Organization (CSGO) in the year of 2019-2020. During that difficult year, I worked with student representatives (Varsha Kishore and Oliver Richardson, among others) on several tough situations, including the start of the COVID-19 pandemic. Kavita Bala as the chair and Bobby Kleinberg as the DGS offered great help during that period, and I learned a lot from both of them about how to handle various political situations. I have also remembered Bobby's praise for me – "a beacon of calm and civility amidst the chaos" since I find his praise to be a great spiritual guidance.

Haobin Ni, Soumya Basu, Makis Arsenis, Mahimna Kelkar, and Shir Cohen have all contributed to wonderful discussions that led to this dissertation. It has been my great honor to talk with them, exchange opinions, and obtain their wonderful insights. I am further grateful to many other friends and I apologize for a very incomplete list: Burcu Canakci, Chunzhi Su, Cong Ding, Drew Zagieboylo, Ethan Cecchetti, Florian Suri-Payer, Kai Mast, Kushal Babel, Mae Milano, Matthew Burke, Natacha Crooks, Sowmya Dharanipragada, Xinwen Wang, Yan Ji, Yunhe Liu, Youer Pu, and Zhen Sun.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This dissertation aims at adding a new dimension to state machine replication (SMR) [101], a fundamental building block in fault-tolerant distributed computing, by introducing a way to express, reason about, and enforce specific properties about how the SMR protocol *orders* client requests.

1.1 Order, the problem of modeling blockchains with SMR

SMR coordinates a set of replicas of a deterministic service so that, collectively, they implement the abstraction of a single, correct server. The deterministic service exposes as its API a set of deterministic *commands* that can be invoked by corresponding client requests. In particular, the protocol sequences client requests to produce a total order, which correct replicas follow when processing the requests by executing the corresponding commands. As long as the system includes sufficiently many correct replicas, voting on replica outputs guarantees that clients of the replicated service recognize and accept only output values that would have been generated by a correct server.

SMR totally orders client requests by running an instance of consensus for each position in the request sequence. The only requirement on this sequence is that it eventually contains all requests from correct clients. Indeed, if all SMR is used for is fault tolerance, nothing further is necessary: the specific sequence of states that correct replicas traverse is immaterial.

Not so, however, when SMR is used (typically, in a Byzantine fault tolerant (BFT) configuration [43, 72, 77]) across multiple administrative domains

to support a blockchain. Consider permissioned blockchains like Diem [11], CCF [98], or HyperLedger Fabric [30]: the specific order of transactions held by their ledger can have significant financial implications [52, 80, 95, 107].

The nodes running these protocols are not just in charge of converging on a ledger: they have a real stake in the specific sequence that the ledger records. They are responsible for treating clients fairly and for preventing malicious parties from manipulating the sequencing to their advantage.

The traditional specification of correctness for (BFT) SMR, however, has no language for expressing such responsibilities. Because it attaches no significance to the sequence it produces, it is intrinsically incapable of characterizing what makes a total order "right" or "wrong".

The aim of this dissertation is to introduce a framework for expressing and enforcing such distinctions. A key challenge is that the lack of expressiveness in the specification of SMR has seeped deeply into the traditional architecture used to implement SMR. Specifically, in standard leader-based SMR [43, 75], the ordering of commands is hard coded in the protocol that adds the commands to the ledger: the leader runs concurrently a set of consensus instances, each dedicated to filling a specific ledger position with a command.

Thus, we pursue a two-pronged approach: toward greater expressiveness, we extend the correctness specification of the SMR primitive; toward effective enforcement, we propose a new architecture and a new system component that, together, make it possible to implement the new correctness requirements in practice. These contributions are instantiated in Pompē-SRO, an ordered consensus protocol guaranteed to order commands in an equitable way.

1.2 Contributions

This dissertation makes five contributions. First, it introduces a new primitive, *ordered consensus*, that makes it possible to express precise correctness conditions about the ordering of commands (Chapter 2.2.1). Second, given this new primitive, it delineates the extent to which order can be shielded from Byzantine influence (Chapter 2.2.2). Third, it motivates and introduces *equal opportunity* as a way to connect the ordered consensus primitive with real-world fairness concerns of systemic bias (Chapter 3.2). Fourth, it introduces a new architecture and a new system component which, together, make it possible to enforce the newly introduced correctness conditions (Chapter 2.3.1 and Chapter 3.3.1). The new architecture separates an ordering phase that decides the ordering of commands from a consensus phase that drives consensus on ever-growing prefixes of the total order decided by the ordering phase. The system component, a *secret random oracle (SRO)* that generates random numbers in a fault-tolerant manner, proves valuable in pursuing equal opportunity. Lastly, it shows the design, implementation, and evaluation of Pompē and Pompē-SRO, two ordered consensus protocols (Chapter 2.3 and Chapter 3.3). The evaluation shows that Pompē-SRO can not only enforce ordering properties but also reduce the impact of malicious manipulation of ordering in real-world blockchains.

Ordered consensus, a new primitive. In the traditional SMR specification, clients invoke commands and replicas execute them. SMR uses a consensus protocol to coordinate the replicas and consensus protocols enforce safety and liveness. Safety requires (*i*) all correct replicas to execute commands in the same total order and (*ii*) all executed commands to have been invoked by some client.

Liveness requires all commands invoked by correct clients to be executed by all correct replicas eventually.

Safety and liveness specify no additional requirement on the ultimate total order of commands. To make it possible to express such requirements, we introduce the language of *ordering indicators*: each replica associates an ordering indicator with each command, which is used to express how this replica would like commands to be ordered with respect to one another. Removing, or at least limiting, the influence of faulty nodes over the total order is clearly desirable. If it were possible to fully remove the influence of faulty nodes, the total order would be decided solely by the preferences of all correct nodes. However, as our next contribution shows, this is in general impossible to achieve.

Drawing the boundaries of Byzantine influence. Inspired by social choice theory [31, 33, 37], our second contribution characterizes the boundaries within which Byzantine nodes can influence the outcome of ordered consensus (*e.g.*, the total order held in a blockchain's ledger). A clear goal of designing ordered consensus protocols is to prevent the establishment of a *Byzantine oligarchy*, a condition (rigorously defined in Chapter 2.2.2) under which Byzantine nodes manage to fully dictate the ledger's total order. Indeed, a natural goal would be to aim for protocols that are completely immune from Byzantine influence. However, we prove that if *correct* nodes are given a chance to influence the total order through their ordering preferences, this natural goal cannot be achieved. Instead, ordered consensus protocols necessarily operate in a *Byzantine democracy* (a notion formally characterized in Chapter 2.2.2 as well), where, in *some* executions, Byzantine nodes are able to determine the relative order of *some* commands in the output ledger. These findings point to a clear direction for

limiting Byzantine influence: identifying conditions under which the ordering preferences of correct nodes *alone* can, with certainty, determine the relative order of commands in the ledger. In Chapter 2.2.3, we introduce one such notion, *ordering linearizability*.

Identifying and limiting systemic bias. Even if Byzantine influence were to be completely eliminated, a deeper issue would remain. It is not enough to base the ledger's ordering on the preferences of correct nodes: it is essential to ensure that the mechanisms used to express, collect, and process those preferences are not affected by some *bias* that systematically favors (even in the preferences of *correct* nodes) some parties over others.

For example, if replicas use timestamps to express their ordering preferences, it is undesirable for commands issued by clients in closer proximity to the replicas to have a systematic edge over those from clients who issued their commands earlier in real-time, but happened to be more remote. Indeed, as we discuss in Chapter 3.1, such systemic bias is not just undesirable but can lead to unsavory or even illegal financial maneuvers such as front-running [52, 80, 107].

We draw on the insights of social sciences to approach this problem. Social sciences have taught us that, whenever ranking is involved, it is important to explicitly identify the set of *relevant features* on which a legitimate ranking can be based on and to avoid *irrelevant features* from playing a role. For example, US law explicitly identifies features (protected classes) that must be deemed irrelevant by US employers and lending agencies [1, 2].

We argue that, in blockchains, the process of expressing ordering preferences should be similarly based on a well-specified and publicly-known set of rele-

vant features, while trying to minimize the influence of irrelevant features. In this context, a relevant feature might be the time at which a client issued its command, while the client's geo-location should be, ideally, irrelevant. We take some initial steps towards realizing this framework in Chapter 3.2, which we then concretely instantiate in the Pompē-SRO protocol in Chapter 3.3. Further steps are sketched in Chapter 5.

A new SMR architecture. We introduce a new SMR architecture for limiting Byzantine influence together with a new system component that can be used to limit systemic bias.

To avoid a Byzantine oligarchy, ordering is factored out of consensus via a separate ordering phase, which collects and aggregates the replicas' ordering preferences to ultimately produce a total order of client requests. A consensus phase then adds commands to the ledger while following the order decided in the ordering phase. When using a leader-based protocol in the consensus phase, this separation prevents the leader from unilaterally determining a command's position in the ledger; at the same time, it retains the efficiency of having a leader in charge of driving consensus.

We propose a framework for expressing and identifying systemic bias in Chapter 3.2. We suspect systemic bias when two commands have identical relevant features (*i.e.,* in our context, they are invoked at the same time), and yet, because of irrelevant features (*e.g.,* the issuer's geo-location) it is much more likely for one of them to be ordered first in the ledger. In the absence of bias, such commands should have *equal* chances of being ordered first in the ledger, as if their order had been decided randomly. Chapter 3.3.3 shows formally how

random numbers drawn from a uniform distribution can be used to limit systemic bias. However, Byzantine nodes can take actions, such as tampering with network behaviors or expressing ordering preferences, that change the distribution from which a specific random number is drawn. To discourage these changes, every random number should be kept secret before Byzantine nodes have taken all possible actions that may influence how this particular random number is used in the system. To this end, we introduce the *secret random oracle (SRO)*, a system component that generates random numbers in a fault-tolerant manner. An SRO generates random numbers and keeps them confidential until other components finish producing their output.

Pompē and Pompē-SRO. Finally, we design, implement, and evaluate Pompē and Pompē-SRO, two ordered consensus protocols that enforce ordering properties in addition to traditional safety and liveness properties. Pompē enforces ordering linearizability (defined in Chapter 2.2.3) while Pompē-SRO enforces ϵ-Ordering equality and Δ-Ordering linearizability (defined in Chapter 3.2.3).

We implement two versions of Pompē that borrow their consensus phase from two state-of-the-art BFT SMR systems [7, 8]. Our evaluation demonstrates that, while Pompē incurs higher latencies than its baselines, it achieves higher throughput at competitive latencies by batching commands in both the ordering phase and the consensus phase. For example, with $n = 4$ nodes in a single data center, a version of Pompē that extends ordering linearizability to HotStuff [8, 110] achieves a throughput of approximately 360,000 commands/s at a latency of 53 ms, which corresponds to 40% higher throughput and 6% higher latency than HotStuff. In addition, since nodes in Pompē can order multiple commands in parallel, we find that, if the computing resources assigned to each

node are scaled up proportionally with the number of nodes, Pompē can then sustain its high throughput in a setup with 100 nodes distributed over three geo-distributed datacenters.

However, our evaluation also shows that in a setup similar to the one in which the Ethereum blockchain operates, the total order produced by Pompē can be significantly biased by geo-location, an irrelevant feature. To redress such bias, we design, implement, and evaluate Pompē-SRO, which extends Pompē by adding a secret random oracle. Pompē-SRO uses the randomness produced by the SRO to significantly reduce the impact of the geo-location of the client proposing a command over that command's ultimate placement in the ledger. Ordering linearizability is redefined based on the invocation time of a request instead of the potentially biased ordering preferences. Removing the impact of irrelevant features from the definitions does incur a price. Specifically, Pompē-SRO needs a greater separation between the invocation time of two requests as the condition for removing Byzantine influence.

CHAPTER 2

FROM OLIGARCHY TO DEMOCRACY

2.1 Background and motivation

The increasing popularity of blockchains as a platform for cooperation and data sharing among mutually distrustful parties has brought about a renewed interest in Byzantine fault tolerance (BFT). In particular, permissioned blockchains, which promise a platform for executing commands without trusting a centralized authority, have adopted as their core the standard BFT SMR architecture [101]. Transitioning BFT to this new application domain has introduced some new challenges. One that has received much attention is scalability. Traditional BFT SMR protocols have typically targeted deployments involving a number of nodes in the single digits, while some permissioned blockchains envision running BFT at scales that are two orders of magnitude or larger. A new breed of BFT SMR protocols has raised to this challenge, finding clever ways to pipeline requests and streamline the communication required to achieve consensus [36, 59, 83, 88, 110]. This dissertation addresses a different challenge that emerges when applying BFT SMR in a blockchain context, one fundamental enough to bring into question whether this primitive is sufficiently expressive to serve as the basis for this new class of applications.

Consider the correctness specification of SMR: it requires all correct nodes replicating a service to traverse the same set of states and produce the same outputs. If replicas are deterministic, an expedient way to satisfy this requirement is to ensure that all correct replicas process the same sequence of inputs: identical inputs translate into identical states and outputs. As long as these inputs are

valid client commands, the correctness specification assigns no semantic meaning to the particular order in which they are executed by the replicas: that order is simply a syntactic mechanism used to achieve the desired safety property.

In blockchains, however, the specific order adopted by the underlying SMR protocol tends to have rich semantic implications, which often translate into substantially different financial rewards for the parties involved. Allowing some users to front-run their commands ahead of others clearly gives them an unfair advantage in applications such as auctions and exchanges [80, 92, 107]; indeed, a recent paper [52] details how bots have reaped from unsuspecting parties profits in excess of $6M by replicating, within the Ethereum network, transaction manipulation strategies already notorious in traditional exchanges [80]. Yet, order manipulation (including censorship, selective inclusion, command reordering, and command injection) does not, per se, violate the specifications of SMR, the technology at the core of projects like Diem [11].

Unfortunately, adding the "BFT" qualifier to SMR does not help address these concerns: all it does is to ensure that the standard SMR specification continues to hold even if some nodes are Byzantine. The crux, rather, is that the correctness expectations of blockchain users do not stop at requiring all ledgers to hold the same total order: *which order* matters.

A symptom of the discomfort caused by this semantic gap is the growing focus on curbing the discretion of the single node that, in Paxos-like BFT SMR, leads the consensus decisions: if Byzantine, this *leader* node can single-handedly control the ledger's order. Proposed solutions include rotating leaders [39, 48, 110]; holding leaders accountable for their actions [61, 64]; or developing outright "leaderless" protocols that give no node a special role in the

execution of consensus [50, 76, 88]. These efforts are a step in the right direction, but they also, arguably, miss the point. While it is clear enough that leaving a single leader in full control of the ledger's order is undesirable, they fiddle with a low-level mechanism without offering a way to express the correctness guarantees that such mechanisms, whatever they may be, should enforce. Recent work on order-fairness [69], concurrent with ours, takes a further step forward by adding to consistency and liveness the additional requirement of *transactional order-fairness*; however, it offers neither a general framework for synthesizing desirable ordering guarantees from the ordering preferences of individual nodes, nor tries to precisely characterize the minimum and maximum degree to which Byzantine nodes can affect ordering.

This dissertation argues that the correct approach to bridge the current semantic gap is instead to start from first principles. Thus, we introduce a new primitive, *Byzantine ordered consensus*, that expands the correctness specifications of BFT SMR so it can express specific correctness guarantees about the total orders it produces. Inspired by classic work in social choice theory [31, 33, 37], Byzantine ordered consensus lets participating nodes not only propose commands but also indicate how they prefer to see them ordered. Essentially, Byzantine ordered consensus makes it possible to specify which total orders a correct BFT SMR is allowed to produce, given the nodes' ordering preferences. For example, assuming that nodes use as their ordering preference the time they first see a command, we show that it is possible to require total orders that satisfy a natural generalization of linearizability: *ordering linearizability*, which ensures that, if the highest timestamp from all correct nodes for command c_1 is lower than the lowest timestamp from all correct nodes for c_2, c_1 is ordered before c_2 in the output.

The design space for ordering properties that we explore is delimited by two overarching concerns. On the one hand, we want to curb as much as possible the clout of Byzantine nodes; on the other hand, we want to ensure that the preferences of correct nodes will carry weight in the final ordering.

These goals can sometimes align; in particular, when it comes to preventing Byzantine nodes from solely controlling the ledger's final ordering. As we noted above, the standard approach to BFT SMR allows a Byzantine leader to alone dictate which command commits in which consensus instance, independent of what other nodes prefer. We aim for, and define, guarantees (such as ordering linearizability) that prevent such Byzantine dictatorships. Indeed, we show that it is possible to rule out a Byzantine oligarchy, in which Byzantine nodes are jointly able to determine the ordering decisions, regardless of the correct nodes' ordering preferences.

Sometimes, however, we find these goals fundamentally at odds with one another: in particular, we find that ensuring that each correct node has a saying in the final order makes it impossible, in general, to completely prevent Byzantine nodes from influencing the final order. This is the price, if you wish, of operating in a Byzantine democracy.

2.2 Ordered consensus

Ordered consensus generalizes SMR to expose the ordering aspect explicitly, but preserves the same system model, which consists of a distributed system of n nodes (with at most f Byzantine faults) that act as clients as well as servers: they both propose commands and execute them. This model simplifies our pre-

sentation without any loss of generality; we discuss how it relates to different real-world deployment scenarios in Chapter 2.3.5.

2.2.1 Ordering indicators

Ordering indicators. As in standard BFT SMR, nodes in Byzantine ordered consensus propose commands as inputs and output a consistent totally-ordered ledger. Unlike standard BFT SMR, each node associates a proposed command c with an *ordering indicator* o, which is metadata indicating the node's ordering preference for c, so proposals are of the form $\langle o, c \rangle$. Let O denote the set of ordering indicators; we define an *order-before relation* $\prec_o$ on $O \times O$ as follows: For any pair of proposals $\langle o_1, c_1 \rangle$ and $\langle o_2, c_2 \rangle$, where $o_1, o_2 \in O$, $o_1 \prec_o o_2$ indicates a preference to order c_1 before c_2.

Examples of ordering indicators include timestamps, sequence numbers, and dependency sets or graphs. For timestamps (or sequence numbers), the order-before relation $\prec_o$ can simply be the $<$ relation on timestamps (or sequence numbers), while for dependency sets/graphs, $\prec_o$ can be the subset/subgraph relation on dependency sets or graphs.

Profiles, executions, and traces. We refer to a set of $\langle o, c \rangle$ proposals as a *profile*. Let $\mathcal{P}^i$ and $\mathcal{P}^C$ denote, respectively, the set of proposals from node i and the set of proposals from all correct nodes. Given a command c, we say that $c \in \mathcal{P}^C$ if and only if there exists a correct node i and an ordering indicator o, such that $\langle o, c \rangle \in \mathcal{P}^i$. In an *execution*, correct nodes follow their prescribed protocol and input their proposals from $\mathcal{P}^C$, whereas Byzantine nodes and the network are

under the control of an adversary. For a given profile, an execution can produce different *traces*; each trace captures a single deterministic run of the protocol, recording the behavior of all nodes (both correct and Byzantine) as well as of the adversarial network. Although all traces of an execution take as input the same $\mathcal{P}^{\mathcal{C}}$, the content of the ledger on which correct nodes agree may be different for different traces, because of the actions of Byzantine nodes or the behavior of the network. But what is the degree to which Byzantine nodes can exert their influence on a given protocol? And what is the price to curb it?

2.2.2 A general impossibility

A minimum guarantee that any protocol should offer is to make it impossible for Byzantine nodes to dictate the ordering of the ledger's entries. It is out of concern for ensuring this guarantee that recent work in BFT SMR has focused on limiting the leader node's discretion in making ordering decision. The formal structure offered by Byzantine ordered consensus allows us to move past the inadequacies of the current mechanisms used to drive consensus and focus instead on a precise characterization of what any such mechanism should guarantee. In particular, we capture the intuition that Byzantine nodes can dictate the ordering decisions through the notion of Byzantine oligarchy. This notion is inspired by the Arrow's impossibility theorem [31] and the Gibbard–Satterthwaite impossibility theorem [57, 99] from social choice theory.

Byzantine Oligarchy. A protocol execution is subject to a *Byzantine oligarchy* if and only if, for all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, for all pairs of commands c_1 and c_2 in $\mathcal{P}^{\mathcal{C}}$, there exists a trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_1 before c_2 in the ledgers of

correct nodes and another trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_2 before c_1 in the ledgers of correct nodes.

Intuitively, this definition conveys that, in a Byzantine oligarchy, the actions of Byzantine nodes can determine the ordering of any two commands c_1 and c_2, regardless of the ordering indicators from correct nodes.

Can we do more, and completely eliminate any influence of Byzantine nodes over the ledger's final ordering? The framework offered by Byzantine ordered consensus allows us to prove that this target can be achieved only at the price of denying *correct* nodes a voice in the ordering decision. The intuition is simple: since in general it is impossible to distinguish a priori between correct and Byzantine nodes, a policy that enfranchises the first group necessarily also gives some influence to the second.

To formalize this intuition, we introduce two new notions. First, we express what it means for a protocol to allow the ordering preferences of its nodes to influence the ledgers' final total order. Note that, if a node can influence the outcome, then there will be some circumstances in which the node's preferences will actually *determine* the outcome. The second notion we introduce characterizes the impact of according such influence to a Byzantine node.

Free Will. We say that a protocol respects the nodes' *free will* if and only if (i) for all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, there exists a trace for $\mathcal{P}^{\mathcal{C}}$, such that all commands in $\mathcal{P}^{\mathcal{C}}$ appear in the ledgers of correct nodes in the trace and (ii) there exist two profiles $\mathcal{P}_A$ and $\mathcal{P}_B$, such that, for all commands c_1 and c_2 that appear in both profiles, there exists a trace for $\mathcal{P}_A$ that results in c_1 before c_2 in the ledgers of correct nodes and there exists a trace for $\mathcal{P}_B$ that results in c_2 before c_1 in the

ledgers of correct nodes.

Free will rules out (i) arbitrarily denying proposed commands and (ii) trivial and predetermined ordering mechanisms (*e.g.,* ordering commands by their hash values) .

Byzantine Democracy. We say that a protocol upholds *Byzantine democracy* if and only if there exists a profile of correct nodes $\mathcal{P}^C$, such that for all pairs of commands c_1 and c_2 in $\mathcal{P}^C$, there exists a trace for $\mathcal{P}^C$ that results in c_1 before c_2 in the ledgers of correct nodes and another trace for $\mathcal{P}^C$ that results in c_2 before c_1 in the ledgers of correct nodes.

Unlike a Byzantine oligarchy, a Byzantine democracy gives Byzantine nodes sway over the final ledger only for *some* profiles of correct nodes, rather than all of them. Precisely, Byzantine democracy uses the existential quantification over profiles while Byzantine oligarchy uses the universal quantification.

We are now ready to formulate a theorem that places fundamental limits to the degree to which it is possible to curb the influence of Byzantine nodes.

Theorem 2.2.1. *Free will* $\implies$ *Byzantine democracy.*

Proof. Consider the following $n + 1$ profiles, where $\mathcal{P}_{\#1} = \mathcal{P}_A$ and $\mathcal{P}_{\#n+1} = \mathcal{P}_B$ and every node proposes the same commands (though, possibly, with different ordering preferences) in $\mathcal{P}_A$ and $\mathcal{P}_B$.

$$
\begin{aligned}
\mathcal{P}_A = \mathcal{P}_{\#1} &= \mathcal{P}_A^1 \cup \mathcal{P}_A^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_{\#2} &= \mathcal{P}_B^1 \cup \mathcal{P}_A^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_{\#3} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
&\ldots \\
\mathcal{P}_{\#n} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_B^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_B = \mathcal{P}_{\#n+1} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_B^{n-1} \cup \mathcal{P}_B^n
\end{aligned}
$$

In profile $\mathcal{P}_{\#i}$, the proposals of the first $i-1$ nodes are the same as in $\mathcal{P}_B$; those of the other $n-i+1$ nodes are the same as in $\mathcal{P}_A$. Because free will (condition (ii)) holds, there is a trace for $\mathcal{P}_{\#1}$ for which the ledgers of correct nodes order c_1 before c_2, and a trace for $\mathcal{P}_{\#n+1}$ where instead they appear in the opposite order. If $P_A = P_B$, then Byzantine democracy holds vacuously.

In the case of $P_A \neq P_B$, because free will (condition (i)) holds, for each index k, there exists a trace for $\mathcal{P}_{\#k}$, such that c_1 and c_2 appear in the final ledgers. Then, there must exist some index i for which the relative order of c_1 and c_2 switches when going from $\mathcal{P}_{\#i}$ to $\mathcal{P}_{\#i+1}$. Consider the the smallest such i. $\mathcal{P}_{\#i}$ and $\mathcal{P}_{\#i+1}$ only differ in what node i proposes: in $\mathcal{P}_{\#i}$ node i's proposals come from $\mathcal{P}_A$; in $\mathcal{P}_{\#i+1}$, they come from $\mathcal{P}_B$. Hence, by choosing whether to $\mathcal{P}_A^i$ or $\mathcal{P}_B^i$, node i determines the relative order of c_1 and c_2. If i is Byzantine, then Byzantine democracy holds for the following correct profile:

$$
\mathcal{P}^C = \mathcal{P}_B^1 \cup \ldots \cup \mathcal{P}_B^{i-1} \cup \mathcal{P}_A^{i+1} \ldots \cup \mathcal{P}_A^n \qquad \square
$$

The definition of Byzantine democracy makes clear that there exist some pro-

files that allow Byzantine nodes to control ordering decisions. A natural question then is: can we design protocols that, by construction, enforce guarantees that specify profiles on which Byzantine nodes cannot have influence? And what would such properties look like? We address the second question next, leaving the answer to the first to Chapter 2.3.

2.2.3 Two natural ordering properties

Since under standard BFT assumptions (Chapter 2.3) correct nodes are more than two thirds of the total (a supermajority!), the profiles less likely to be influenced by Byzantine nodes are intuitively those in which the ordering preferences of correct nodes are aligned. We examine two natural ordering properties that one might want to see holding in such profiles; other definitions are possible. The first requires that, if the ordering indicators of correct nodes are unanimous on how to relatively order two commands, their preference should be reflected in the final ledger.

Ordering unanimity. For all profiles of correct nodes $\mathcal{P}^C$, for all commands c_1 and c_2 that appear in $\mathcal{P}^C$ and in the ledgers of correct nodes, if, for every correct node i, $\langle o_1, c_1 \rangle \in \mathcal{P}^i \wedge \langle o_2, c_2 \rangle \in \mathcal{P}^i \Rightarrow o_1 \prec_o o_2$, and there exists at least one correct node j, such that $\langle o_1, c_1 \rangle \in \mathcal{P}^j \wedge \langle o_2, c_2 \rangle \in \mathcal{P}^j$ holds, then c_1 must precede c_2 in the ledgers of correct nodes.

The second ordering property is inspired by linearizability [65], which orders a command c_1 before a command c_2 if the first ends before the second starts.

Ordering linearizability. For all profiles of correct nodes $\mathcal{P}^C$, for all com-

mands c_1 and c_2 in $\mathcal{P}^C$ and in the ledgers of correct nodes, let $O_1 = \{o_1 | \langle o_1, c_1 \rangle \in \mathcal{P}^C\}$ and $O_2 = \{o_2 | \langle o_2, c_2 \rangle \in \mathcal{P}^C\}$, if $o_1 \prec_o o_2$ holds for all $o_1 \in O_1$ and $o_2 \in O_2$, then c_1 must precede c_2 in the ledgers of correct nodes.

Informally, the "lowest" and "highest" ordering indicators in O_1 (or O_2) indicate when c_1 (or c_2) start and end in the collective perception of correct nodes. Hence, by analogy with linearizability, if all ordering indicators in O_1 are lower than those in O_2, then c_1 is to be ordered before c_2.

Unfortunately, even when correct nodes are unanimous, their wishes are not guaranteed to come true. The issue again arises from the tension between the desire of giving a voice to every correct node and the inability to distinguish a priori between correct and Byzantine nodes.

Theorem 2.2.2. *No protocol can uphold both free will and ordering unanimity.*

Proof (sketch). Consider the four-node profile ($f = 1$) in Figure 2.1. It is an example of what classic social choice theory calls a Condorcet cycle [37, 49]: for any two commands c_i and c_{i+1} (modulo 4) , three nodes prefer the first before the second; the fourth begs to differ.

$$\begin{aligned}
\mathcal{P}^1 &= \{\langle 1, c_1 \rangle, \langle 2, c_2 \rangle, \langle 3, c_3 \rangle, \langle 4, c_4 \rangle\} \\
\mathcal{P}^2 &= \{\langle 1, c_2 \rangle, \langle 2, c_3 \rangle, \langle 3, c_4 \rangle, \langle 4, c_1 \rangle\} \\
\mathcal{P}^3 &= \{\langle 1, c_3 \rangle, \langle 2, c_4 \rangle, \langle 3, c_1 \rangle, \langle 4, c_2 \rangle\} \\
\mathcal{P}^4 &= \{\langle 1, c_4 \rangle, \langle 2, c_1 \rangle, \langle 3, c_2 \rangle, \langle 4, c_3 \rangle\}
\end{aligned}$$

Figure 2.1: A Condorcet cycle

Since any single node may be Byzantine, the requirement of ordering unanimity applies to all ordering preferences endorsed by at least three nodes—but in this example they form a *cycle*, and thus cannot be all satisfied. □

Like ordering unanimity, ordering linearizability also promises to respect the collective preferences of correct nodes; fortunately, unlike the former property, it *is* achievable. What allows ordering linearizability to escape the Condorcet cycle trap is a simple insight: it expresses ordering preferences in terms of real-time *happened before*, a relation that is inherently acyclical. Indeed, as we show next, it is not only achievable, but can be efficiently implemented.

2.3 Pompē

Pompē is a new protocol explicitly designed for ordered consensus. It preserves the same interface as a standard BFT protocol: clients invoke commands and correct nodes reach consensus on a sequence of committed commands. In addition to satisfying the standard safety and liveness properties of BFT SMR, Pompē introduces an ordering phase for ordered consensus and prevents Byzantine oligarchies by enforcing ordering linearizability.

2.3.1 Architecture and protocol design

A new architecture. Pompē's two-phase architecture is designed to mirror the decoupling of ordering from consensus made possible by the framework of ordered consensus. First, an *ordering phase* decides the total ordering of commands, "locking" the relative position among the commands invoked in this phase in a way that Byzantine nodes cannot alter; then, a *consensus phase* allows all correct nodes to agree on a stable prefix of the final sequence, following the total ordering decisions in the ordering phase, and to record it in the

ledger. We refer to commands in the ledgers of correct nodes as *stable commands*. Note that, since the total order of commands that have completed the ordering phase cannot be changed during the consensus phase, it is again safe to put a single leader node in charge of finalizing consensus. Thus, Pompē can retain the performance benefits of leader-based BFT SMR without fears of enabling a Byzantine oligarchy.

System model. As in prior works in the BFT SMR literature, we consider a distributed system with a set of $n = 3f + 1$ nodes, where up to f nodes can be *Byzantine* (*i.e.,* deviate arbitrarily from their prescribed protocol) and the rest are *correct*. We assume the existence of standard cryptographic primitives (unforgeable digital signatures and collision-resistant hash functions) and that cryptographic hardness assumptions necessary to realize these primitives hold. Furthermore, we assume that each node holds a private key to digitally sign messages, and that each node knows the public keys of other nodes in the system. We consider an adversarial network that can drop, reorder, or delay messages. However, for liveness properties, we assume that the network satisfies a weak form of synchrony [43, 54, 55]. Finally, we assume that each node has access to a timer, which produces monotonically increasing timestamps each time the timer is queried.

Protocol description We now describe how Pompē instantiates each of the phases in our new architecture. Throughout the protocol, we assume that correct recipients of messages that are not well-formed (*e.g.,* because they carry an incorrect signature) will drop them: we omit these actions for brevity.

(1) Ordering phase. Pompē uses timestamps as ordering indicators. To "lock" a position for a command in a total order, Pompē proceeds in two steps.

In the first step, a node N_i with a command c collects signed timestamps on c from a quorum of $2f + 1$ nodes. The median timestamp in the set of $2f + 1$ signed timestamps is the *assigned timestamp* for c, and it determines the position of c in the total order. Because there are at most f Byzantine nodes, by picking the median value, the assigned timestamp is both upper- and lower-bounded by timestamps from correct nodes. This is the key observation that allows the protocol to achieve ordering linearizability.

To lock this position in the total order for c, in the second step N_i broadcasts c along with its assigned timestamp and waits for it to be accepted by a quorum of $2f + 1$ nodes (we explain below what it means for a command to be accepted). If a command c is accepted by a quorum of $2f + 1$ nodes, c is not only guaranteed to be included in the totally-ordered ledgers of correct nodes, but also that its position in the ledgers is determined by the assigned timestamp of c. We refer to such commands as *sequenced*.

Local state. Each node maintains the following local data structures: (1) localAcceptThresholdTS, an integer, initialized to 0, that tracks what N_j believes to be, currently, the highest possible timestamp of any stable command in the ledger;

(2) localSequencedSet, a set, initially empty, that tracks all commands that the node has accepted; (3) highTS, an n-sized vector of integers where highTS$[i]$, initialized to 0, stores the highest timestamp received from node N_i; and (4) highTSMsgs, an n-sized vector of messages where highTSMsgs$[i]$, initialized to *null*, stores the message signed by node N_i that carried the value currently stored

in highTS[i].

To complete our discussion of each node's local state, we first need to introduce a simple protocol that nodes use to update their timers.

The timer protocol. Let $\mathcal{T}$ be the $(f+1)^{th}$ highest timestamp in highTS. Because at most f nodes are Byzantine, $\mathcal{T}$ is upper-bounded by a timestamp from a correct node. Let each node reset its timer to $\mathcal{T}$ whenever $\mathcal{T}$ is higher than the current value of the local timer. Periodically, each node broadcasts its current value of $\mathcal{T}$ in a Sync message to indicate that all correct nodes can now set their timer to be $\mathcal{T}$ or higher. To prove to its recipients that the $\mathcal{T}$ value is valid, the Sync message also includes the sender's highTSMsgs vector. □

We are now ready to define two additional data structures: (4) globalSyncTS stores the highest $\mathcal{T}$ received so far in a Sync message; and (5) localSyncTS stores instead the node's local timestamp at the time it received that Sync message.

Actions. Each node N_i with a command c executes the following two steps to lock a position for c in a total ordering of commands:

1. N_i broadcasts $\langle \mathsf{RequestTS}, c \rangle_{\sigma_{N_i}}$ and waits for responses from $2f + 1$ nodes, where σ_{N_i} is a signature on the payload using N_i's private key.

    - A node N_j responds with $\langle \mathsf{ResponseTS}, c, ts \rangle_{\sigma_{N_j}}$, where ts is a timestamp from N_j's local timer.

2. N_i broadcasts $\langle \mathsf{Sequence}, c, T \rangle_{\sigma_{N_i}}$, where T is a set of $2f + 1$ responses received in the first step, and waits for responses from $2f + 1$ nodes.

    - A node N_j *accepts* the broadcast message and adds it to its localSequencedSet if the assigned timestamp of c is higher than localAccept-

ThresholdTS. If so, N_j responds with $\langle \text{SequenceResponse}, \text{ack}, h \rangle_{\sigma_{N_j}}$; otherwise, it responds with $\langle \text{SequenceResponse}, \text{nack}, h \rangle_{\sigma_{N_j}}$, where h is the cryptographic hash of the Sequence message.

The second step above is crucial to establishing stable prefixes in the sequence of commands. Intuitively, it requires every correct node N_j to refuse sequencing a command if its assigned timestamp may be lower than that of a stable command.

Note that, during sufficiently long periods of synchrony (which are necessary for liveness), nodes can get their commands sequenced in just two round-trips—a lower latency than recent BFT protocols [45, 110]. However, sequenced commands are not yet suitable for execution until they become stable: only then it is guaranteed that commands with lower timestamps in the output have all been sequenced.

Nodes *can* execute commands speculatively in their localSequencedSet, but they must wait for the consensus phase to finish before externalizing output and be ready to perform selective reexecution if their speculation is incorrect.

(2) Consensus phase. The principal goal of the consensus phase is to ensure that all correct nodes agree that a certain prefix of the total order constructed in the previous phase is stable, meaning that the prefix is forever immutable.

To accomplish this, Pompē employs any standard leader-based BFT SMR protocol (*e.g.,* [43, 59, 110]) that offers a primitive to agree on a value for each slot in a sequence of consensus slots. We generically refer to this protocol as Consensus. For simplicity, we assume that each consensus slot is associated with

non-overlapping time intervals $[ts, ts')$ such that $ts' > ts$, and that for the first consensus slot $ts = 0$. We further assume that the mapping from consensus slot numbers to time intervals is common knowledge. In practice, this can be implemented by making the interval of the first consensus slot as $[0, \tau)$, where τ is the system initialization time, and then assigning each subsequent consensus slot a fixed window of time (*e.g.*, $[ts, ts + 100\,\text{ms})$). Note that this does not mean that nodes must agree on a value during these time intervals.

For liveness, Pompē relies on a bound Δ on the sum of two terms: the maximum difference Δ_1 between the values returned, at any time, by local timers of correct nodes, which in turn depends on the time it takes for a Sync to travel from one node to another and be processed at the recipient; and the maximum time Δ_2 needed by a correct node to execute the ordering phase (we assume that these bounds include additional slack to account for clock drifts across nodes). Pompē's safety properties hold even when Δ does not hold, but, during sufficiently long periods of synchrony, we assume that the bound holds for proving liveness later in this chapter.

Local state. The local state of each node is a totally-ordered ledger which is initially empty.

Actions. Suppose that consensus slot k maps to time interval $[ts, ts')$, meaning that all commands with assigned timestamp in this interval are expected to be included in this slot. If node N_i wishes to serve as a leader in reaching consensus on a value for slot k using Consensus, it proceeds as follows.

1. N_i broadcasts $\langle \text{Collect}, k \rangle_{\sigma_{N_i}}$, and waits for responses from $2f + 1$ nodes.

    - Node N_j waits until two conditions hold. First, the value of N_j's glob-

alSyncTS is higher than ts', meaning that some node sent N_j a Sync message with $\mathcal{T} \geq ts'$. Second, since that Sync message was received, a time interval of at least Δ has elapsed on N_j's timer (*i.e.*, N_j's timer reads at least localSyncTS $+\Delta$). Note that, during sufficiently long periods of synchrony, these delays give all correct nodes enough time to sequence all their commands with assigned timestamps lower than ts' before N_j advances its localAcceptThresholdTS to ts'. In more detail, after Δ_1, all correct nodes should have received and processed a Sync message with $\mathcal{T} \geq ts'$ to set their local timer to be at least $\mathcal{T}$, so after this point, any new command entering the ordering phase will not have an assigned timestamp lower than ts'. After an additional Δ_2, any command with an assigned timestamp lower than ts' must have completed the ordering phase.

- N_j updates its localAcceptThresholdTS $\leftarrow max(ts', \text{localAcceptThresholdTS})$.
- N_j responds with $\langle \text{CollectResponse}, k, \mathcal{S} \rangle_{\sigma_{N_j}}$, where $\mathcal{S}$ is the set of messages in the localSequencedSet of N_j with assigned timestamps in the interval $[ts, ts')$.

2. N_i runs Consensus to agree on value $\mathcal{U}$ for consensus slot k, where $\mathcal{U}$ is the union of CollectResponse messages from $2f + 1$ nodes for consensus slot k.

Constructing a totally-ordered ledger. Once a prefix of consensus slots are agreed upon, nodes can construct a totally-ordered prefix of the ledger by sorting commands in each slot (of the prefix) by their assigned timestamps, breaking ties by their cryptographic hashes. When a node constructs a longer prefix of the ledger, it can execute the commands in the order specified by the ledger.

2.3.2 Proofs of safety, liveness and ordering properties

This chapter proves that Pompē satisfies ordering linearizability in addition to the safety and liveness properties.

Theorem 2.3.1 (Consistency)**.** *For every pair of correct nodes N_i and N_j with local ledgers $\mathcal{L}_i$ and $\mathcal{L}_j$, the following holds: $\mathcal{L}_i[k] = \mathcal{L}_j[k]\ \forall k :: 0 \leq k \leq \min(len(\mathcal{L}_i), len(\mathcal{L}_j))$, where $len(\cdot)$ computes the number of entries in a ledger.*

Proof. By the safety properties of BFT SMR, every pair of correct nodes agrees on the same value for each consensus slot. Furthermore, the transformation from values in consensus slots to a totally-ordered ledger is deterministic. Together, these observations imply the desired result. □

Theorem 2.3.2 (Validity)**.** *If a correct node appends a command c to its local totally-ordered ledger, then at least one node in the system proposed c in the ordering phase.*

Proof. Each command in the ledger of a correct node is constructed from a valid value agreed upon in one of the consensus slots. Furthermore, for a given consensus slot k with assigned time interval $[ts, ts')$, by our construction, a valid value is a set of CollectResponse messages for slot k from at least $2f + 1$ nodes, where each CollectResponse contains commands with timestamps in the interval $[ts, ts')$. Additionally, for a command to have an assigned timestamp, it must have been proposed in the first step of the ordering phase. Together, these observations imply the statement of the theorem. □

Lemma 2.3.1. *The assigned timestamp of a command is bounded by timestamps provided by correct nodes.*

Proof. By assumption, there are at most f Byzantine nodes. Thus, at least $f + 1$ (out of $2f + 1$) timestamps provided in the ordering phase for a given command are from correct nodes. Furthermore, the assigned timestamp of a command discards f lowest and f highest timestamps in the $2f + 1$ ResponseTS messages, thus the assigned timestamp of a command is bounded by timestamps provided by correct nodes. □

Theorem 2.3.3 (Ordering linearizability)**.** *If the highest timestamp provided by any correct node for a command c_1 is lower than the lowest timestamp provided by any correct node for another command c_2 and if both c_1 and c_2 are committed, then c_1 will appear before c_2 in the totally-ordered ledgers constructed by correct nodes.*

Proof. By Lemma 2.3.1, the assigned timestamp of a command is bounded by timestamps provided by correct nodes. As a result of this and the pre-condition in the statement of the theorem, the assigned timestamp of c_1 will be smaller than the assigned timestamp of c_2. Thus, if both c_1 and c_2 are committed, c_1 will appear before c_2 in the totally-ordered ledgers of correct nodes because nodes sort commands by their assigned timestamps. □

Lemma 2.3.2. *During sufficiently long periods of synchrony, a correct node can get its command (along with its assigned timestamp) added to* localSequencedSet *of at least $2f + 1$ nodes.*

Proof (sketch). Suppose a correct node executes the first step of the ordering phase for its command c and obtains an assigned timestamp of ts. During sufficiently long periods of synchrony, by the choice of Δ, a Sequence message that includes c will reach $2f + 1$ correct nodes and be added to their localSequencedSet *before* they advance their localAcceptThresholdTS past ts, which implies the statement of the lemma. □

Lemma 2.3.3. *If a command c with assigned timestamp ts is added to* localSequencedSet *of at least $2f + 1$ nodes, then c will eventually be included in the value committed by a unique consensus slot whose time interval includes ts.*

Proof (sketch). Let k denote the consensus slot whose time interval includes ts. When a leader broadcasts Collect for consensus slot k, the local timers on correct nodes will eventually meet the condition required to send CollectResponse messages. Since c appears in the localSequencedSet of at least $2f + 1$ nodes, and, by assumption, since at most f of them are Byzantine, at least $f + 1$ correct nodes will include c in their CollectResponse for consensus slot k. Denote these $f + 1$ correct nodes with C.

Since Pompē's use of BFT SMR requires proposals that are constructed by taking a union of $2f + 1$ CollectResponse messages, a leader must include at least one message from nodes in C. Thus, c must be included to construct a valid proposal for consensus slot k. These combined with the liveness property of the employed BFT SMR protocol (which ensures that a valid value will eventually be chosen for each consensus slot) implies the desired result. □

Theorem 2.3.4 (Liveness)**.** *During sufficiently long periods of synchrony, a correct node can get an assigned timestamp for its command c such that c will eventually be included in the total order constructed by correct nodes at a position determined by the assigned timestamp of c.*

Proof. During sufficiently long periods of synchrony, by Lemmas 2.3.2 and 2.3.3, c will eventually be included in the value committed by a unique consensus slot whose time interval includes the assigned timestamp of c. Since the algorithm to construct a total ordering of commands from values committed by consensus

slots sorts commands by their assigned timestamps, the position of c is determined by the assigned timestamp of c. □

Byzantine influence in Pompē Pompē greatly diminishes the leverage of Byzantine nodes. Once a command is sequenced, Byzantine nodes can neither censor it nor affect its position in the totally-ordered ledgers of correct nodes. Furthermore, they cannot violate ordering linearizability. Nonetheless, as we saw in Chapter 2.2, in a Byzantine democracy, it is impossible to completely eliminate the influence of Byzantine nodes, and Pompē is not immune from it.

Byzantine democracy in action. Consider the following execution of Pompē, where $n = 4$ and $f = 1$. There are two commands, c_1 and c_2, that in the ordering phase obtained the following timestamps from a quorum of $2f + 1$ nodes.

	N_1	N_2	N_3
c_1	5:00pm	5:03pm	5:03pm
c_2	5:01pm	5:04pm	5:02pm

Assume, without loss of generality, that N_3 is Byzantine, and that the remaining nodes are correct. The timestamps make clear that correct nodes prefer to order c_1 before c_2. However, since the median timestamp of c_1 is higher than the median timestamp of c_2, it is c_2 that will be ordered before c_1. On a positive note, we observe that, in the normal case where the timers on correct nodes are sufficiently synchronized and network delays are small, this window of vulnerability to Byzantine manipulation is small.

Early stopping and deferred selective inclusion. Pompē cannot prevent a Byzantine node from obtaining an assigned timestamp for its command, but not proceeding with the rest of the ordering phase, as this misbehavior is indistinguishable from what may result from a network failure. This ambiguity allows a Byzantine node (possibly with the aid of a Byzantine leader) to decide later, during the consensus phase, whether or not to include its timestamped-but-not-yet-sequenced command in the ledger.

Preventing or reliably detecting this type of misbehavior is impossible, but mechanisms to mitigate the risks and raise suspicion do exist. One possibility is for each node to employ an append-only linear hash chain to record the timestamps it assigns to other nodes' commands. Nodes exchange those hash chains and refer to the corresponding hash value (in the hash chain) in each ResponseTS message. Such hash chains constrain the ability for Byzantine nodes to assign timestamps abnormally (*e.g.,* out of order), and allow after-the-fact auditing (which could be used to expose nodes that routinely timestamp their commands, but do not always sequence those commands). In addition, a correct node N_i can piggyback the tail of a hash chain of all previously timestamped commands of N_j whenever N_j requests a timestamp; this makes it hard for a Byzantine N_j to blame on the network when silently dropping an earlier timestamped command. An alternative mechanism is for correct nodes to hide their commands using a threshold encryption scheme until those commands are totally ordered. This additional step prevents Byzantine nodes from observing the contents of other timestamped commands before deciding whether to drop their timestamped commands.

	base	extensions
Concord [7]	22,141	1122
HotStuff [110]	4,983	900

Figure 2.2: Number of lines of C++ code in Pompē, which we build atop a base BFT library with a set of extensions.

2.3.3 Implementation

We implement two variants of Pompē, where the artifacts differ in the specific BFT protocol they employ for the consensus phase. Specifically, we extend two prior state-of-the-art leader-based BFT protocols: SBFT [59] and HotStuff [110]. SBFT implements a variant of PBFT [43] that includes many optimizations for scalability. HotStuff uses a rotating leader paradigm while incurring low network costs and serves as the foundation of the Diem blockchain [11]. For SBFT, we use its implementation in VMware's Concord [7], and for HotStuff, we use the authors' implementation [8].

Ease of implementation. Implementing Pompē atop an existing consensus protocol involves modest system effort. Figure 2.2 reports the numbers of lines of code we add to the base BFT protocol implementations. These extensions primarily focus on implementing the two steps of the ordering phase in our new architecture. Specifically, we implement four new message types, as described in Chapter 2.3.1. We then implement message handlers to sign and verify timestamps and to manage data structures for localSequencedSet and localAcceptThresholdTS. Additionally, we modify the leader logic so that, for each time interval, a leader starts a consensus phase after assembling a proposal by collecting responses from a quorum of $2f + 1$ nodes, as described in Chapter 2.3.1. The rest

of the consensus protocol is unmodified: the leader of an instance runs the original consensus protocol for a slot with a proposal assembled as described above. Within each slot, commands are ordered by their assigned timestamps.

Optimizations. In Pompē's consensus phase, the CollectResponse message used for consensus slot k contains all commands in a node's localSequencedSet whose assigned timestamp falls within the time interval associated with k. This can lead to large message sizes. However, when the network is synchronous and correct nodes respond in a timely manner, CollectResponse messages will contain the same set of commands. Therefore, we optimize Pompē by having CollectResponse messages sent to the leader carry only a hash of the set of commands in the sender's localSequencedSet. The leader compares the hash of its own localSequencedSet with the hashes carried in the CollectResponse messages received from $2f$ other nodes. If the hashes match, then the leader proceeds to reach consensus for slot k on the commands from its localSequencedSet, using the $2f + 1$ signed hash values (those received from the other nodes as well as its own) as proof that $2f + 1$ nodes reported the same set of commands. Otherwise, the leader requests a new set of CollectResponse messages, this time including the actual set of commands. We enable this optimization by default.

2.3.4 Experimental evaluation

This chapter experimentally evaluates Pompē. We ask two main questions: (1) How does the performance of Pompē compare with that of state-of-the art BFT protocols? (or, what is the price of transitioning from a Byzantine oligarchy to a Byzantine democracy that enforces Byzantine-tolerant ordering guarantees?)

Pompē incurs higher latency than its baselines, but by batching in both phases, Pompē achieves higher throughput at competative latencies.
Pompē's throughput degrades when n increases, but Pompē can scale up each node for higher throughput.
Pompē incurs modest network overheads over its baselines.

Figure 2.3: Summary of evaluation results.

and (2) What is the impact of separating ordering from consensus on end-to-end performance? Figure 2.3 provides a summary of our findings.

We choose as baselines two prior state-of-the-art BFT protocol implementations: Concord [7, 59] and HotStuff [8, 110]. Both are leader-based (and hence subject to Byzantine oligarchy) and hardcode ordering decisions within consensus. As described in Chapter 2.3.3, we implement two variants of Pompē, both upholding ordering linearizability (and hence free of Byzantine oligarchy), by augmenting those two BFT protocols. We refer to Pompē that extends HotStuff as *Pompē-HS*, and to Pompē that extends Concord as *Pompē-C*.

Methodology, testbed, and metrics. We run our experiments on 100 Standard D16s_v3 (16 vcpus, 64 GB memory) VMs on the Azure cloud platform spanning three datacenters, each running Ubuntu Linux 18.04: 34 in Western US, 33 in Southeast Asia, and 33 in Northern Europe. We run single-datacenter experiments using VMs in Western US. We report results only for failure-free executions, as failures do not alter how Pompē performs relative to its baselines.

Our workload is generated by clients that invoke their commands in a closed loop, *i.e.*, they wait to receive a response to their currently outstanding command before invoking the next one. To run experiments with different loads, we vary the number of clients. For HotStuff and Pompē-HS, as in prior work [110],

we run experiments where commands are random, 32-bytes-long values.[1] Similarly, for Concord and Pompē-C, as in prior work [59], we use a benchmark that writes a random value to a randomly-selected key in a key-value store.

Our principal performance metrics are client-perceived latency (measured in ms) and throughput (in commands/second). To measure latency, each client records the latency of each command using its local clock and our scripts aggregate latencies across clients and across commands. For throughput, we compute the total number of commands processed by the system and divide it by the duration of the experiment. To measure the peak throughput of a given system, we increase the number of clients until saturation.

Since Pompē separates ordering from consensus, clients in Pompē receive two responses, one for confirming the relative position of the command in the totally-ordered ledger (when a command is sequenced; see Chapter 2.3.1 for details), and another for the execution result of the command. Therefore, we report two types of latency for Pompē, which we refer to as *ordering latency* and *consensus latency*. Since our baselines hardcode ordering decisions within consensus, both ordering and consensus complete at the same time, so we report a single type of latency for baselines.

End-to-end throughput and latency. We begin by measuring the performance of Pompē and its baselines in a four-node configuration (we report results for larger system sizes later). We run clients on a separate set of virtual machines so that clients and nodes do not contend for resources.

[1]The HotStuff implementation reaches consensus not on actual commands, but on their 32-byte-long cryptographic hashes; clients communicate the actual commands to the replica nodes outside of the consensus protocol.

A note about batching. Batching is a standard technique in SMR protocols to increase throughput at the cost of higher latency by amortizing the cost of running consensus across all the commands in a batch. Both Pompē and its baselines can take advantage of it, and we report experiments for different batch sizes. However, Pompē's separation of ordering from consensus has two significant implications for batching.

First, it eliminates the unintended leverage that Byzantine nodes can gain through batching even in BFT SMR protocols that rotate leaders out of concern for "fairness". The larger the batch, the larger the number of commands whose ordering is left to the unchecked discretion of the current leader: throughput gains thus come at the cost of expanding opportunities for Byzantine oligarchy. Pompē removes these concerns: its ordering guarantee (*e.g.,* ordering linearizability) is unaffected by either the existence of batches or by their sizes.

Second, separating ordering from consensus affects the tradeoff between latency and throughput that comes with batching. When Pompē's baselines do not batch commands, they achieve lower latency *and* lower peak throughput than Pompē. Latency is higher under Pompē because a leader in Pompē must wait for a fixed time window before initiating a proposal; peak throughput is higher because Pompē implicitly batches commands whose timestamps fall within a time window during consensus. However, when the baselines batch commands to match Pompē's latencies, they achieve significantly higher peak throughput than Pompē. Pompē's peak throughput is lower because nodes must produce and validate signed timestamps during the ordering phase, which causes nodes to saturate earlier.

Fortunately, the separation gives Pompē an additional batching opportunity:

	throughput (cmds/s)	median latency (ms)
HotStuff ($\beta_c = 1$)	474	8.2
HotStuff ($\beta_c = 800$)	253,360	49.9
Pompē-HS ($\beta_o = 1$)	1,642	2.3 (o), 47.7 (c)
Pompē-HS ($\beta_o = 200$)	361,687	5.7 (o), 53.1 (c)
Concord ($\beta_c = 1$)	40	53
Concord ($\beta_c = 800$)	6,633	67
Pompē-C ($\beta_o = 1$)	1,415	17 (o), 67 (c)
Pompē-C ($\beta_o = 200$)	249,221	18 (o), 74 (c)

Figure 2.4: Peak throughput and median latency for Pompē and the corresponding baselines in a single datacenter with $n = 4$ nodes. Pompē's leader starts the consensus phase every 50 ms with $\Delta = 10$ ms. Pompē's *ordering latency* is denoted with "o" and its *consensus latency* is denoted with "c".

each node can execute the ordering phase once to assign a single timestamp to an ordered sequence of its own commands (or of commands from clients that belong to the same organization as the node). Such batching does not affect Pompē's ordering properties (*e.g.*, ordering linearizability) because each batch contains commands from a single node. The throughput boost that comes from this additional source of batching can potentially make up for Pompē's lost ground, but raises the question of how to fairly compare the Pompē variants to their baselines.

We balance these different considerations in our experiments as follows: if, in a configuration with n nodes, the baseline's consensus protocol uses a batch size $\beta_c = S\,(> 1)$, then we allow each node in the corresponding variant of Pompē to use batches of size $\beta_o = S/n$ during its ordering phase.

Performance results. Figure 2.4 shows peak throughput and median latency at peak throughput for Pompē and its baselines, for different batch sizes. Since Pompē-C and Pompē-HS perform similarly compared with their respec-

tive baselines, we focus on interpreting the performance of Pompē-HS.

Performance without batching. When $\beta_o = 1$, Pompē-HS's median ordering latency is 28% of the median latency of HotStuff with $\beta_c = 1$, while its peak throughput is about 3.5× higher than HotStuff's. The lower ordering latency is due to Pompē's ordering phase, which incurs only two RTTs compared to the four RTTs required by HotStuff; the higher throughput, perhaps surprisingly given that $\beta_o = 1$, is instead due to batching. In Pompē-HS, setting $\beta_o = 1$ means that nodes do not batch in the *ordering phase*; however, since Pompē-HS does not start consensus until a time window has elapsed, it can still collect commands from multiple clients: for a 50 ms time window, we observed an effective batch size of 82 commands. Unsurprisingly, the flip side of this higher throughput is significantly higher consensus latency. Pompē-HS starts the consensus phase every 50 ms; with $\Delta = 10$ ms, every client waits on average 35 ms for the next consensus phase, ultimately leading to a consensus latency of 47.7 ms.

Performance with batching. We fix the batch size for HotStuff to $\beta_c = 800$ commands, and accordingly set the ordering-phase batch size of each of the four nodes in Pompē-HS to $\beta_o = 200$. Unsurprisingly, the throughput increases significantly for both Pompē-HS and HotStuff, respectively by 220× and 535× over the values we measured for Pompē-HS ($\beta_o = 1$) and HotStuff ($\beta_c = 1$): both systems are CPU-bound, and batching allows them to amortize the cost of cryptographic operations across all commands in a batch. In absolute terms, we find that Pompē-HS achieves 1.4× the throughput of HotStuff; as explained in the previous paragraph, the reason is the additional batching effect due to the 50 ms interval in Pompē-HS that separates successive invocations of consensus.

	throughput (cmds/s)	median latency (ms)
HotStuff ($\beta_c = 800$)	6,160	915.8
Pompē-HS ($\beta_o = 200$)	315,753	259.7 (o), 1518.1 (c)
Concord ($\beta_c = 800$)	1,461	616
Pompē-C ($\beta_o = 200$)	172,774	325 (o), 1415 (c)

Figure 2.5: Peak throughput and median latency for Pompē and baselines with $n = 4$ nodes spanning three geo-distributed datacenters. Batch sizes are the same as the previous experiment in a single datacenter. Pompē's leader starts the consensus phase every 500 ms with $\Delta = 400$ ms.

Performance in a geo-distributed setup. We consider next a geo-distributed setup, where $n = 4$ nodes are deployed in three separate datacenters, with one datacenter running two nodes. We use the same batch size as in the single datacenter setup (*i.e.*, $\beta_c = 800$ for baselines and $\beta_o = 200$ for each node's ordering phase in the corresponding Pompē variants).

Peak throughput. Figure 2.5 shows our results. For HotStuff, geo-replication causes throughput to drop dramatically, to only 2.4% of its value for the same configuration in a single datacenter. For geo-distributed Pompē-HS instead the loss is much more contained: throughput is at 87.3% of its single-datacenter value. Two main factors explain these results. First, as in the single-datacenter case, Pompē-HS can take advantage of effective batching, now with a time interval between successive proposal of 500 ms and $\Delta = 400$ ms; second, HotStuff is hampered by its use of rotating leaders, as a new leader does not propose a new batch until after collecting enough votes for the previous leader's batch: in a geo-distributed setting, this delay can become significant and have a negative effect on throughput.

Latency. Figure 2.6 shows the maximum ordering and consensus latencies

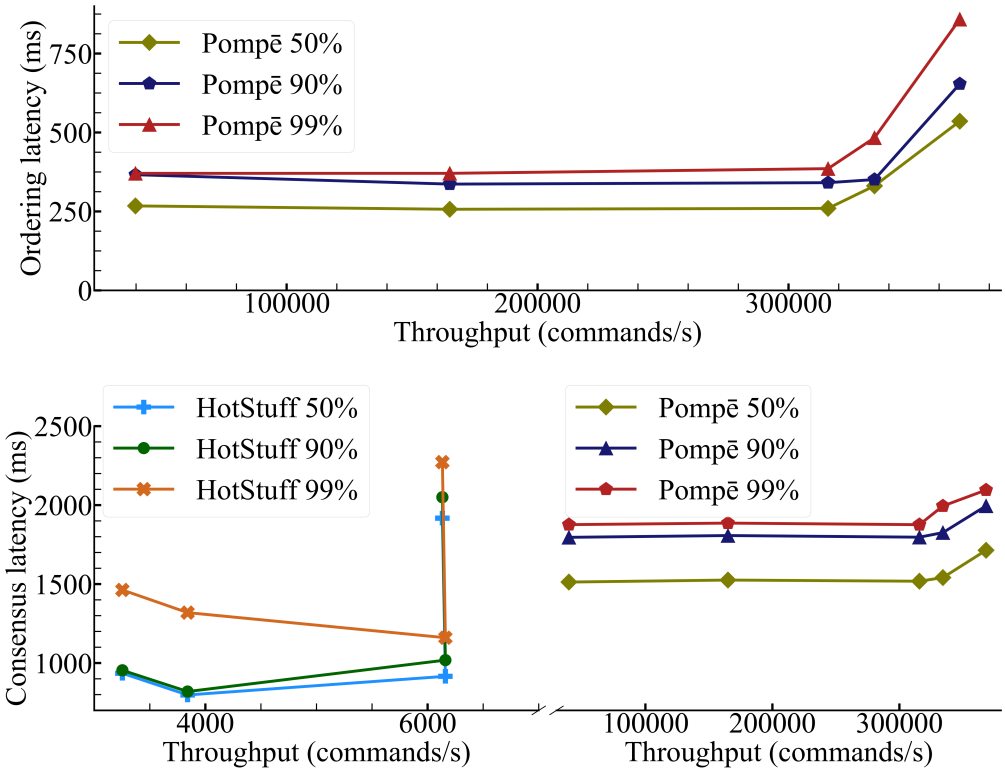

Figure 2.6: Latency vs. throughput for HotStuff and Pompē-HS in a geo-distributed deployment. The two graphs show respectively the maximum ordering latency and consensus latency experienced by different percentiles of the fastest commands. The experimental setup is the same as in Figure 2.5. Pompē-HS achieves higher throughput at the cost of higher consensus latency, but the low ordering latency lets nodes know quickly when their commands are guaranteed to appear in the ledger.

experienced by the fastest 50%, 90%, and 99% of commands. The key take-away is that Pompē-HS achieves higher throughput at the cost of higher consensus latencies. As expected, in Pompē-HS, both types of latency stay stable until system saturation. HotStuff's latency drops at the beginning because, with more clients, it fills up a batch more quickly while also increasing the throughput. Furthermore, the ordering latency is lower than the median consensus latency (since the latter adds more communication rounds to the former) meaning that

nodes can get early notification for when their commands are sequenced and thus guaranteed to appear in the ledger.

Scalability To understand how well Pompē scales to a larger number of nodes, we experiment with increasing values of n. We vary the number of nodes in an experiment from 4 to 100. Our results for Pompē-C (in comparison with Concord) are qualitatively similar to our results for Pompē-HS (in comparison with HotStuff), so we focus on interpreting the performance of Pompē-HS. HotStuff uses the same batch size as before (*i.e.*, $\beta_c = 800$). For Pompē-HS, we experiment with three different configurations.

1. Light: We set $\beta_o = 800/n$ and allocate a single VM to each node.
2. Scale-up: We set $\beta_o = 800/n$ and, as n increases, so does proportionally the number of VMs associated with each node to equal $\lfloor n/4 \rfloor$. So, for example, for $n = 4$, we use one VM per node; but when $n = 10$, each node uses two.
3. Fixed batch: We set $\beta_o = 200$ regardless of n.

Figures 2.7 and 2.8 depict the throughput and latency achieved by Pompē and its baselines for different values of n.

Throughput. HotStuff scales well as n grows, whereas throughput quickly degrades under Pompē-HS (light). This is because batch sizes under Pompē-HS (light) are inversely proportional to n, so throughput degrades as n increases. This is confirmed by the scaling behavior of Pompē-HS (fixed batch) where $\beta_o = 200$ regardless of n. Of course, using a fixed β_o regardless of n may not be a fair comparison as explained in our earlier notes on batching.

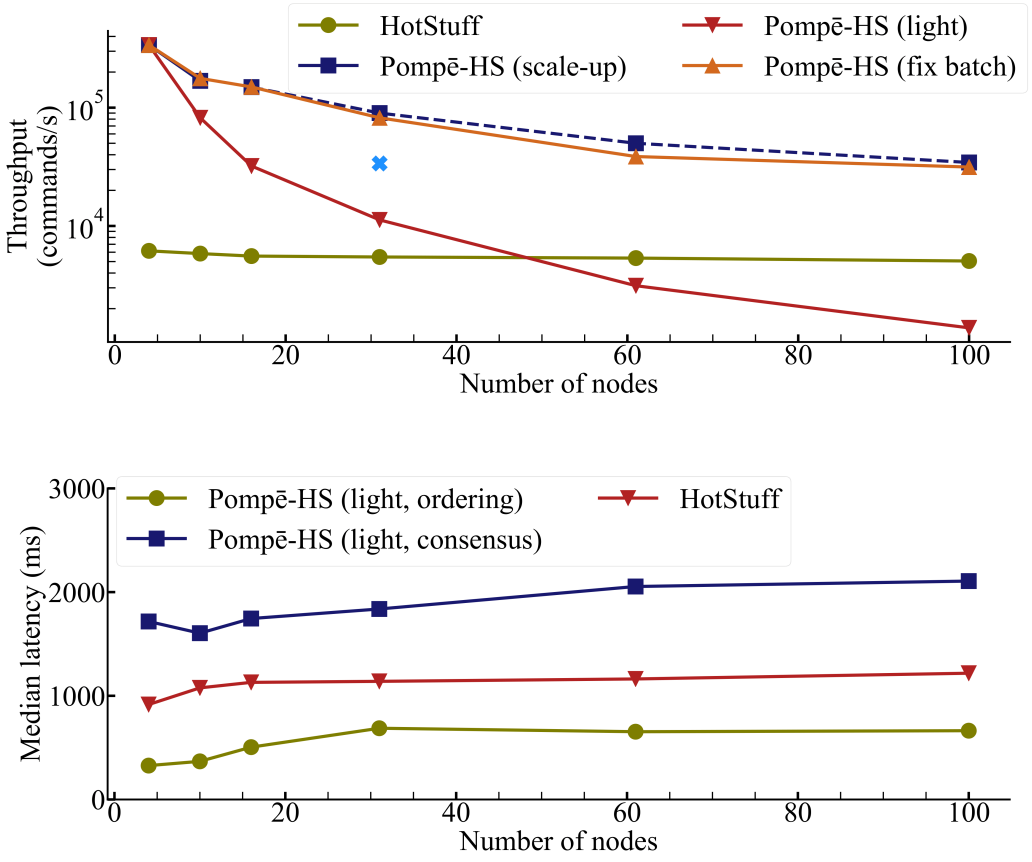

Figure 2.7: Peak throughput and median latency of Pompē-HS in different configurations and of HotStuff, as a function of the number of nodes (n) in a geo-distributed deployment. The light blue cross at $n = 31$ depicts the performance of Pompē-HS (scale up) with 3 VMs per node; the blue square above it shows the *predicted* throughput when each node is assigned $\lfloor 31/4 \rfloor = 7$ VMs. The prediction is based on benchmarks showing that the ordering phase scales near linearly as more VMs are assigned to each node. The blue squares connected by a dotted line at $n = 61$ and $n = 100$ are similarly predicted rather than measured.

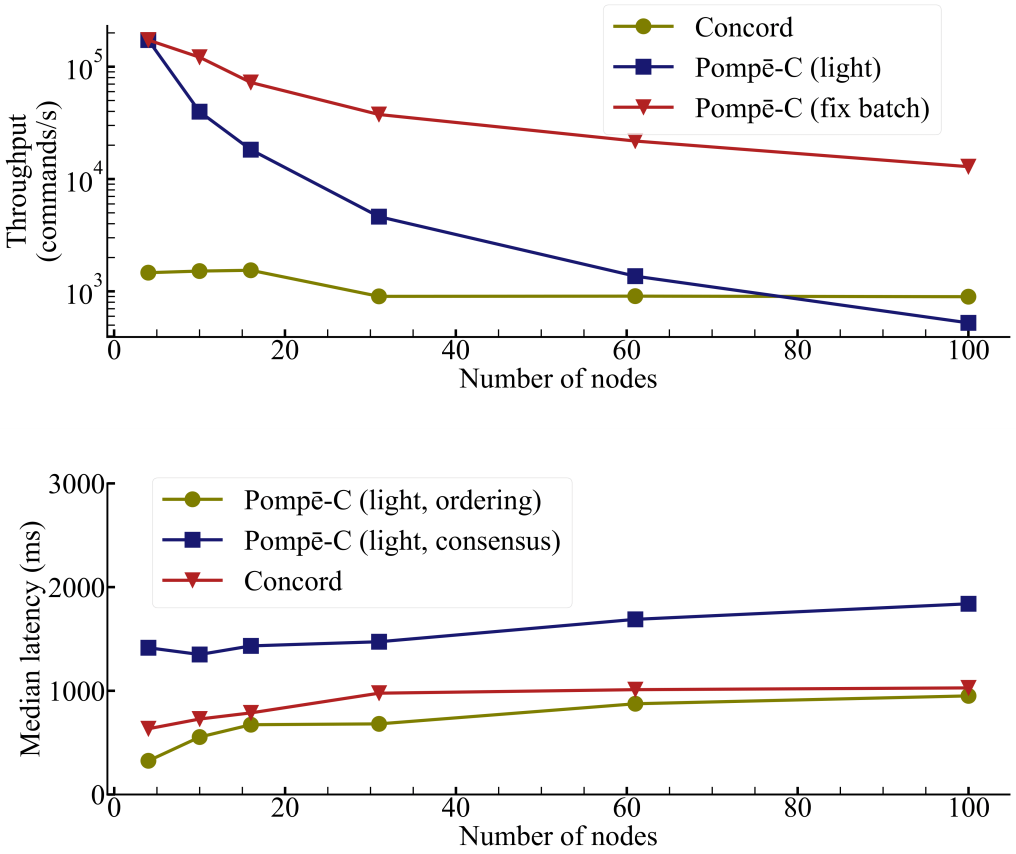

Figure 2.8: Peak throughput and median latency of Pompē-C and Concord with varying number of nodes (n) in a geo-distributed setup. We use $\beta_c = 800$ for the baseline; see the text for the different configurations of Pompē.

Fortunately, we find that Pompē-HS (scale-up) can achieve a behavior similar to Pompē-HS (fixed batch) without having to use a fixed β_o. In Pompē-HS (scale-up), each node uses multiple VMs to run the ordering phase, thereby avoiding the throughput degradation experienced by Pompē-HS (light). Our testbed has 100 nodes, so we could only run Pompē-HS (scale-up) for $n \in \{4, 10, 16\}$. For higher values of n, we predict the throughput of Pompē-HS (scale-up) using experimental results from smaller-scale experiments and ad-

ditional benchmarks which validate that the ordering phase of Pompē achieves a near-linear scaling as each node gets more VMs.

Latency. For both Pompē-HS and HotStuff, latency stays relatively stable when the system scales out. This is because latency is dominated by network communication in a geo-distributed deployment.

Network overhead Compared to its baselines, Pompē incurs higher network costs to attach timestamps with each command when executing the separate ordering phase. To understand the increased network costs, we use $n = 4$ and experiment with both Pompē and its baselines. We experiment with Pompē-HS ($\beta_o = 1$) and HotStuff ($\beta_c = 1$), and record the total number of bytes sent by each node during the experiment. We find that Pompē-HS incurs about 18% higher network costs compared to HotStuff, which, we believe, is a tolerable price for the stronger ordering properties ensured by Pompē.

2.3.5 Discussion

Deployment models. Chapter 2.3.1 describes our protocol in a simplified deployment model centered on nodes, without explicitly mentioning clients, for ease of exposition. This is a reasonable model in the context of our target application, permissioned blockchains, where each node is owned and operated by a separate organization: we can expect clients that belong to an organization to submit their transactions to a node owned by the same organization (so the financial incentives of clients and nodes are aligned). This deployment model also increases the opportunity for batching in the ordering phase at each node

on behalf of all clients in the same organization.

Nevertheless, other deployment models are possible (*e.g.*, those involving clients explicitly without associating them with trusted organizational nodes). Pompē's separation of ordering from consensus makes the following possible: each client executes the ordering phase for its commands by communicating with nodes and then nodes execute the consensus phase. After nodes complete the consensus phase, they send the execution results back to the clients. The protocol does have to account for the revised client/node communication pattern in the calculation of the delay (previously, Δ in the consensus phase) in order to ensure liveness.

Powerful network adversaries. Our network model assumes partial synchrony (as do prior BFT protocols). This does not eliminate a network-level adversary from affecting the assigned timestamps of commands. For example, a powerful adversary that controls the entire network connecting honest nodes can selectively reorder or delay messages among honest nodes to make timestamps assigned to commands *biased*. We will explain the meaning of bias in the next chapter and show how to reduce such bias.

Another commonly adopted network-adversary model [84] assumes that an adversary cannot influence the network connecting correct nodes. In this model, an adversary does not gain additional power in biasing the assigned timestamps beyond what Byzantine nodes could already do. However, without Byzantine nodes, correct nodes could also give biased timestamps leading to real-world fairness concerns, which becomes the starting point of the next chapter.

Command dependencies and replay protection. As in prior BFT protocols, Pompē does not consider dependencies among different commands, nor does it prevent the same command from appearing multiple times in the total order. However, one can embed additional metadata inside commands (*e.g.*, nonces, explicit dependencies, etc.), which correct nodes can use at the time of execution (*i.e.*, after Pompē's consensus phase outputs a total order) to enforce desirable dependencies among commands or to defend against replay attacks.

CHAPTER 3

FROM SYSTEMIC BIAS TO EQUAL OPPORTUNITY

The starting point of this chapter is the observation that, while limiting the influence of Byzantine nodes is a necessary first step towards providing fairness, unfairness can and does arise in practice even when all replicas are correct.

Consider, for example, the practice known in financial markets as *front-running*, where a party, aware of the existence of a large buy order for some stock, places beforehand its own buy order for the same stock. This party is then able to buy low and later sell high, once the stock's value has been driven up by the ensuing large buy order. It has been widely reported how a faster network can enable front-running not just in traditional financial markets [80], but also in decentralized ones [52, 107]. No Byzantine replica is necessary for these attacks to succeed in a blockchain based on SMR: when using timestamps as ordering indicators, the difference in network latency (either due to physical proximity or access to faster network facilities) between clients and replicas may provide some clients with a systemic advantage over others. In this chapter, we call this phenomenon *systemic bias*.

This chapter offers a framework in ordered consensus to reason about and address systemic bias. Taking inspiration from social sciences, which have a long history of reasoning about bias and unfairness, we observe that, whenever ranking is involved, the position of an entry in the ranking depends on the entry's specific characteristics (or *features*). Some of the features are *relevant* to the stated purpose of the ranking, while others may be *irrelevant*. For example, US employers and lending agencies are legally forbidden to consider certain irrelevant features (*protected classes*) in making decisions [1, 2]. Intuitively, a fair

ranking is one that relies only on the entries' relevant features and ignores all the other features. Entries with indistinguishable relevant features should then have an equal chance of being ranked ahead of each other.

Building on the expressiveness offered by the previous chapter, this chapter instantiates this general notion of fairness, *equal opportunity*, in the concrete context of SMR-based blockchains. Since we aim for the ledger to reflect an order respecting the time clients issued requests, we deem the *time of issue* as the relevant feature in determining the ledger's order. Other features, such as geographic location, are considered irrelevant.

Unfortunately, existing protocols for ordered consensus neither distinguish between relevant and irrelevant features, nor reason about such distinctions. As a result, protocols like Pompē, Aequitas [69], and Themis [68] are all vulnerable to parties that leverage irrelevant features, such as faster network facilities, to engage in front-running and in its close relative, *sandwich attacks*, as illustrated in the following chapter.

3.1 Background and motivation

Informally, if two commands have the same invocation time, equal opportunity says that the two possible orders should be equally likely to appear in the system output. Similarly, if three invocations all have the same invocation time, the six possible orders should be equally likely. To show how equal opportunity is often violated in the real world, we analyze publicly available traces of Ethereum. While Ethereum is a permissionless blockchain, none of the issues we identify depend on Ethereum being permissionless.

Case #1: Two invocations. Violating equal opportunity for two invocations may not only indicate bias but provide opportunities for front-running [52, 107]. Empirical studies show that both phenomena have been significant factors in the allocation of $89M over 32 months in the Ethereum blockchain [95].

For example, an invocation from Europe is likely to be ordered before a simultaneous one from Australia because more Ethereum nodes are located in Europe. If the system orders the invocation from Europe earlier in its output more than half the time, we say the system is biased toward Europe. While the geographical location is typically an irrelevant feature in the context of equal opportunity, such bias has been reported in other blockchains as well [85].

Geographical bias can lead to undesirable consequences on blockchain liquidations. In real life, liquidations occur when an individual goes bankrupt. For example, if someone cannot pay their debts but owns a house, a court can sell the house to repay the debts. If the market price of the house is $1.2M, the court may sell it for only $1M. Therefore, many parties would compete for pocketing a $200K profit. Similar liquidations happen on blockchains, where they provide a common way of making a profit in the stable coin [6] and lending[13] applications. The buyer whose command is ordered first on the blockchain is typically the one that realizes the profit.

Consider two clients from Europe and Australia. Suppose they invoke the liquidation command simultaneously, and the system is biased toward Europe, ordering the European command first with a 75% chance. In that case, the expected value of the European client will be $0.2M * 0.75 = $150K, and that will be $0.2M * 0.25 = $50K for the Australian client. In other words, geographical bias could cause very different profits to clients who should be treated equally.

System output	Victim's profit	Attacker's profit
i_2, i_1, i_3	-$500	$800
i_3, i_1, i_2	$700	-$400
other order	$300	$0

Figure 3.1: An example of sandwich attacks. The semantics of the three invocations are explained in Chapter 3.3.5.

Similarly, a client intent on becoming the beneficiary of a liquidation's profits could leverage faster network connections to violate equal opportunity and front-run other clients.

Case #2: Three invocations. In this case, violating equal opportunity among three simultaneous commands could enable sandwich attacks [113]. Empirical studies show that victims of sandwich attacks have lost more than $174M over 32 months in the Ethereum blockchain [95].

Figure 3.1 shows an example of sandwich attacks observed in the decentralized exchange applications. Right after the victim invokes command i_1, the attacker invokes commands i_2 and i_3. The attacker only profits if the order the system outputs is i_2, i_1, i_3. Therefore, the key to sandwich attacks is making i_2, i_1, i_3 a much more likely output than equal opportunity would allow. A common strategy to influence the odds is to privately relay i_2 and i_3 to colluding nodes, which will then exclusively propose blocks containing the sequence i_2, i_1, i_3 [95].

Of course, the attacker is always free to decide which specific commands to invoke as their trading strategy, and different trading strategies may still lead to different expected profits – but, crucially, the system should not allow attackers to tamper with the odds of its different possible outputs: all six permutations of the three commands should be roughly equally likely.

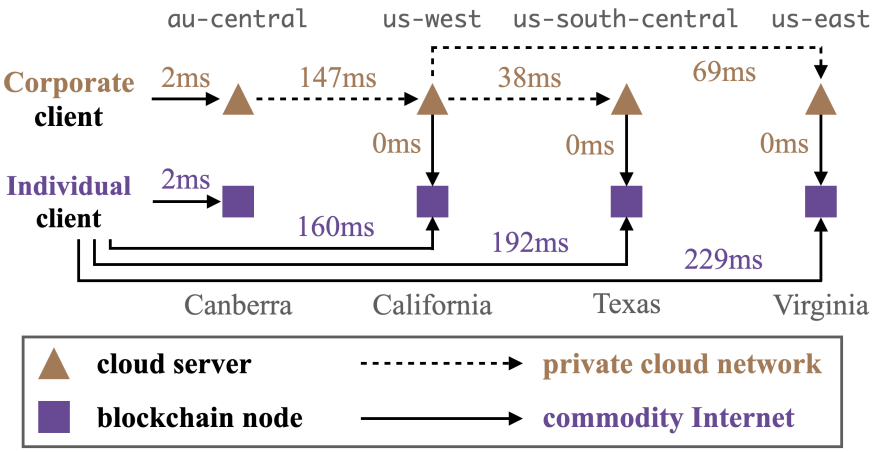

Figure 3.2: Consider a corporate and an individual client both in Canberra. The individual, using the commodity Internet, sends its command to the four nodes in Canberra, California, Texas and Virginia. The corporate client tries to front-run using faster cloud networks and the cloud services are not affordable to the individual. In the cloud, `us-west` is the entry from Australia to the US so that latency$_{\texttt{au-central}\rightarrow\texttt{us-south-central}}$ is 185ms and latency$_{\texttt{au-central}\rightarrow\texttt{us-east}}$ is 216ms. The latency numbers of the cloud are from Microsoft Azure [16] and the numbers of the Internet are from WonderNetwork [20].

Systemic bias in state-of-the-art. Figure 3.2 shows how systemic bias could happen in blockchains. An individual client sends its command to the four blockchain nodes through the Internet while a corporate client, such as a trader within a financial trading company, attempts to front-run the individual with faster and expensive cloud networks. *Front-run* means that the corporate client attempts to have its command ordered earlier by the system although it is sent later than the individual's command.

The state-of-the-art SMR protocols attempt to be "fair" by adopting different fairness concepts, including rotating leadership [110], removing oligarchy [112] and first-come-first-serve [68, 69]. However, as we will show next, they can all be biased to the corporate client even if all the nodes are correct and faithfully follow the given protocol, without any Byzantine behaviors.

Rotating leadership. Suppose a leader node unilaterally decides the order of commands and, to constrain the influence of Byzantine leaders, the system rotates the leader frequently. With 75% chance, the leader would be one of California, Texas or Virginia who receive the corporate client's command first due to its faster network. If nodes order the commands by the receiving order, the system will thus be biased to the corporate client. Rotating leadership has been adopted by influential blockchain projects such as Diem [11, 110].

Removing oligarchy. Many protocols argue against the use of a leader or an oligarchy that unilaterally decides the output order. Pompē, as described in Chapter 2.3, requires a client to collect timestamps from a quorum of nodes and the median of these timestamps is used to order a command. While using such a median timestamp constrains Byzantine influence, there will be nearly *zero chance* for Pompē to order the individual before the corporate client since the median of any quorum must be one of California, Texas and Virginia.

First-come-first-serve. Ideally, commands are ordered by their invocation time (*i.e.*, when a client sends a command), which spawns the idea of first-come-first-serve. However, it seems impossible to measure the invocation time accurately without trusting a centralized party which blockchains try to avoid. Therefore, Aequitas [69] and Themis [68] propose a decentralized version of first-come-first-serve: If a majority of the nodes receive a command first, this command would be ordered first in the output. Unfortunately, this property immediately implies that the system should be biased towards the corporate client in Figure 3.2.

The lesson is that system designers should identify the relevant features (*e.g.*, invocation time) and reason about how violations of equal opportunity (*e.g.*,

bias shown in Figure 3.2) could happen in a system, instead of blindly adhering to normative fairness concepts such as rotating leadership or first-come-first-serve. The same lesson has also been given in social sciences [111] and, inspired by this work, we now specify the principles of equal opportunity.

3.2 Ordered consensus with equal opportunity

The notion of equal opportunity derives from the combination of two well-known principles [111] – *impartiality* and *consistency*. When applied to our settings, impartiality informally requires that the order of commands should not be influenced by irrelevant features, such as clients' geolocation. Consistency instead requires that the invocation of a new command should not cause the relative order of existing commands to change.

We now introduce these notions more formally.

3.2.1 Relevant features

We start by introducing the language of *relevant features* into the framework of ordered consensus described in Chapter 2.2. Consider a system with n nodes in which f are faulty (*i.e.,* Byzantine) and the rest are correct. There are an unbounded number of clients who invoke commands.

Client invocation. An *invocation* is a pair $\langle c, \vec{f_r} \rangle$ where c is a command and $\vec{f_r}$ is a vector of relevant features, *i.e.,* of the only features that should be considered

in determining how a client's commands should be ordered. Typically, features to be ignored include a client's identifier, geolocation, wealth, and network facilities. In blockchains, relevant features typically include invocation time and transaction fee. An *invocation profile*, denoted as $\mathcal{I}$, is a set of invocations.

Node preference. Nodes observe invocations and express preferences. The *preference* of a node is a set of $\langle i, o \rangle$ pairs, where i is an invocation and o is an ordering indicator (*i.e.,* a piece of metadata such as a score or a timestamp). The preference of a node represents how a node would like to order the invocations with respect to one another. A *preference profile*, denoted as $\mathcal{P}$, is a vector of preferences from all correct nodes.

World and chance relation. A *world* is a pair of $\langle \mathcal{I}, \mathcal{P} \rangle$, representing the scenario where clients invoke commands $\mathcal{I}$ and correct nodes express preferences $\mathcal{P}$. For all worlds, $\mathcal{P}$ is well-formed under $\mathcal{I}$, meaning all the invocations in $\mathcal{P}$ should also appear in $\mathcal{I}$.

It is uncertain which world will actually happen. When one admits that nothing is certain, one must also add that some things are more nearly certain than others [97, 62]. We thus introduce *chance relations*: for any two worlds, w_1 and w_2, $w_1 \succ_c w_2$ denotes that w_1 has a higher chance than w_2, and $w_1 \sim_c w_2$ denotes that the two worlds have an equal chance of happening in the system.

$$\mathcal{I} = \{\langle c_1, \langle 5\text{pm}\rangle\rangle_{i_1}, \langle c_2, \langle 5\text{pm}\rangle\rangle_{i_2}, \langle c_3, \langle 5{:}01\text{pm}\rangle\rangle_{i_3}\}$$
$$\mathcal{P}_1 = \langle\{\langle i_1, 1\rangle, \langle i_2, 2\rangle, \langle i_3, 3\rangle\}\rangle$$
$$\mathcal{P}_2 = \langle\{\langle i_2, 1\rangle, \langle i_1, 2\rangle, \langle i_3, 3\rangle\}\rangle$$

Figure 3.3: An example of invocation and preference profiles for a system with one node ($n = 1, f = 0$). The system uses invocation time as the only relevant feature and sequence numbers as ordering indicators. $\mathcal{I}' = \{i_1, i_2\}$ for impartiality because they have the same relevant feature $5pm$. If there is more than one correct node, $\mathcal{P}_2$ could permutate $\mathcal{I}'$ for one or more entries in $\mathcal{P}_1$.

3.2.2 Two general principles

Impartiality. A system is *impartial* if and only if, for all world $\langle \mathcal{I}, \mathcal{P}_1\rangle$, for all $\mathcal{I}' \subseteq \mathcal{I}$, for all $\mathcal{P}_2$ permutating $\mathcal{I}'$ in $\mathcal{P}_1$, if all the invocations in $\mathcal{I}'$ have the same $\vec{f_r}$, then $\langle \mathcal{I}, \mathcal{P}_1\rangle \sim_c \langle \mathcal{I}, \mathcal{P}_2\rangle$.

Impartiality is the first pillar of equal opportunity, and Figure 3.3 shows an example. Since 5pm is the relevant feature of both i_1 and i_2, and $\mathcal{P}_2$ swaps i_1 and i_2 from $\mathcal{P}_1$, impartiality says that the two worlds, $\langle \mathcal{I}, \mathcal{P}_1\rangle$ and $\langle \mathcal{I}, \mathcal{P}_2\rangle$ should be equally likely to happen in the system. In other words, the order of i_1 and i_2 should be based on invocation time and independent of irrelevant features.

Consistency is the second pillar of equal opportunity. Figure 3.4 shows an example. $\langle \mathcal{I}, \mathcal{P}_1\rangle$ and $\langle \mathcal{I}, \mathcal{P}_2\rangle$ are two worlds where c_3 is invoked but never received by the correct nodes so that i_3 is missing from the preferences. Consistency says that $\langle \mathcal{I}, \mathcal{P}_1\rangle \succ_c \langle \mathcal{I}, \mathcal{P}_2\rangle \iff \langle \mathcal{I}, \mathcal{P}_3\rangle \succ_c \langle \mathcal{I}, \mathcal{P}_4\rangle$. In other words, the order of i_1 and i_2 should be based on their *own* features and be independent of the features of i_3, even the invocation time of i_3. Let $\mathcal{P}_1 =_{\mathcal{I}} \mathcal{P}_2$ denote $\mathcal{P}_1$ equals to $\mathcal{P}_2$ over invocations in $\mathcal{I}$ and $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2\}$ denote a partition of $\mathcal{I}$. We now define

$$\mathcal{I} = \{\langle c_1, 1\text{pm}\rangle_{i_1}, \langle c_2, 1{:}01\text{pm}\rangle_{i_2}, \langle c_3, 1{:}05\text{pm}\rangle_{i_3}\}$$
$$\mathcal{P}_1 = \langle\{\langle i_1, 1\rangle, \langle i_2, 2\rangle\}\rangle \quad \mathcal{P}_3 = \langle\{\langle i_1, 1\rangle, \langle i_2, 2\rangle, \langle i_3, 3\rangle\}\rangle$$
$$\mathcal{P}_2 = \langle\{\langle i_1, 2\rangle, \langle i_2, 1\rangle\}\rangle \quad \mathcal{P}_4 = \langle\{\langle i_1, 2\rangle, \langle i_2, 1\rangle, \langle i_3, 3\rangle\}\rangle$$

Figure 3.4: An example of consistency where $\mathcal{I}_1 = \{i_1, i_2\}$ and $\mathcal{I}_2 = \{i_3\}$ is a partition of $\mathcal{I}$. This example actually corresponds to real-world scandals: Some stock exchanges used to introduce special commands like c_3 to illegally help certain trading firms profit by manipulating the order of c_1 and c_2 [80].

consistency formally.

Consistency. A system is *consistent* if and only if, for all worlds $\langle \mathcal{I}, \mathcal{P}_1\rangle$, $\langle \mathcal{I}, \mathcal{P}_2\rangle$, $\langle \mathcal{I}, \mathcal{P}_3\rangle$, $\langle \mathcal{I}, \mathcal{P}_4\rangle$, for all partition $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2\}$, $\mathcal{P}_1 =_{\mathcal{I}_1} \mathcal{P}_3 \ \wedge \ \mathcal{P}_2 =_{\mathcal{I}_1} \mathcal{P}_4 \ \wedge \ \mathcal{P}_1 =_{\mathcal{I}_2} \mathcal{P}_2 \ \wedge \ \mathcal{P}_3 =_{\mathcal{I}_2} \mathcal{P}_4$ implies $\langle \mathcal{I}, \mathcal{P}_1\rangle \succ_c \langle \mathcal{I}, \mathcal{P}_2\rangle \iff \langle \mathcal{I}, \mathcal{P}_3\rangle \succ_c \langle \mathcal{I}, \mathcal{P}_4\rangle$.

A straightforward approach to achieving both impartiality and consistency is to establish a *point system*. In a point system, the system designer decides a formula mapping each invocation to a *score*, and this score only depends on the relevant features of the invocation. The output of a point system orders all the invocations by their scores and breaks ties by uniformly sampling a permutation. Impartiality is guaranteed because the score assigned to each invocation only depends on its relevant features, so the same relevant features lead to the same scores. Consistency is guaranteed because the score of each invocation depends on its own features and is thus independent of the other invocations. Indeed, it has been proved that, when it comes to ranking, a point system is the only mechanism that can satisfy both impartiality and consistency [111].

3.2.3 Point system and its ordering properties

In distributed systems, invocation time is typically the only relevant feature: commands are ordered according to *when* they are invoked. Equal opportunity can be achieved with a point system directly using the invocation time as the score. Unfortunately, in practice, the invocation time of commands cannot be measured accurately. The invocation time can only be approximately measured by the time nodes observe the invocation. Therefore, such a measurement reflects the invocation time and also irrelevant features such as geolocation. To accommodate such measurement inaccuracy in distributed systems, we relax a point system with two parameters, ϵ and Δ, and define two properties, ϵ-*Ordering equality* and Δ-*Ordering linearizability*.

ϵ-Ordering equality. For all invocation profile $\mathcal{I}$ and subset $\mathcal{I}' \subseteq \mathcal{I}$, for all two total orders of $\mathcal{I}'$ denoted as $\succ_1$ and $\succ_2$, if all the invocations in $\mathcal{I}'$ have the same invocation time as their relevant feature, then $|Pr[\succ_1] - Pr[\succ_2]| \leq \epsilon(|\mathcal{I}'|)$.

In this definition, ϵ is a function with the cardinality of $\mathcal{I}'$ as its parameter, and $Pr[\succ]$ denotes the probability of $\succ$ appearing in the system output under the condition that $\mathcal{I}$ is invoked. Different systems could enforce this property with different ϵ functions. Intuitively, by making ϵ approach 0, clients with the same relevant features would have similar chances, reflecting the tie-breaking part of a point system. We now define another property that reflects how a point system deals with different scores.

Δ-Ordering linearizability. For all invocation profile $\mathcal{I}$ and two invocations $i_1, i_2 \in \mathcal{I}$, if i_1 is invoked more than Δ time earlier than i_2, then i_1 will appear

before i_2 in the output.

Ideally, Δ can be 0 so that commands invoked earlier will always appear earlier in the output. Consider a naive setup in which all the nodes are correct, and they measure the invocation time accurately. The ideal could then be achieved, as stated by the following theorem.

Theorem 3.2.1. *If all nodes are correct and accurately measure the invocation time of all invocations, ordering equality and ordering linearizability with $\epsilon = \Delta = 0$ can be achieved using a point system.*

Proof. By properties of a point system. □

3.3 Secret Random Oracle (SRO)

The point system mechanism suggests that randomness can be crucial to ordering equality. We thus specify a system component, *secret random oracle* (SRO), that generates random numbers in a fault-tolerant manner and keeps the random numbers secret until other components have finished producing their outputs. By keeping those numbers secret, the outputs of other system components cannot depend on them.

Design overview. Consider two clients who both invoke a command at time t. Say the first client uses faster network facilities for front-running, and the two commands are received by all correct nodes before $t + 200ms$ and $t + 400ms$, respectively. If two timestamps are chosen from $[t,\ t + 200ms]$ and $[t,\ t + 400ms]$ for ordering, the system could sample two random numbers independently and

```
Reveal(Int k, Set<Signature> s) → Int | Error
Generate(Int k) → Proof
Verify(Int k, Proof p, Int r) → Bool
```

Figure 3.5: The interface of a secret random oracle (SRO).

add to the timestamps, so the probability of the two commands being ordered one way or the other is close. This *random noise* affects the ϵ of ordering equality and the Δ of ordering linearizability. As the intensity of the random noise approaches infinity, ϵ will approach 0 because ordering is then dominated by randomness, but Δ will unfortunately approach infinity. We will study such a trade-off between ϵ and Δ quantitatively.

3.3.1 Interface and guarantees

Consider a set of n nodes, at most f of which can behave arbitrarily. Every node holds a private key and knows the public keys of all other nodes. They provide a secret random oracle service with an interface shown in Figure 3.5.

A random function (not shown) maps an integer k to a pseudorandom number. `Reveal` is invoked to reveal the random number after all nodes in a quorum demonstrate, by providing signatures of k, that they wish to reveal the random number. `Generate` takes integer `k` and returns a cryptographic proof. Given a proof, `Verify` verifies whether the random number returned by `Reveal` is correct. An SRO provides the following guarantees:

Uniqueness. `Reveal` maps every integer `k` to a random number. Multiple queries of `Reveal` with the same `k` and any valid set of signatures return the

same random number. A set of signatures is valid if it contains valid signatures of `k` from at least $n - f$ different nodes. `Reveal` returns an error when given an invalid set of signatures.

Secrecy. For all integers `k`, if `r` is the unique integer that `Reveal` would return and the adversary does not have valid signatures of `k` from $n - f$ different nodes, then it is computationally infeasible for the adversary to distinguish `r` from a uniform random sample with non-negligible probability.

Randomness. For all integers `k`, the unique integer `r` that `Reveal` would return is a cryptographically secure random number in that `r` is a non-error uniform random sample from the codomain of `Reveal`.

Validity. For all integers `k`, if `r` is the unique integer returned by `Reveal` and `p` is the proof returned by `Generate`, then `Verify(k,p,r)` $\rightarrow$ `True` and it is computationally infeasible for the adversary to find integer `r'`$\neq$`r` such that `Verify(k,p,r')` $\rightarrow$ `True`.

We show two SRO designs, one using trusted hardware and another using cryptography. We then integrate an SRO with Pompē–a state-of-the-art ordered consensus protocol–and prove ordering equality and ordering linearizability. We further demonstrate a quantitative trade-off between the two ordering properties, which helps system designers decide how much random noise to add.

3.3.2 Two SRO designs and Pompē-SRO

An SRO design using trusted hardware Trusted Execution Environments (TEEs) provide a hardware enclave that protects the integrity and confidentiality of user software. Using TEEs to enforce secrecy in blockchains has been actively advocated [25, 10] and practically adopted by the Ethereum testnet [17]. Here, we show how to implement an SRO based on TEEs.

Initialization. Consider that every node has a TEE running the SRO. During initialization, each TEE generates a random number with special CPU instructions (*e.g.,* using `RDRAND` in x86) and runs a distributed consensus protocol to agree on one such number. This number is kept confidential within the TEEs and will be used as the `seed` of a pseudorandom function denoted as `RAND`.

Reveal, Generate and Verify. Given an integer and a set of signatures, a node invokes `Reveal` by forwarding the arguments to its local TEE. The TEE returns `RAND(seed, k)` or an error depending on whether the second parameter contains enough valid signatures of `k`. Similarly, `Generate` forwards `k` to the TEE, which returns `HASH(RAND(seed, k))` where `HASH` is a one-way function. Lastly, `Verify` returns whether parameter `p` equals `HASH(r)`.

Correctness. The same seed and the deterministic pseudorandom function ensures uniqueness and randomness. Secrecy is ensured by the integrity and confidentiality of the TEEs, which keep the seed and random numbers secret and only reveal the random numbers after seeing enough valid signatures. The security properties of one-way functions ensure validity.

We tacitly assume that the initialization (*i.e.,* consensus on the random seed) eventually finishes. This assumption ensures liveness: all invocations of the

Threshold VRF node-side function

```
Produce(Int k) → Share
```

Threshold VRF client-side functions

```
Combine(Set<Share> s) → Int | Error
Valid(Int k, Int node_id, Share s) → Bool
```

The modified node-side function

```
Produce(Int k, Set<Signature> s) → Share | Error
```

Figure 3.6: The interface of (modified) threshold VRF.

SRO functions on a correct node eventually return a result.

An SRO design using cryptography A Threshold Verifiable Random Function (or threshold VRF) is a cryptographic construction used by several Byzantine agreement protocols [41, 56]. TVRFs have been used to select a random set of nodes as a committee and to ensure safety and liveness under a fully asynchronous network model. We show how to use a threshold VRF to construct an SRO for the purpose of equal opportunity.

Let `TVRF` denote a function from an integer to a pseudorandom number, similar to `RAND` in the first design. Figure 3.6 shows the interface of threshold VRF for evaluating `TVRF(k)`. Each node invokes `Produce(k)`, which produces a *share* using its private key. After collecting a sufficient number of shares, a client invokes `Combine`, which returns `TVRF(k)`. However, shares from Byzantine nodes could be invalid. To this end, `Valid` checks, using the corresponding public key, whether a share from a node is valid or not. Threshold VRFs provide two important properties, informally [41]:

Robustness. For all integers `k`, it is computationally infeasible for an adversary to produce enough valid shares such that the integer output of `Combine` is

not `TVRF(k)`.

Unpredictability. Without enough valid shares for `TVRF(k)`, an adversary cannot distinguish `TVRF(k)` from a uniform random sample with non-negligible probability.

We made a slight modification to threshold VRF. For the `Produce` interface, we added a second parameter, a set of signatures of `k` to be verified. If verification fails, correct nodes must return an error instead of a share. We can now design an SRO as follows: `Reveal` forwards the two parameters to all nodes, collects enough valid shares, and invokes `Combine`. `Generate` returns the set of public keys. `Verify` takes all the public keys and a set of shares as input and returns whether all the shares are valid (using the `Valid` interface).

Correctness. Robustness implies uniqueness and validity: for all integers `k`, combining enough valid shares can only produce `TVRF(k)` since an adversary cannot create valid shares leading to a combined value different from `TVRF(k)`. The properties of threshold VRF have been proven under the random oracle model, which implies randomness–informally, hash values are cryptographically secure pseudorandom numbers. Unpredictability implies secrecy because, without valid signatures of `k` out of $n - f$ nodes, an adversary cannot gain enough valid shares and–without enough valid shares–it has no information about `TVRF(k)`. Liveness is ensured, assuming all messages are eventually delivered by the network.

Integrating an SRO with Pompē We now demonstrate how to integrate an SRO with Pompē (*i.e.*, Pompē-SRO). The goal is to enforce ϵ-Ordering equality and Δ-Ordering linearizability. To recap, Pompē associates an *assigned timestamp*

with each command in the output ledger. Consider $\langle c,\ ats\rangle$ in the output ledger where c is a command and ats is the assigned timestamp. Pompē guarantees that (1) ats falls in the time interval associated with the consensus slot; (2) ats is bounded by the minimum and maximum of the timestamps provided by correct nodes. Commands are then ordered by their assigned timestamps. The details have been given in Chapter 2.3.

Intuitively, the plan is to add a random number to the assigned timestamp of each command. Concretely, after consensus is reached for slot `k` in Pompē, a correct node obtains a set of signatures, which proves that consensus has been reached. With `k` and this set of signatures, a correct node invokes the `Reveal` SRO interface and obtains a random number that is used to seed in a random number generator. Importantly, no node can determine what seed the correct nodes will use until it has received sufficiently many signatures, and therefore, the seed is independent of the consensus decision itself.

The random number generator assigns a pseudorandom number r to each command, each uniformly sampled from $[0,\ \Delta_{noise})$. The next chapter describes how to select Δ_{noise}. Instead of directly ordering commands by their assigned timestamps, commands are ordered by $ats + r$.

We call a command *c stable* (or finalized) if the output ledger produced contains c. In Pompē, commands are ordered by the assigned timestamps, and commands in a slot become stable when this slot reaches consensus. After adding random noise, it takes longer for a command to become stable in Pompē-SRO. Suppose the latest slot that reaches consensus is associated with time interval $[ts, ts')$, a command c becomes stable in Pompē-SRO if $ats + r < ts'$, meaning that command c and all commands before c can be produced to the ledger.

Safety and liveness. Pompē guarantees the same safety and liveness properties as the classic SMR protocols [43, 75]. Pompē-SRO provides the same guarantees and differs only by how commands are ordered. We will now focus on proving the new ordering properties.

3.3.3 Proofs of a quantitative trade-off in Pompē-SRO

The remaining question is how to decide Δ_{noise}. We give a quantitative answer based on the upper bound Δ_{net} on message delivery latency, node processing time, and clock drift of correct nodes. This Δ_{net} parameter is defined by the partial synchrony model [54].

Partial synchrony model. One variant of the partial synchrony model introduces the *Global Stabilization Time* (GST). Specifically, there is an unknown time GST such that, after this time, there is a known bound Δ_{net} on network latency and processing time. The safety and liveness of Pompē are proven under this model. We now analyze the ordering properties of Pompē-SRO in the same model. More precisely:

Assumption 3.3.1. *After the global stabilization time (GST), if a command is invoked at time T, correct nodes will assign timestamps within $[T, T + \Delta_{net}]$ for this command.*

Note that a simple clock synchronization protocol has been given as part of the Pompē protocol. We now prove Δ-Ordering linearizability and ϵ-Ordering equality with $\Delta = \Delta_{net} + \Delta_{noise}$ and $\epsilon = 1 - (1 - \Delta_{net}/\Delta_{noise})^2$ in the two invocation case. This result implies a trade-off: as Δ_{noise} approaches infinity, ϵ will approach 0 while Δ will approach infinity.

Lemma 3.3.1. *The assigned timestamp of a command is bounded by timestamps provided by correct nodes.*

Proof. See Chapter 2.3.2.

Theorem 3.3.1. *(Δ-Ordering linearizability) After the global stabilization time (GST), for all invocations i_1 and i_2, if i_1 is invoked more than $\Delta_{net} + \Delta_{noise}$ earlier than i_2, then i_1 will appear before i_2 in the output.*

Proof. Suppose i_1 and i_2 are invoked at time T_1 and T_2. By Assumption 3.3.1 and Lemma 3.3.1, the assigned timestamp of i_1 is in the range $[T_1, T_1 + \Delta_{net}]$. After adding the random noise, the resulting timestamp is in range $[T_1, T_1+\Delta_{net}+\Delta_{noise}]$. Similarly, the resulting timestamp for i_2 is in range $[T_2, T_2+\Delta_{net}+\Delta_{noise}]$. Therefore, if $T_2 > T_1 + \Delta_{net} + \Delta_{noise}$, i_2 will have a higher timestamp and appear after i_1 in the output ledger. □

Theorem 3.3.2. *($\epsilon(2)$-Ordering equality) After the global stabilization time (GST), assuming that $\Delta_{net} < \Delta_{noise}$, for all invocations i_1 and i_2 invoked at the same time, $|Pr[i_1 \prec i_2] - Pr[i_2 \prec i_1]| \leq 1 - (1 - \Delta_{net}/\Delta_{noise})^2$.*

Proof. Let t_i be the assigned timestamp for command c_i and $t'_i = t_i + r$ where r is the random noise. By the definition of our protocol, t'_i is uniformly sampled from the interval $[t_i, t_i + \Delta_{\text{noise}}]$. Thus, we have:

$$Pr[c_1 \prec c_2] = \frac{1}{{\Delta_{\text{noise}}}^2} \int_{t_1}^{t_1+\Delta_{\text{noise}}} \int_{t_2}^{t_2+\Delta_{\text{noise}}} (t'_1 < t'_2) dt'_2 dt'_1$$

By Assumption 3.3.1, $t_1, t_2 \in [T, T + \Delta_{\text{net}}]$. The optimal strategy for the adversary who controls assigned timestamps is assigning $T + \Delta_{\text{net}}$ to c_1 and T to c_2 to minimize the probability of $c_1 \prec c_2$. Thus, we have:

$$Pr[c_1 \prec c_2] \geq \frac{1}{{\Delta_{\text{noise}}}^2} \int_{T+\Delta_{\text{net}}}^{T+\Delta_{\text{net}}+\Delta_{\text{noise}}} \int_{T}^{T+\Delta_{\text{noise}}} (t'_1 < t'_2) dt'_2 dt'_1$$

Eliminate T and, because of $\Delta_{\text{net}} < \Delta_{\text{noise}}$, we have:

$$Pr[c_1 \prec c_2] \geq \frac{1}{{\Delta_{\text{noise}}}^2} \int_{\Delta_{\text{net}}}^{\Delta_{\text{noise}}} \int_{x}^{\Delta_{\text{noise}}} 1 dy dx = \frac{1}{2}(1 - \frac{\Delta_{\text{net}}}{\Delta_{\text{noise}}})^2$$

Because this is a tight lower bound following this strategy, the difference in probabilities of those two output orders is given by:

$$|Pr[c_1 \prec c_2] - Pr[c_2 \prec c_1]| = |1 - 2 * Pr[c_1 \prec c_2]| \leq 1 - (1 - \frac{\Delta_{\text{net}}}{\Delta_{\text{noise}}})^2$$

□

As suggested in Chapter 3.1, real-world concerns about ordering could also arise due to violating ordering equality with three invocations (*i.e.*, enforcing $\epsilon(3)$-Ordering equality is necessary to limit sandwich attacks).

Theorem 3.3.3. *($\epsilon(n)$-Ordering equality) After the global stabilization time (GST), assuming that $\Delta_{net} < \Delta_{noise}$, for all invocations $i_1..i_n$ invoked at the same time, for all two total orders of $i_1..i_n$ denoted as $\succ_1$ and $\succ_2$, $|Pr[\succ_1] - Pr[\succ_2]| \leq 1/n!((1+\alpha)^n - (1-\alpha)^n - n\alpha^n)$ where α denotes the ratio of $\Delta_{net}/\Delta_{noise}$.*

Proof. To prove the bound of $|Pr[\succ_1] - Pr[\succ_2]|$, it suffices to prove a lower bound and an upper bound on any $Pr[\prec]$ (Lemma 3.3.3 and Lemma 3.3.4). Let α denote $\frac{\Delta_{\text{net}}}{\Delta_{\text{noise}}}$ and *modified timestamp* denote the sum of the assigned timestamp and the random number. We also use the following lemma:

Lemma 3.3.2. *For any n independent random variables uniformly sampled from the same interval $[a, b]$, all orderings of those n variables have the same probability of $\frac{1}{n!}$.*

Proof. By symmetry. □

Lemma 3.3.3 (Tight lower bound)**.** $Pr[\prec] \geq \frac{1}{n!}(1 - \alpha)^n$.

Proof. Consider the time interval $[T + \Delta_{\text{net}}, T + \Delta_{\text{noise}}]$. Any command invoked at T has a probability of $1 - \alpha$ being assigned a modified timestamp uniformly

sampled from the interval because the assigned timestamp, t, is picked from $[T, T + \Delta_{\text{net}}]$ and the additional noise, η, is sampled uniformly from $[0, \Delta_{\text{noise}}]$. Thus, the probability of all n modified timestamps being within this interval is $(1 - \alpha)^n$. Using the lemma above, all $n!$ possible permutations of these n commands will have equal probabilities, giving the lower bound $\frac{1}{n!}(1 - \alpha)^n$.

This lower bound is tight by assigning the first command in $\prec$ an assigned timestamp of $T + \Delta_{\text{net}}$ and the last command in $\prec$ an assigned timestamp of T. In this case, all commands' modified timestamps must be included in this interval for $\prec$ to be the output order. □

Lemma 3.3.4 (Tight upper bound)**.** $Pr[\prec] \leq \frac{1}{n!}((1 + \alpha)^n - n\alpha^n)$

Proof. We prove this tight upper bound by induction on n.

The base case, $n = 1$, holds trivially.

For the inductive case, consider how the adversary can pick assigned timestamps to maximize the probability of $\prec$.

For the first command in $\prec$, c_1, because the adversary would like its modified timestamp to be as small as possible, the optimal strategy is assigning an assigned timestamp of exactly T and thus a modified timestamp uniformly sampled from $[T, T + \Delta_{\text{noise}}]$.

There are two possible outcomes for t_1': $t_1' \leq T + \Delta_{\text{net}}$ and $t_1' > T + \Delta_{\text{net}}$.

In the first case, the adversary can assign an assigned timestamp to every other command of at least t_1' to ensure c_1 is the first command in the permutation. But the adversary must still make the other commands follow $\prec$. The optimal strategy is to achieve the tight upper bound of $n - 1$ commands with a

different $\alpha' = \alpha - \frac{t_1' - T}{\Delta_{\text{noise}}}$, which gives the following integral over the probability density function:

$$\begin{aligned}
&\int_0^{\alpha} \frac{1}{(n-1)!}((1+(\alpha-x))^{n-1} - (n-1)(\alpha-x)^{n-1})dx \\
=&\int_0^{\alpha} \frac{1}{(n-1)!}((1+y)^{n-1} - (n-1)y^{n-1})dy \\
=&\frac{1}{n!}((1+\alpha)^n - 1 - (n-1)\alpha^n)
\end{aligned}$$

In the second case, where $t_1' > T + \Delta_{\text{net}}$, the optimal strategy is to delay the latter commands as much as possible. Thus, every other command gets an assigned timestamp of exactly $T + \Delta_{\text{net}}$, and their modified timestamps are all sampled uniformly from $[T + \Delta_{\text{net}}, T + \Delta_{\text{net}} + \Delta_{\text{noise}}]$. To have $\prec$ being the output order, all other commands must be in the interval $[t_1', T + \Delta_{\text{net}} + \Delta_{\text{noise}}]$, which has a probability of $(1 + \alpha - \frac{t_1'}{\Delta_{\text{noise}}})^{n-1}$. Furthermore, those commands need to satisfy $\prec$. Because all those timestamps are sampled uniformly from the same interval, by the lemma above, we have the following integral:

$$\begin{aligned}
&\int_{\alpha}^{1} \frac{1}{(n-1)!}(1+\alpha-x)^{n-1}dx \\
=&\int_{\alpha}^{1} \frac{1}{(n-1)!}y^{n-1}dy \\
=&\frac{1}{n!}(1-\alpha^n)
\end{aligned}$$

Adding the probabilities of the two cases together, we have:

$$Pr[\prec] \leq \frac{1}{n!}((1+\alpha)^n - 1 - (n-1)\alpha^n) + \frac{1}{n!}(1-\alpha^n) = \frac{1}{n!}((1+\alpha)^n - n\alpha^n)$$

This bound is tight following our construction.

□

Returning to the proof of Theorem 3.3.3, given the tight lower and upper bounds, the difference $\epsilon(n)$ is bounded by $\frac{1}{n!}((1+\alpha)^n - (1-\alpha)^n - n\alpha^n)$, $\epsilon(n) \sim 2n\alpha$.

□

Choosing the Δ_{noise} parameter. With Pompē and many other protocols, system designers tune their systems by choosing Δ_{net}. Suppose they now want to mitigate systemic bias, particularly front-running and sandwich attacks. In this case, they would assume $n = 3$ and tune Δ_{noise} based on a target ϵ in Pompē-SRO. In certain legal contexts, $\epsilon \approx 0.1$ is used for equal opportunity: the so-called four-fifth rule [2, 3] says that if two candidates from different ethnic groups are equally qualified for a job, the difference between their chances of getting an offer cannot be more significant than 45% vs. 55%. This rule has also influenced machine learning fairness (*e.g.,* demographic parity) [35].

3.3.4 Implementation

We implement two SRO variants based on Chapter 3.3.2. For the first variant, we choose Intel's software guard extensions (SGX) [22] as the TEE. Random numbers are generated by the function `SHA256(seed + k)` where `seed` is the random seed decided during SRO initialization and `k` is the parameter of `Reveal`. The `SHA256` function is provided by the official SGX library for Linux [21]. This library does not provide a `SHA512` function, which would make the SRO more secure in the blockchain context. Note that the `Int` type in Figure 3.5 and Figure 3.6 does not have to be the typical 4-byte integer. In

blockchains, 32-byte and 64-byte integers are both commonly used.

For the second variant, we start with an implementation of threshold VRF in C++ [56, 15] and modify the `Produce` interface by adding signature verification. Random numbers here are the `SHA512` hash of signatures combining a threshold of shares. The default configuration of this implementation uses the mcl cryptographic library [23] and the BN256 curve. Unlike many BFT systems, cryptographic libraries are not the performance bottleneck of Pompē-SRO. As we will explain later in the evaluation, the overhead is dominated by the noise and trade-off described in Chapter 3.3.3. Therefore, we have chosen the cryptographic libraries based on the convenience of implementation. In the two SRO variants, signature verification in the `Reveal` interface is implemented with the secp256k1 library from Bitcoin [24].

Ease of integration. We integrate the two SRO variants with a Pompē implementation based on HotStuff [12]. Recall that Pompē employs any leader-based BFT SMR protocol and transforms it into a new protocol that enforces ordering properties. This Pompē implementation employs HotStuff because HotStuff is the foundation of Diem [11], an influential blockchain project. Our modifications involve modest system effort and do not modify the complex components that achieve consensus. Instead, we only modify the component producing the totally ordered output, and we wrap this component into a new one that applies the random noise and waits for the new stability condition, as described in Chapter 3.3.2. Our experience suggests that it could be easy to integrate other consensus systems with an SRO for the purpose of equal opportunity.

Clients from different geolocation (or using different network facilities) would be treated equally by Pompē-SRO.
The expected profit of sandwich attacks could decrease significantly.
Baselines could all be significantly biased and there is a trade-off in Pompē-SRO between the two ordering properties.
For performance, Pompē-SRO maintains the same throughput as Pompē and incurs moderate latency overhead.

Figure 3.7: Summary of evaluation results.

3.3.5 Experimental evaluation

We ask two main questions in our evaluation: (1) How do the new ordering properties mitigate front-running, geographical bias, and sandwich attacks? Why are baselines vulnerable?; and (2) What is the end-to-end performance of Pompē-SRO? Figure 3.7 shows a summary of our findings.

We choose three baselines: HotStuff [110], Pompē [112] and Themis [68] representing different existing fairness concepts. HotStuff adopts a *rotating leadership* fairness concept, and we configure HotStuff by making all nodes serve as the leader for the same amount of time. Themis guarantees a fairness property called γ-*batch-order-fairness*, and we choose $\gamma = 1$ which informally means that if all correct nodes receive i_1 before i_2, then i_1 should be ordered before i_2 in the system output. Themis only provides the simulation code instead of an actual system implementation. Pompē adopts the concept of *removing oligarchy* and the output order in Pompē is not unilaterally decided by a leader. We implement two variants of SRO as described in Chapter 3.3.4 and integrate them with Pompē as Pompē-SRO.

Configuration and metrics. We run Pompē-SRO and the three baselines on 52 machines in CloudLab [18] (m400, 8-core ARMv8, 64GB memory, Ubuntu

Linux 20.04 LTS). We run the SROs on a separate set of machines with the Intel Xeon Silver 4410Y and Intel Xeon D-1548 CPUs because our two SRO implementations rely on cryptographic acceleration, and the TEE-based SRO uses Intel SGX. Network latency across different geolocations is emulated using the Linux traffic control (`tc`) utility.

Statistics show that the top countries running Ethereum nodes are: US (40%), Germany (12%), Singapore (5%), UK (4%), Netherlands (4%), France (3%), Japan (3%), Canada (3%), Australia (3%) and Finland (3%) [19]. To answer the evaluation question about fairness, we run 80 nodes simulating these statistics and then evaluate the impact of bias or attacks. For the US, we simulate 15 nodes in Washington, 15 in San Francisco, and 10 in Austin, representing the eastern, western, and central US. For other countries, we simulate all the nodes in one of their major cities. Latency information is from WonderNetwork [20]. The maximum latency in such a configuration is 296ms from Canberra in Australia to Oulu in Finland. We thus assume $\Delta_{net} = 300ms$. We use this configuration of Ethereum to reflect the uneven distribution of blockchain nodes despite all systems we evaluate being permissioned systems.

Our metric for fairness for two commands is the difference between the probabilities of the two possible relative orders of the two commands in the ledger. When two commands are invoked at the same time in Washington and London, the probability of the Washington invocation appearing first in the ledger is 0.76 (denoted as $Pr[W \prec L] = 0.76$) in our measurement of HotStuff. Therefore, $Pr[W \prec L] - Pr[L \prec W] = 0.76 - 0.24 = 0.52$, which is much higher than a reasonable target ϵ (*e.g.*, 0.1).

To answer the evaluation question about performance, we measure the la-

Baselines	**HotStuff**	**Pompē/Themis**
$Pr[W \prec L] - Pr[L \prec W]$	0.52	1
$Pr[W \prec T] - Pr[T \prec W]$	0.93	1
$Pr[L \prec M] - Pr[M \prec L]$	0.45	1
$Pr[M \prec T] - Pr[T \prec M]$	0.85	1

Figure 3.8: Geographical bias measured in HotStuff, Pompē and Themis. W, L, M and T stand for Washington, London, Munich and Tokyo. For two simultaneous invocations from two cities, Pr[City1 $\prec$ City2] stands for the probability of the invocation from City1 being the first in the ledger.

Pompē-SRO	$\Delta_{noise} = \Delta_{net}$	$\Delta_{noise} = 5 * \Delta_{net}$
$Pr[W \prec L] - Pr[L \prec W]$	0.036	0.007
$Pr[W \prec M] - Pr[M \prec W]$	0.158	0.033
$Pr[W \prec T] - Pr[T \prec W]$	0.399	0.087
$Pr[L \prec M] - Pr[M \prec L]$	0.119	0.25
$Pr[L \prec T] - Pr[T \prec L]$	0.367	0.82
$Pr[M \prec T] - Pr[T \prec M]$	0.269	0.56

Figure 3.9: Geographical bias measured in Pompē-SRO with $\Delta_{noise} = \Delta_{net}$ (*i.e.*, 300ms) and $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.*, 1500ms).

tency and throughput of Pompē and Pompē-SRO by scaling the systems from 4 to 49 nodes. In these experiments, each node runs on a separate machine, and each machine is configured with a 150ms outbound network latency with the `tc` utility. We fix a setup with 800 concurrent clients, and each client invokes commands in a closed loop (*i.e.,* it waits for the consensus result of its currently outstanding command before invoking the next one). We aim to measure the overhead of Pompē-SRO over Pompē in a geo-distributed setup. As for batching, we use 1500ms as the time interval associated with each consensus slot, and each assigned (*i.e.,* median) timestamp is associated with only one command.

Bias and front-running. Figure 3.8 shows that geographical bias could be significant in the baselines. The output order produced by Pompē and Themis is deterministic: simultaneous invocations from the four cities are always or-

dered as Washington ≺ London ≺ Munich ≺ Tokyo. The reason is that in our configuration, simulating the node distribution of Ethereum, Washington has a lower network latency than most of the 80 nodes in the other cities. Therefore, in Pompē, the median timestamp of any quorum for the Washington invocation will be lower. The fairness property of Themis directly implies that, in this configuration, Washington should be ordered first.

HotStuff rotates leadership, and the invocation from Tokyo would have a chance to appear first in the ledger when a Tokyo node serves as the leader. However, this chance is still low compared to Munich invocations. Specifically, we find that $Pr[\mathrm{M} \prec \mathrm{T}] = 0.925$, leading to the 0.85 in the table. This difference increases to 0.93 when compared to Washington. While Tokyo and Washington are geographically distant, a similar bias can happen between nearby cities. London and Munich are both in Europe, but $Pr[\mathrm{L} \prec \mathrm{M}]$ is 0.725 leading to the 0.45 in the table.

Figure 3.9 shows how geographical bias could be effectively reduced in Pompē-SRO. By adding a random noise sampled from $[0, \Delta net]$ (*i.e.*, $[0, 300ms]$) to the median timestamp of Pompē, all probability differences could be controlled under 0.4. If $\epsilon = 0.1$ is a target for the system, system designers could then choose $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.*, $1500ms$), and the worst case bias across the four cities could be controlled under 0.087 as shown in the third line. In real-world deployments, Δ_{noise} could be empirically chosen by considering the most distant clients for fairness.

Front-running typically occurs when one client has lower network latencies to most nodes than another, as Washington does in this configuration. By adding random noise, Pompē-SRO could give all the clients involved in the liq-

uidation events (explained in Chapter 3.1) more balanced chances of obtaining the $42.6M profit over 32 months.

Sandwich attacks. An exchange maintains a pool of some token A (*e.g.,* USD) and some token B (*e.g.,* CNY). For example, people traveling from the US to China may put some USD into the pool and take some CNY away. Similar pools exist on blockchains, and the trading volume of Uniswap, a decentralized exchange, has exceeded one trillion dollars [9].

Pools in these exchanges follow a constraint: *amount of tokenA* $*$ *amount of tokenB* = *constant*, which is called the automated market makers (AMMs) approach [26]. Suppose the constant is 1800; the number of tokens A and B in the pool could be, for instance, $\langle 45, 40\rangle$, $\langle 60, 30\rangle$, or $\langle 75, 24\rangle$. Say $\langle 75, 24\rangle$ is the current status, and Alice needs 15 token A. Alice can put 6 token B into the pool and take 15 token A away so that the pool state moves to $\langle 60, 30\rangle$.

The sandwich attack works as follows. After seeing Alice's transaction, an attacker, Bob, first buys 15 token A, moving the pool status to $\langle 60, 30\rangle$. Alice now needs to pay 10 (instead of 6) token B in order to exchange for 15 token A and move the pool status to $\langle 45, 40\rangle$. Bob can thus exchange the 15 token A back to 10 token B, making a 4 token B profit. The three steps reflect the three invocations in Figure 3.1. If the market prices of the tokens are $100 and $200, respectively, we will get the dollar values in Figure 3.1.

The success of sandwich attacks depends on whether the attacker's first invocation (i_1 in Figure 3.1) can front-run the victim's invocation since the attacker can always close the attack by delaying its last invocation. Suppose the network conditions of the attacker and victim are similar to those of London and Munich.

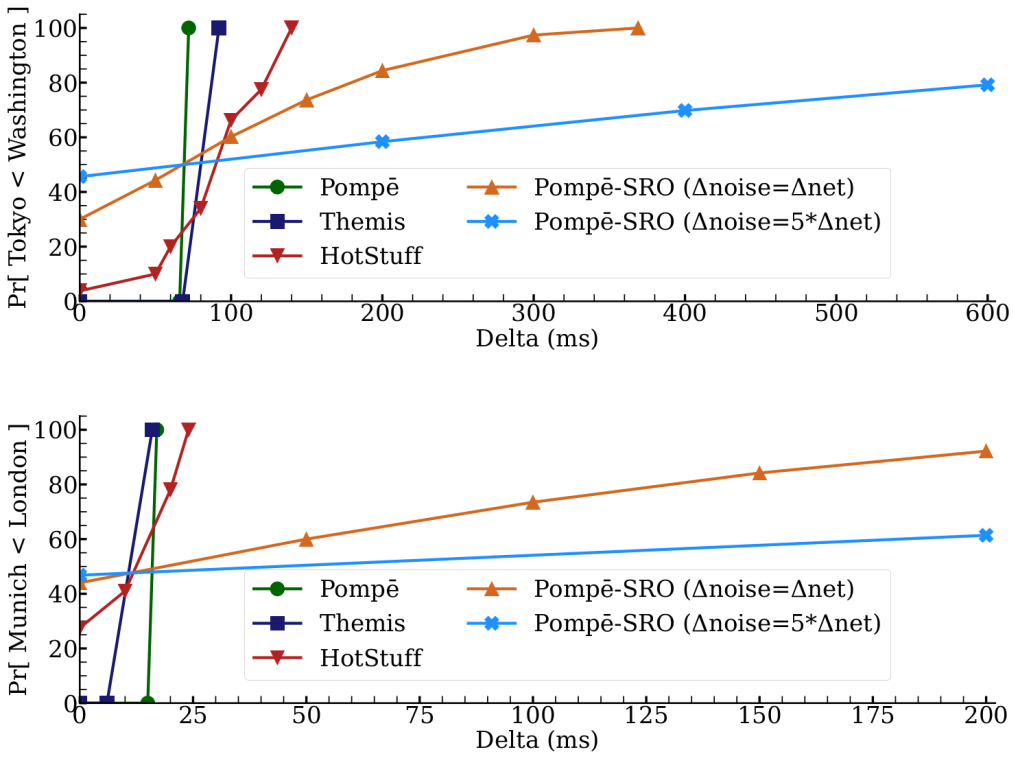

Figure 3.10: Trade-off between ϵ-Ordering equality and Δ-Ordering linearizability in Pompē-SRO. The top figure shows the results for two clients in Washington and Tokyo. The bottom figure shows the results for two clients in London and Munich.

Figure 3.8 shows that, in systems like Pompē or Themis, the attacker can succeed with a high probability. In Pompē-SRO, the attacker would have a lower expected profit because of ϵ-Ordering equality.

The trade-off between ordering equality and ordering linearizability. Figure 3.10 shows the relation between ordering and invocation time for two invocations from Washington and Tokyo (or London and Munich). Specifically, the x-axis represents how long the Tokyo client invokes its commands *before* the

Washington client. The y-axis represents the probability of the Tokyo invocation being ordered first. Intuitively, when the Tokyo client invokes its command early enough, its probability of being ordered first will be 100%.

For Pompē-SRO, with $\Delta_{noise} = \Delta_{net}$, the two invocations are treated less equally when the x-axis is 0 compared to $\Delta_{noise} = 5 * \Delta_{net}$, but it leads to a lower Δ (*i.e.*, 369ms) for ordering linearizability. In other words, for Pompē-SRO with $\Delta_{noise} = \Delta_{net}$, the y-axis will reach 100% when the x-axis is 369ms. With $\Delta_{noise} = 5 * \Delta_{net}$, the probability of the Tokyo invocation ordered first is only 69.8% even if it is invoked 400ms earlier. This shows the trade-off between removing the influence of irrelevant features and preserving the signal strength of the relevant features when adding the random noise in Pompē-SRO.

The results for Pompē and Themis look similar. For Pompē, the Tokyo invocation is guaranteed to be ordered first if it is invoked more than 72ms earlier, while this threshold is 92ms for Themis. The reason is that Pompē and Themis order commands based on when nodes receive the invocations. When the difference in invocation time is above this threshold, most of the nodes in our configuration will receive the Tokyo invocation earlier. The result for HotStuff starts from 3.8% when the x-axis is 0 and grows to 100% when the x-axis is 140ms. This result follows the intuition that the Tokyo invocation will have better chances of reaching the leader node earlier if it is invoked earlier.

The bottom part of Figure 3.10 shows the results of invocations from Munich and London. Compared to the left part, we make two observations. First, for HotStuff and Pompē-SRO, the chances are closer to 50% when the x-axis is 0. Second, for Pompē and Themis, the threshold becomes 17ms, which is lower than 72ms or 92ms. Both observations are because these two cities are both in

	|s|=0 (base)	|s|=200
TEE	3us	base+20.2ms
TVRF (67/100)	95.2+4.3ms	base+19.9ms
TVRF (133/200)	185.7+9.7ms	base+19.9ms

Figure 3.11: Latency of the Reveal interface in two SRO implementations. |s| denotes the number of signatures that need to be verified in the second parameter of Reveal. The base case of TVRF consists of generating and combining shares. The numbers after TVRF denote the threshold and total number of nodes.

Europe and thus geographically closer.

Given $\epsilon = 0.1$ as a target, our results in Chapter 3.3.3 show that $\Delta_{noise} = 20 * \Delta_{net}$ is necessary to provide a guarantee of this target. This is because our results assume the worst case: two simultaneous invocations would obtain timestamps that differ by Δ_{net}. In practice, the worst-case difference could be much lower than Δ_{net} so that system designers can choose this parameter according to their actual deployment setup.

The latency of secret random oracles. Figure 3.11 shows the latency of the two SRO implementations. We show two cases of |s|=0 and |s|=200, where |s| denotes the number of signatures to be verified in the second parameter of Reveal. When |s|=0, Reveal does not verify signatures, and the latency is solely for generating random numbers. For the TEE variant, the latency consists of entering an SGX enclave and computing a SHA256 function, which takes only $3us$. For the threshold VRF variant, the latency consists of three parts: (1) generating shares, (2) collecting the shares over the network, (3) combining a threshold of shares. The results in Figure 3.11 show (1) and (3). Under a setup of 100 nodes and 67 as the threshold, the latency of producing a share is $95.2ms$, and the latency of combining 67 shares is $4.3ms$. When moving to a setup of

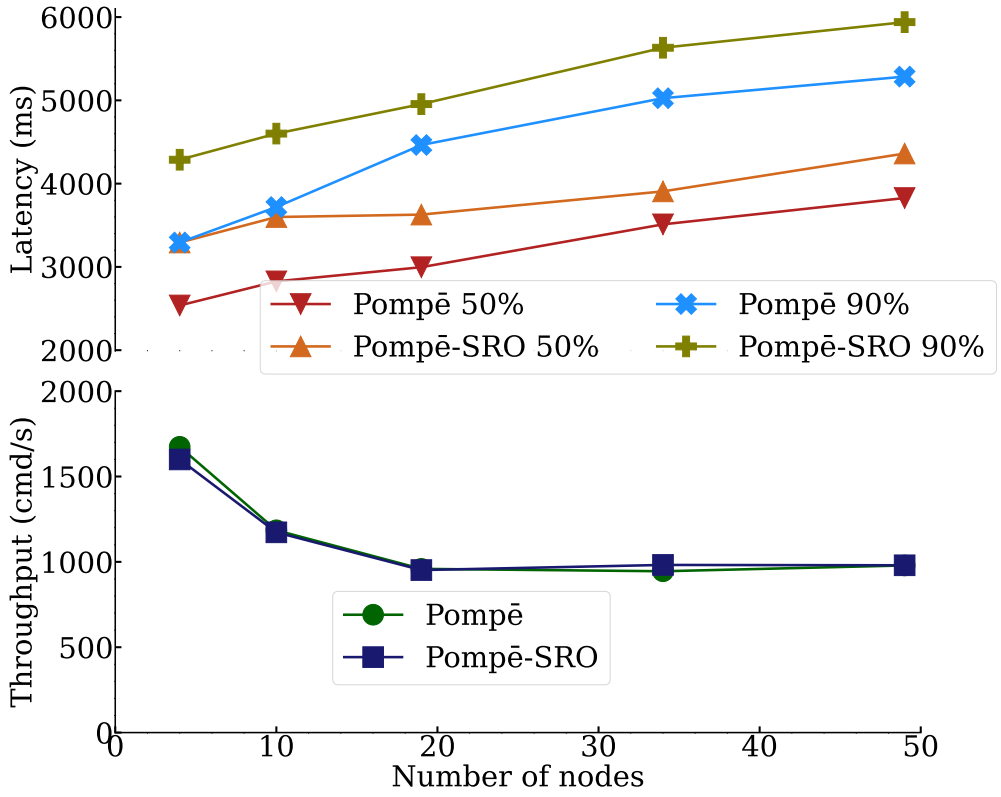

Figure 3.12: End-to-end performance of Pompē and Pompē-SRO with $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.*, 1500ms).

200 nodes with 133 as the threshold, the two latencies roughly double. This is because threshold VRF algorithms have a high constant factor in their computational complexity, which dominates latency in these setups. Lastly, when $|s|$=200, the latency of verifying the 200 signatures is around 20ms, both within and outside an SGX enclave.

These results show a trade-off between performance and decentralization. The SRO based on SGX has a much lower latency, making it more practical, but it requires trusting a centralized party, Intel. In the following experiments, we choose performance in this trade-off and use the TEE variant of SRO, since we have observed TEE-based projects running on the Ethereum testnet [17].

End-to-end performance of Pompē-SRO. Figure 3.12 shows the latency and throughput of Pompē-SRO and Pompē when scaling from 4 to 49 nodes. The configurations of these experiments are described at the beginning of this section. The x-axis represents the number of nodes. The y-axis of the upper part represents end-to-end latency in milliseconds. The y-axis of the lower part represents system throughput in command per second.

Pompē-SRO achieves the same throughput as Pompē, meaning that integrating with an SRO does not impact throughput. This is expected because obtaining random numbers from an SRO is much cheaper than consensus, so the consensus part of Pompē-SRO is the throughput bottleneck.

As for latency, the median latency for 49 nodes increases from 3827ms in Pompē to 4361ms in Pompē-SRO, increasing by 534ms. The 90 percentile tail latency increases from 5285ms to 5939ms, increasing by 654ms. The relative increases are only 14% and 12.4%, respectively.

In real-world blockchains, the typical end-to-end latency is on the magnitude of minutes (*e.g.,* Bitcoin) or seconds (*e.g.,* Ethereum) which are higher than the latency results in Figure 3.12. However, the random noise being added could be similar to our experiments (*e.g.,* $\Delta_{noise} = 1500ms$), which would lead to a potentially lower relative latency increase.

CHAPTER 4

RELATED WORK

4.1 Social choice, game theory and mechanism design

Social choice. Social choice theory studies desirable properties in the context of elections. A seminal work in this area is by Kenneth Arrow [31], who won the Nobel Prize in Economics Sciences in 1972. In this work, Arrow proves that every voting rule cannot satisfy three properties altogether: *non-dictatorship*, *unanimity*, and *independence of irrelevant alternatives* (IIA). IIA is a controversial property in this result so we only borrowed the first two properties into ordered consensus as Byzantine oligarchy and ordering unanimity.

Arrow's impossibility theorem is fundamental because, instead of trying to propose a new voting rule competing with existing rules, it reveals the general limitations that all voting rules are subject to. Inspired by this theorem, Gibbard and Satterthwaite defined the *manipulation* property and proved that any voting rule is either dictatorial or manipulable [57, 99]. This property inspired our definition of Byzantine democracy and impossibility theorem, Theorem 2.2.1. This theorem attempts to reveal the general limitations of all SMR protocols, following the spirit of Arrow's and Gibbard-Satterthwaite impossibilities.

In the past few decades, computer scientists have become interested in social choice theory, leading to the field of computational social choice [37]. While this field focuses on preventing manipulation using results from computational complexity theory, our work is the first attempt to connect the seminal results in social choice with SMR and distributed consensus.

Game theory. The state-of-the-art has connected BFT with game theory [28, 27] and the foundation of game theory is the Nash theorem [90]. John Nash, as a mathematician, proved that every normal-form game must have an equilibrium by connecting the rationality model [100, 108] with the Brouwer fixed-point theorem (*i.e.,* a fixed point is interpreted as an equilibrium). A concise proof of this theorem can be found in this book [96]. As practitioners, we find the concept of equal opportunity and its violations (*i.e.,* systemic bias) to be empirically more prevalent in real-life scenarios than Nash equilibrium, which will be illustrated with examples in Chapter 4.2. Therefore, without reusing the concept of Nash equilibrium, this work deviates from game theory since its foundation.

We do reuse the concept of *expected value* from the rationality model when explaining violations of equal opportunity in Chapter 3.1. Specifically, a biased system could give higher expected value to certain clients and the rationality model says that *rational* (or selfish) clients would prefer such a system giving them higher expected value to a system that treats all clients equally. The study of the rationality model has taken an axiomatic approach: if the preferences of a client (*i.e.,* a person) satisfy a set of axioms, then this client is provably an expected value maximizer (aka. expected utility maximizer).

This axiomatic approach has influenced our study of equal opportunity in two ways. First, instead of simply saying that a point system is a fair mechanism for ordering, we explore the principles (*i.e.,* axioms) behind a point system and identify how violations of each principle lead to real-world fairness concerns. The two principles specified in Chapter 3.2.2 may not be a complete set of axioms and, as a future work, we plan to identify all axioms that make a mechanism provably a point system. Second, our definition of the consistency

principle in Chapter 3.2.2 is borrowed from one of the axioms specified in the study of rationality [100].

Fairness of voting in game theory. A branch of game theory studies notions of fairness in the context of voting. Key results include the Banzhaf power index [34] and the Shapley–Shubik power index [104]. The consideration is that voters may have *different* powers: for example, different states have different voting power in the US House of Representatives according to the population. The power index is a quantitative measure of how much power each voter has, given a voting rule. Fairness concerns could arise when there is a mismatch between the power index of a voter and the actual population represented by that voter. If we consider correct nodes and Byzantine nodes as two voters with different power, the power index could be used to quantify Byzantine influence in protocols like Pompē. Applying the notion of a power index to Pompē-SRO is difficult because it has been studied in the context of deterministic voting rules (*e.g.,* those used in the United Nations), while Pompē-SRO adopts randomness for equal opportunity.

Mechanism design. The Gibbard-Satterthwaite impossibility theorem is also the starting point of mechanism design [105], which studies how to design a *strategyproof* auction. An auction is strategyproof if all participants within an auction are rational and their best strategy is to be honest instead of lying. An example is that the second-price auction mechanism can be strategyproof under certain conditions. In the context of SMR, a strategyproof protocol would mean that all the nodes report timestamps honestly instead of trying to manipulate the output order by lying on the timestamps. Given this similarity, insights

from the field of mechanism design (*e.g.*, assumptions on utility functions) may help design new SMR protocols that get around the impossibility in Chapter 2.2, assuming that Byzantine nodes are rational instead of arbitrary. We decided to first explore insights around equal opportunity because we find them to be more connected with real-world fairness concerns in blockchains, leaving the exploration of insights from mechanism design as future work.

4.2 Equal opportunity in the legal context

Employment. A landmark legislation for equal opportunity is the Civil Rights Act that outlaws discrimination based on race, gender, religion, and national origin [1]. A further legislation is the Equal Employment Opportunity Act [2, 3]. In this line of legislation, there is a *four-fifth rule* saying that if two candidates from different ethnic groups are equally qualified for a job, then the difference between their chance of employment cannot be more biased than 45% vs. 55%, leading to the suggestion of $\epsilon \approx 10\%$ at the end of Chapter 3.3.3. Slightly different from our model, the four-fifth rule is named after the ratio of probabilities (*i.e.*, $\frac{45\%}{55\%} \approx 80\%$ which is four fifth).

Financial exchange. Financial regulation laws require stock exchanges to be *impartial* to all traders. However, some stock exchanges went public themselves so they wished to increase their revenue in order to grow the stock price. These exchanges started to have real incentives to accept bribes from traders, resulting in serious scandals about *consistency*.

A few traders came up with many new transaction types and rules so that

the existence (or absence) of certain types of transactions will change how to order and process other types of transactions. These traders, through bribery, convinced the stock exchanges to add these rules into their system. While the rules are public, only these bribers know how to use them to take advantage of other traders. These inconsistent ordering rules have caused serious fairness concerns and law enforcement actions [80].

Kidney exchange. Humans have two kidneys and only one is absolutely necessary so that kidneys from healthy people can be transplanted to patients with kidney failures. Kidneys used to be exchanged through the market and rich patients had a better chance of getting kidneys. This raised serious fairness concerns and led to legislation that transferred the operation to the government in order to mark wealth as an irrelevant feature [4]. While this law enforced *impartiality* between the rich and the poor, *consistency* became a concern. The algorithm was inconsistent and the order of two patients getting kidneys could switch due to a third patient joining the kidney matching system. As a result, a new algorithm was proposed as an amendment later enforcing consistency [5]. Details of the point system in this context, including the relevant features and the formula being used, are also explained in Chapter 2 of this book [111].

Olympic games. The Olympic games define and enforce a set of rules. Based on the rules, a game decides an order of athletes and is required to give equal opportunities to all athletes. The rules are just like laws and they specify which relevant features should be measured in each game, either through equipment or human judges. A key irrelevant feature here is nationality and judgements should not be biased to any nation. Besides, due to cognitive bias, a judge may

make correlated decisions when judging a sequence of observations. Consider the diving game, athletes dive alternatively and the performance of an athlete may impact the scores given to the next one [74]. Judges typically need to take professional training in order to combat such implicit bias and make consistent judgments, following the consistency principle defined in Chapter 3.2.2.

Immigration visa. International students working in the US need government authorization. A visa type is called H-1B and the government conducts a lottery to decide who can get this visa. The only relevant feature is education level and graduate students have a higher chance of winning the lottery. For students at the same education level, the lottery will give them equal opportunity if the lottery is *independent* of anything that students control or influence. Otherwise, if students could manipulate the lottery, such a lottery would be regarded as an unfair mechanism. This gives an intuition of why the SRO outputs need to be independent of any information accessed by other system components.

4.3 Order fairness and the state-of-the-art of SMR

Order fairness. Recent work by Kelkar et al. [69] also recognizes the need to introduce a new ordering property for BFT, which they characterize as order fairness. Their work shows that a natural definition of Receive-Order-Fairness, which states that the total order of commands in the output ledger must follow the actual receiving order of at least a γ-fraction of all nodes, is impossible to achieve due to the Condorcet paradox. They thus relax Receive-Order-Fairness and design SMR protocols that detect and resolve the Condorcet cycles [67, 68].

Recall that Condorcet cycles have been illustrated in Theorem 2.2.2.

Starting from a similar motivation, our work takes a different approach and there are four major differences.

First, rather than trying to characterize the fairness of a few new SMR protocols, we focus on specifying the degrees of Byzantine influence that apply to *all* SMR protocols. Specifically, the notions of Byzantine oligarchy and Byzantine democracy, together with the corresponding theorems, characterize the degree to which it is possible (and impossible) to curtail the influence of Byzantine nodes in determining the total order in all SMR protocols. Thus, while Kelkar et al. argue that protocols that use timestamps from a quorum of nodes to order commands are not suitable for ensuring fairness (as they suffer from the type of manipulation described at the end of Chapter 2.3.1), we can prove (see Theorem 2.2.1) that *any* protocol is subject to such manipulation, as long as we uphold free will. As explained in Chapter 4.1, this more general approach that we take is inspired by the spirit and results of social choice theory, in particular, the Arrow's and Gibbard-Satterthwaite's impossibilities.

Second, we argue that the Receive-Order-Fairness property is actually *unfair* because it can lead to systemic bias. As shown in Chapter 3.3.5, Themis, which enforces a variant of Receive-Order-Fairness, can be significantly biased in a deployment setup similar to the Ethereum blockchain. Such systemic bias can further lead to significantly different expected values for clients who should be treated equally. Different from Receive-Order-Fairness, the notion of removing Byzantine oligarchy does not imply systemic bias and it could indeed co-exist with equal opportunity, as demonstrated by Pompē-SRO.

Third, as shown in Chapter 3.1, Receive-Order-Fairness cannot address real-world fairness concerns such as front-running or sandwich attacks. Similarly, simply removing Byzantine oligarchy and enforcing ordering linearizability can neither address these concerns. Therefore, Chapter 3 turns to the concept of equal opportunity and attempts to connect the ordering properties with real-world concerns in blockchains and other legal contexts (see Chapter 4.2). To the best of our knowledge, the study of Receive-Order-Fairness is specific to the context of blockchains and it is isolated from the rich literature on fairness or equal opportunity in social sciences.

Lastly, while Pompē and Pompē-SRO can reuse any existing BFT protocol in its consensus phase, Kelkar et al. designed a compiler to automatically convert a standard consensus protocol into one that satisfies order fairness. However, protocols output by this compiler require more resources than a standard BFT protocol for the same level of fault tolerance; for example, in the same setting as in standard BFT (leader-based, partial synchrony network model) with γ set to 1 (their best case), these protocols require at least $4f + 1$ nodes to tolerate f Byzantine failures, rather than the $3f + 1$ nodes needed by Pompē.

Leader-based BFT protocols. There is a long line of work on practical Byzantine consensus protocols [36, 44, 47, 59, 63, 72, 73, 82, 83, 84, 86, 94, 106, 109], starting with the seminal work of PBFT [43]. These works focus on improving performance, round complexity, fault models, etc. Some works also focus on using trusted hardware to improve fault thresholds [36, 46, 66, 79]. However, all of them employ a special leader node to orchestrate both ordering and consensus, so they suffer from Byzantine oligarchy.

There are some works that defend against faulty leaders, but they focus only on preventing faulty leaders from affecting the system's performance or defenses for a restricted class of attacks. For example, Aardvark [48] employs periodic leader changes to prevent a faulty leader from exercising full control over the system's performance. It achieves this by having correct nodes set an expectation on minimal acceptable throughput that a leader must ensure and trigger a leader election in case the current leader fails to meet its expectation. While Aardvark [48] focuses on achieving acceptable performance in the presence of faulty leaders, Prime [29] targets a different performance property: any transaction known to a correct node is executed in a timely manner. The Prime ordering protocol consists of a pre-ordering phase and a global ordering phase. Unlike Pompē's ordering phase, the pre-ordering phase imposes only a partial order, rather than a timestamp-based global ordering in Pompē.

Instead of monitoring leaders to detect (or prevent) certain attack vectors, Pompē separates ordering from consensus, which completely eliminates a leader's power in selecting which transactions to propose and in what order. More generally, our work provides the first systematic study of desirable properties when employing BFT protocols for systems that span multiple administrative domains, proves what are impossible, and designs mechanisms to realize desirable properties that are achievable.

Rotating leaders. BART [28] enables cooperative services to tolerate both Byzantine faults and rational (selfish) behavior under the new BAR (Byzantine, altruistic, and rational) model. The consideration of rational behavior leads to an SMR design with rotating leaders, which has now become a standard practice for blockchains based on BFT [14, 45, 110]. However, the rotating

leader paradigm still suffers from Byzantine dictatorship because a Byzantine node can still dictate ordering when it is in the leadership role, whereas Pompē achieves stronger properties by separating ordering from consensus. Furthermore, the rotating leader paradigm suffers from systemic bias, as illustrated in Chapter 3.1. Pompē-SRO achieves properties that control systemic bias to a low degree that is considered acceptable.

Leaderless BFT protocols. Recognizing the implications of relying on a special leader, Lamport offers a leaderless Byzantine Paxos protocol [76]. Unfortunately, it relies on a synchronous consensus protocol to instantiate a "virtual" leader, which requires at least $f + 1$ rounds, where f is the maximum number of faulty nodes in the system and the duration of each round must be set to an acceptable round trip delay. When the number of nodes is high or when nodes are geo-distributed, this protocol adds unacceptable latencies. Democratic Byzantine Fault Tolerance (DBFT) [50] is another leaderless Byzantine consensus protocol, which builds on Psync, a binary Byzantine consensus algorithm. As in Lamport's leaderless protocol [76], Psync terminates in $O(f)$ message delays, where f is the number of Byzantine faulty nodes, even though DBFT relies on a weak coordinator for a fast path through optimistic execution.

EPaxos [89], or Egalitarian Paxos, is an SMR protocol that attempts to make the system egalitarian. While the concept of egalitarianism is closely related to equal opportunity, EPaxos does not specify nor enforce the egalitarian ideals except being a leaderless protocol. Moreover, EPaxos ensures safety and liveness only in a crash fault model and it is unclear how to ensure those properties in a Byzantine fault model, which is our target setting.

Building on the work of Cachin et al. [40, 42], HoneybadgerBFT [88] and BEAT [53] propose leaderless protocols that preserve liveness even in asynchronous and adversarial network conditions. To achieve these properties, they rely on randomized agreement protocols, which bring significant complexity and costs. Unfortunately, these works do not defend against the formation of a Byzantine oligarchy nor do they satisfy ordering linearizability.

Censorship-resistance. HoneybadgerBFT [88] and Helix [32] run consensus on transactions encrypted with a threshold encryption scheme to prevent malicious nodes from censoring transactions, but faulty nodes can always filter transactions based on metadata, a point made by Herlihy and Moir [64]. In contrast, Pompē's separation of ordering from consensus offers a simple mechanism to prevent censorship: once a correct node executes the ordering phase, the transaction is not only guaranteed to be included in the ledgers of correct nodes, it will also be included in a position determined by the assigned timestamp of the transaction.

Accountability and proofs. Herlihy and Moir [64] propose several mechanisms to hold participants accountable in a consortium blockchain. These techniques extend and generalize prior work on accountability [60, 61] and untrusted storage [81, 87]. Similarly, nodes can produce succinct (zero-knowledge) proofs of their correct operation, which other nodes can efficiently verify [38, 93, 102, 103]. Recent work [78, 91] employs such proofs to reduce CPU and network costs in large-scale replicated systems (*e.g.,* blockchains). Unfortunately, such proofs do not prevent a Byzantine leader node from deciding which commands to propose and in what order.

Permissionless blockchains. A trend in the blockchain community is to avoid energy-intensive proof-of-work mechanisms. This has led to permissionless blockchains that employ a BFT protocol among a set of nodes chosen based on different mechanisms (*e.g.,* verifiable random functions, financial stake, etc.) to agree on a value [51, 58, 70, 71]. Pompē and Pompē-SRO can be used as a building block in some of these blockchains.

Specifically, most real-world blockchains have moved to proof-of-stake, including Ethereum, Solana, and Avalanche, among others. In practice, all of them assume *precise membership*, meaning that, at any time, a node knows how many other nodes are in the system as well as their amount of stake. Based on this information, a leader's schedule is calculated. Therefore, proof-of-stake blockchains are a lot similar to SMR, compared to proof-of-work.

CHAPTER 5

CONCLUDING REMARKS AND FUTURE WORK

It is non-trivial to develop an empirical and general theory. The development of ordered consensus, including the theory and practice, has been inspired by great works and minds in the history of science. Chapter 4 has introduced how ordered consensus is inspired by social choice, game theory, mechanism design, and social sciences. This concluding chapter introduces how ordered consensus is inspired by Newton's laws of motion, the pioneer and role model of empirical theories.

The first Newton's law, which states that a body remains at rest or in motion at a constant speed in a straight line without a force, introduces the language of *frame of reference*. This language is borrowed from the concept of coordinate system invented by French mathematician, Descartes. As a clumsy imitation, Chapter 2 borrows the concept of voting preference from Arrow, an American mathematician and economist, and creates the language of *ordering indicators* as the foundation of ordered consensus.

The second Newton's law introduces a key abstraction, *the mass of a body*. Together with the concept of acceleration based on a frame of reference, Newton gives an imperfect but simple definition of *force*. As another clumsy imitation, Chapter 3 introduces the language of *chance relation* and, together with ordering indicators and preferences, gives a simple definition of *equal opportunity*.

There are two insightful lessons taught by the second Newton's law. First, Newton's definition of force flourished because it can explain a good number of real-world phenomena, although not all of them. Second, it tells us that mass is

the right concept to study for an object and the more obvious concept of *weight* is a combination of mass and a special acceleration (*i.e.,* gravity of Earth). These two lessons have guided our development of ordered consensus. First, Chapter 4.2 attempts to explain real-life phenomena beyond computer systems with the definition of equal opportunity in Chapter 3. Some explanations are indeed borrowed from related works in social sciences. Second, while the study of blockchains has focused on the obvious concept of *extractable value* [52, 95], our work focuses on the concept of *chance*. Extractable value is the difference of two *expected values*, with and without an attack; and expected value is the combination of a *probability distribution* over ledgers and a *value function* that maps a ledger to a real number. The argument is that system designers should study the chances that a system gives to clients (*i.e.,* the probability distribution) and extractable value can thus be derived, just like the concept of weight can be derived from mass in Newton's construction.

The journey of developing ordered consensus will continue and below are some possible directions.

Connecting with mechanism design and differential privacy. There may be more mathematical tools that ordered consensus can borrow for constraining Byzantine influence and achieving equal opportunity. For constraining Byzantine influence, mathematical tools from mechanism design may be helpful as explained in Chapter 4.1. For achieving equal opportunity, mathematical tools from differential privacy may actually help. Specifically, the random noise used in Pompē-SRO and the quantitative analysis in Chapter 3.3.3 are related to the *Laplacian mechanism* in differential privacy. In this mechanism, random noise sampled from a Laplacian distribution (instead of a uniform distribution) is

used to bound the difference between two probabilities. In the future, we plan to explore how tools from these fields may introduce more insights into the study of ordered consensus.

Pushing for an industrial impact. As systems researchers, we hope that our research is not only impactful in academia but also guides industrial practice. To this end, the author of this dissertation is joining an industrial blockchain project called Firedancer [1]. This project aims to reshape the foundation of the Solana blockchain, the largest blockchain other than Ethereum, and Firedancer is launched by a Wall Street trading firm that is famous for front-running.

[1] `https://github.com/firedancer-io/firedancer/`

BIBLIOGRAPHY

[1] Civil rights act. https://www.congress.gov/bill/88th-congress/house-bill/7152, 1964.

[2] Equal employment opportunity act. https://www.eeoc.gov/history/equal-employment-opportunity-act-1972, 1972.

[3] Uniform guidelines on employee selection procedures. https://www.ecfr.gov/current/title-29/subtitle-B/chapter-XIV/part-1607, 1978.

[4] National organ transplant act. https://www.congress.gov/bill/98th-congress/senate-bill/2048, 1984.

[5] National organ transplant program extension act. https://www.congress.gov/bill/101st-congress/house-bill/5146, 1990.

[6] MakerDAO. https://makerdao.com, 2017.

[7] Concord Byzantine fault tolerant state machine replication library. https://github.com/vmware/concord-bft, 2018.

[8] libhotstuff: A general-purpose BFT state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff, 2018.

[9] Uniswap. https://uniswap.org/, 2018.

[10] Blockchains in trusted execution environments (TEEs). https://medium.com/@nadeem.bhati/, 2019.

[11] Diem. https://www.diem.com/en-us/, 2019.

[12] A Pompē implementation based on HotStuff. https://github.com/Pompe-org/Pompe-HS, 2020.

[13] Aave liquidity protocol. https://aave.com/, 2020.

[14] State machine replication in the Diem blockchain. https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/, 2020.

[15] A threshold VRF implementation. https://github.com/fetchai/research-dvrf, 2020.

[16] Azure network round-trip latency statistics. https://learn.microsoft.com/en-us/azure/networking/azure-network-latency, 2023.

[17] Block building inside SGX. https://writings.flashbots.net/block-building-inside-sgx, 2023.

[18] CloudLab. https://cloudlab.us/, 2023.

[19] Ethereum mainnet statistics. https://ethernodes.org/countries?synced=1, 2023.

[20] Global ping statistics: Ping times between WonderNetwork servers. https://wondernetwork.com/pings, 2023.

[21] Intel software guard extensions for Linux OS. https://github.com/intel/linux-sgx, 2023.

[22] Intel software guard extensions (SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2023.

[23] mcl: A portable and fast pairing-based cryptography library. https://github.com/herumi/mcl, 2023.

[24] The secp256k1 library from Bitcoin. https://github.com/bitcoin-core/secp256k1, 2023.

[25] The sting framework (SF). https://initc3org.medium.com/the-sting-framework-sf-ef00702c88c7, 2023.

[26] What are automated market makers (AMMs)? https://chain.link/education-hub/what-is-an-automated-market-maker-amm, 2023.

[27] Ittai Abraham, Lorenzo Alvisi, and Joseph Y Halpern. Distributed computing meets game theory: Combining insights from two fields. *ACM Sigact News*, 42(2):69–76, 2011.

[28] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2005.

[29] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, July 2011.

[30] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2018.

[31] Kenneth J Arrow. *Social choice and individual values*, volume 12. Yale University Press, 1951.

[32] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira. A fair consensus protocol for transaction ordering. In *Proceedings of the International Conference on Network Protocols (ICNP)*, 2018.

[33] David Austen-Smith and Jeffrey S Banks. *Positive political theory I: Collective preference*, volume 1. University of Michigan Press, 2000.

[34] John F Banzhaf III. Weighted voting doesn't work: A mathematical analysis. *Rutgers L. Rev.*, 19:317, 1964.

[35] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning: Limitations and Opportunities*. MIT Press, 2023.

[36] Johannes Behl, Tobias Distler, and Rudiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[37] Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D Procaccia. *Handbook of computational social choice*. Cambridge University Press, 2016.

[38] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In

Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2013.

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

[40] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 524–541, 2001.

[41] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, page 123–132, 2000.

[42] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 167–176, 2002.

[43] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[44] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, pages 236–269, 2003.

[45] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. Cryptology ePrint Archive, Report 2020/088, 2020.

[46] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[47] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[48] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine

faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[49] Marquis de Condorcet. Essay on the application of analysis to the probability of majority decisions. *Paris: Imprimerie Royale*, 1785.

[50] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless byzantine consensus and its application to blockchains. In *Proceedings of the International Symposium on Network Computing and Applications (NCA)*, 2018.

[51] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Proceedings of the International Financial Cryptography Conference*, 2019.

[52] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[53] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 2028–2041, 2018.

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

[55] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–7, 1983.

[56] David Galindo, Jia Liu, Mihair Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 88–102, 2021.

[57] Allan Gibbard. Manipulation of voting schemes: a general result. *Econometrica: Journal of the Econometric Society*, pages 587–601, 1973.

[58] Yossi Gilad, Rotem Hemo, Silvio M Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurren-

cies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[59] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable decentralized trust infrastructure for blockchains. arxiv:1804/01626v1, April 2018.

[60] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for Byzantine fault detection. In *Proceedings of the USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[61] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 2007.

[62] Joseph Y Halpern. *Reasoning about Uncertainty*. MIT Press, 2017.

[63] James Hendricks, Shafeeq Sinnamohideen, Gregory R Ganger, and Michael K Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 363–372, 2010.

[64] Maurice Herlihy and Mark Moir. Enhancing accountability and trust in distributed ledgers. *CoRR*, abs/1606.07490, 2016.

[65] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[66] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[67] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *Proceedings of the ACM on ASIA Public-Key Cryptography Workshop*, pages 3–14, 2022.

[68] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in Byzantine consensus. In *Pro-*

ceedings of the ACM Conference on Computer and Communications Security (CCS), page 475–489, 2023.

[69] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for Byzantine consensus. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[70] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2017.

[71] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the USENIX Security Symposium*, 2016.

[72] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[73] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 575–584, 2004.

[74] Robin SS Kramer. Sequential effects in Olympic synchronized diving scores. *Royal Society Open Science*, 4(1):160812, 2017.

[75] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[76] Leslie Lamport. Leaderless Byzantine Paxos. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 141–142, December 2011.

[77] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. In *Concurrency: the works of Leslie Lamport*, pages 203–226. 2019.

[78] Jonathan Lee, Kirill Nikitin, and Srinath Setty. Replicated state machines

without replicated execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[79] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2009.

[80] Michael Lewis. *Flash boys: A Wall Street revolt*. W. W. Norton & Company, 2014.

[81] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[82] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[83] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1), 2019.

[84] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–500, 2016.

[85] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous BFT consensus. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, page 264–280, 2023.

[86] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1988.

[88] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The

Honey Badger of BFT Protocols. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[89] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[90] John Nash. Non-cooperative games. *Annals of Mathematics*, pages 286–295, 1951.

[91] Alex Ozdemir, Riad S. Wahby, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the USENIX Security Symposium*, 2020.

[92] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Conference on Auctions, Market Mechanisms and Their Applications (AMMA)*, 2015.

[93] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[94] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 8:1–8:14, 2015.

[95] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 198–214, 2022.

[96] Tim Roughgarden. *Twenty lectures on algorithmic game theory*. Cambridge University Press, 2016.

[97] Bertrand Russell. *Am I an Atheist or an Agnostic?* 1950.

[98] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. CCF: A framework for building confidential verifiable replicated services. Technical report, Microsoft Research Technical Report MSR-TR-2019-16, 2019.

[99] Mark Allen Satterthwaite. Strategy-proofness and arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10(2):187–217, 1975.

[100] Leonard J Savage. *The Foundations of Statistics*. Courier Corporation, 1972.

[101] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[102] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[103] Srinath Setty, Sebastian Angel, and Jonathan Lee. Verifiable state machines: Proofs that untrusted services operate correctly. *ACM SIGOPS Operating Systems Review*, 54(1):40–46, August 2020.

[104] Lloyd S Shapley and Martin Shubik. A method for evaluating the distribution of power in a committee system. *American political science review*, 48(3):787–792, 1954.

[105] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[106] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[107] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the Ethereum blockchain. In *Proceedings of the USENIX Security Symposium*, pages 1343–1359, 2021.

[108] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1947.

[109] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault

tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.

[110] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[111] H Peyton Young. *Equity: In Theory and Practice*. Princeton University Press, 1995.

[112] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649, 2020.

[113] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 428–445, 2021.