Scaling the Unscalable: A Study About Consensus

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by

Ignacio Amores Sesar

from Spain

Supervisor of the doctoral thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Accepted by the Faculty of Science.

Bern,
16th January 2024

The Dean
Prof. Dr. Marco Herwegh

Scaling the Unscalable: A Study About Consensus

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by

Ignacio Amores Sesar

from Spain

Supervisor of the doctoral thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

La voz dormida.

Acknowledgments 

I wish to commence by extending my heartfelt gratitude to Christian, who transcended the role of an advisor and became an indispensable companion. Without your unwavering support and the wealth of knowledge you imparted, this journey would have remained beyond imagination.

My journey was made even more meaningful by the shared experiences with my colleagues and co-authors, moments that will forever resonate with me. Special thanks are due to Jovana, who not only excelled as a colleague but also became a cherished neighbor, and to Enrico, who felt like my own shadow. Matteo and abhi, too, welcomed me with open arms from the moment I set foot in Cambridge.

On a more personal note, I owe the person I am today to the incredible amount of love and support I received from my family, especially my parents, Mela and Juan, and my godmother, Conchi. My heartfelt thanks also go out to my childhood friends, Samu, Adri, and Sergio. Despite the occasional headaches we caused each other, I truely believe that I would not be the same without you.

High school was a pivotal stage in my life, where I crossed paths with exceptional individuals. Tomás, undoubtedly the finest teacher I have encountered, Cris, with whom I shared unforgettable experiences, and Alexis, a remarkable friend who supported me not only in our academic pursuits but also in life. During this time, I also met Javi, a language teacher from whom I learned much more than just English or German. It was during those memorable nights of gaming with Alexis that I met Miguel, Peñita, Sergi, and Gabi. Regardless of the physical distance,

you have always felt like close friends.

I also want to express my gratitude to all the wonderful people I had the privilege of meeting in Bern. Alex, with his unwavering positivity that brightened even the darkest days, and Yael, who never failed to offer a comforting cup of tea when I needed it most. Lastly, but most importantly, I want to extend my deepest appreciation to my partner, Klara. Your unwavering love and support over the past few years have brought me immeasurable joy, and words cannot adequately convey my appreciation for all you have done.

In closing, I would like to acknowledge all the other remarkable individuals who have played a significant role in my life, even if I regrettably could not mention each of you by name. Your presence and contributions have shaped my journey in profound ways.

Danksagung ()

Zu Beginn möchte ich Christian meinen herzlichsten Dank aussprechen. Er hat nicht nur seine Rolle als Betreuer übertroffen, sondern ist zu einem unverzichtbaren Begleiter geworden. *Ohni dini unerschütterlichi Unterstützig, und das Wüsse wo du mir vermittlet hesch, wäri die Reis undenkbar gsi.*

Meine Reise wurde noch bedeutungsvoller durch die geteilten Erlebnisse mit meinen Kollegen und Mitautoren, Momente, welche für immer in meiner Erinnerung bleiben werden. Ein besonderer Dank gebührt Jovana, die nicht nur eine exzellente Kollegin war, sondern auch zu einer geschätzten Nachbarin wurde, sowie Enrico, der immer wie ein treuer Schatten an meiner Seite stand. Matteo und abhi haben mich ebenfalls , ab dem Moment, als ich in Cambridge ankam, mit offenen Armen empfangen.

Ich persönlich verdanke meinen Werdegang zur Person, die ich heute bin, der unglaublichen Menge an Liebe und Unterstützung, die ich von meiner Familie erhalten habe, insbesondere von meinen Eltern, Mela und Juan, sowie meiner Patin, Conchi. Mein aufrichtiger Dank geht auch an meine Kindheitsfreunde, Samu, Adri und Sergio. Trotz der gelegentlichen Kopfschmerzen, die wir einander bereitet haben, bin ich fest davon überzeugt, dass ich ohne euch nicht derselbe wäre.

Die Schulzeit war eine entscheidende Phase in meinem Leben, in der ich auf außergewöhnliche Menschen gestoßen bin. Tomás, zweifellos der beste Lehrer, den ich je getroffen habe, Cris, mit der ich unvergessliche Erlebnisse geteilt habe, und Alexis, ein bemerkenswerter Freund, der mich nicht nur in unseren schulischen Belangen, sondern auch im Leben

unterstützt hat. Während dieser Zeit lernte ich auch Javi kennen, einen Sprachlehrer, von dem ich viel mehr als nur Englisch oder Deutsch lernte. Es war in diesen denkwürdigen Nächten des Spielens, dass ich Miguel, Peñita, Sergi und Gabi kennengelernt habe. Ungeachtet der räumlichen Entfernung wart ihr immer wie enge Freunde für mich.

Ich möchte auch meine Dankbarkeit gegenüber all den wunderbaren Menschen ausdrücken, die ich in Bern getroffen habe. Alex mit seiner unerschütterlichen Positivität, die selbst die dunkelsten Tage erhellt hat, und Yael, die mir immer dann eine beruhigende Tasse Tee angeboten hat, wenn ich sie am meisten brauchte. Am meisten möchte ich mich bei meiner Partnerin Klara bedanken. Ich schätze die unerschütterliche Liebe und Unterstützung, die du mir in den letzten Jahren geschenkt hast, extrem. Es fällt mir schwer, in Worten auszudrücken, was du für mich getan hast

Abschließend möchte ich alle anderen bemerkenswerten Personen würdigen, die zwar leider nicht namentlich erwähnt werden konnten, aber dennoch eine wichtige Rolle in meinem Leben gespielt haben. Eure Anwesenheit und Beiträge haben meinen Weg auf tiefgreifende Weise geprägt.

Agradecimientos ()

Quiero empezar dando las gracias de todo corazón a Christian, quien trascendió su papel de supervisor y se convirtió en un compañero indispensable. Sin tu apoyo inquebrantable y la riqueza de conocimiento que me transmitiste, este viaje habría sido inimaginable.

Mi viaje se enriqueció aún más gracias a las experiencias compartidas con vosotros, mis colegas y coautores; momentos que quedarán grabados en mi memoria para siempre. Quiero dar las gracias especialmente a Jovana, quien no solo destacó como colega, sino que también se convirtió en una querida vecina, y a Enrico, quien siempre estuvo a mi lado como mi propia sombra. Matteo y abhi quienes también me hicieron sentir como en casa desde el momento en que llegué a Cambridge.

En un plano más personal, debo mi la persona que soy hoy a la increíble cantidad de amor y apoyo que recibí de mi familia, especialmente de mis padres, Mela y Juan, y de mi madrina, Conchi. También quiero agradecer a mis amigos de la infancia, Samu, Adri y Sergio. A pesar de los quebraderos de cabeza que nos dimos en varias ocasiones, estoy seguro de que no sería la misma persona sin vosotros.

El instituto fue una etapa decisiva en mi vida, donde conocí a personas excepcionales. Tomás, sin duda el mejor profesor que he conocido, Cris, con quien compartí experiencias inolvidables, y Alexis, un amigo singular que me apoyó no solo en nuestros estudios, sino también en la vida. También fue en este momento cuando conocí a Javi, un profesor de idiomas que me enseño mucho mas que sólo inglés o alemán. Fue en esas noches memorables con el ordenador donde conocí a Miguel, Peñita, Sergi y Gabi. A pesar de la distancia física, siempre os sentí como

amigos cercanos.

También quiero expresar mi gratitud a todas las personas maravillosas que tuve el privilegio de conocer en Berna. Alex, con su positividad que iluminaba incluso los días más oscuros, y Yael, quien siempre ofrecía una taza de té reconfortante cuando más lo necesitaba. Por último, pero más importante, quiero dar las gracias a mi pareja, Klara. Estoy encantado por todo el amor y apoyo que me has brindado en los últimos años. Es imposible expresar en unas con palabras todo lo que has hecho por mí.

Para concluir, quiero agradecer a todas las demás personas que desempeñaron un papel en mi vida, a pesar de que lamentablemente no pude mencionaros a todos por nombre, habéis moldeado mi vida.

Agradecimientos ()

Quiero entamar dando-y les gracies más fonderes a Christian, que trescendió'l so papel de supervisor y convirtióse nun compañeru indispensable. Ensin el to apoyu inquebrantable y la bayura de conocimientu que me tresmitisti, esti viaxe nun sedría nin imaxinable.

El mio viaxe arriqueció más tovía coles esperiencies compartíes con vós, los mios collacios y coautores; momentos que van quedar grabaos nes mios alcordances pa siempre. Quiero dar les gracies sobre too a Jovana, que non solo destacó como collacia, sinón que tamién se convirtió nuna vecina mui querida, y a Enrico, que siempre tuvo al mio llau como la mio solombra. Matteo y abhi que tamién me ficieron sentir como en casa dende'l momentu nel qu'aporté a Cambridge.

Nun planu más personal, debo mi la persona que soi anguaño a la cantidá increíble d'amor y sofitu que recibí de la mio familia, sobremanera de mio madre y de mio padre, Mela y Juan, y de la mio madrina, Conchi. Tamién quiero agradecer a los mios amigos de cuando neños, Samu, Adri y Sergio. Pesie a los enguedeyos y esmolecimientos que tuviemos dacuando, toi seguru de que nun sedría la mesma persona ensin vós.

L'institutu foi una etapa decisiva na mio vida, ellí conocí a persones escepcionales. Tomás, ensin dulda'l meyor profesor que conocí; Cris, cola que compartí esperiencies inolvidables, y Alexis, un amigu singular que m'apoyó non solo nos estudios, sinón tamién na vida. amién foi nesti momentu cuando conocí a *Javi, un profesor d'idiomes qu'enseño enforma mas que namá inglés o alemán. Foi neses nueches pal recuerdu, col ordenador, onde conocí a Miguel, Peñita, Sergi y Gabi. Pesie a la distancia física, siempre vos sentí y vos tuvi por amigos cercanos.

Tamién quiero espresar la mio gratitú a toles persones maraviyoses que tuvi'l privilexu de conocer en Berna. Alex, con esa positividá qu'allumaba inclusive los díes más escuros, y Yael, que siempre m'ofrecía una taza de té reconfortante cuando más falta me facía. A lo último, pero más importante, quiero dar les gracies a la mio pareya, Klara. Toi encantáu con tol amor y sofitu que m'apurristi nestos años últimos . Abúltame imposible espresar con cuatro lletres tolo que ficisti por min.

Pa rematar, quiero agradecer a toles demás persones que desempeñaron un papel na mio vida, magar que por desgracia nun pudi mencionavos a toes y toos pol nome, moldeastis la mio esistencia.

Abstract

Consensus protocols form the bedrock of various distributed systems integral to modern life, ranging from basic *clock synchronization* to sophisticated *blockchains*. Yet, the proliferation of consensus protocols reveals a fundamental limitation: their scalability.

In centralized systems, boosting performance can often be achieved with the addition of more participants. However, in decentralized systems, this approach can be counterproductive. The delicate equilibrium between *safety* and *liveness* becomes more restricting as the number of participants increases, especially in the permissionless setting, due to fortifications against *sybil attacks*. This thesis endeavors to make a contribution towards scaling permissionless protocols in a vast landscape of efforts currently addressing the topic.

Commencing with an exhaustive examination of *Nakamoto* consensus and an attempt to address throughput and latency constraints via *GHOST*, we establish a unified model for both protocols. This model manifests the intricate interplay between safety, liveness, and performance, paving the way for a family of protocols that arbitrarily approximate the performance of GHOST while remaining resilient against balance attacks, a primary vulnerability of GHOST. Nevertheless, the scope for improvement within the Nakamoto-GHOST paradigm remains constrained by the limitations of GHOST.

Avalanche boosts throughput orders of magnitude higher than Nakamoto and GHOST while maintaining latency in the order of seconds, employing a directed acyclic graph (DAG) instead of a chain. Despite its impressive performance, formal analyses of its safety and

liveness were absent, except for the work encompassed in this thesis. Our deep analysis of Avalanche consensus reveals a significant liveness vulnerability, prompting us to enhance its mechanism with *Glacier* without compromising performance.

DAG protocols have revolutionized consensus protocols in the permissioned setting in recent years. These protocols achieve remarkable throughput but carry an increase in latency. In this work, we introduce an atomic broadcast protocol that continues this line of work but achieves latency similar to leader-based protocols.

These studies serve as inspiration for further results in this thesis. Leveraging techniques used to address consensus protocol performance, we devise a construction that mitigates *sandwich attacks* in longest-chain consensus protocols. Additionally, our exploration of Avalanche, alongside Conflux, a less familiar protocol, lays the groundwork for the last contribution in this thesis. We craft another construction that transforms a blockchain protocol into a DAG protocol, proving formally that for every blockchain protocol, a corresponding DAG protocol exists that achieves higher throughput, similar or lower latency, and maintains safety and liveness under the same assumptions. Furthermore, this construction allows to determine a set of protocols with the potential to be optimal. Furthermore, the atomic broadcast protocol introduced in this work falls in this category.

Contents

Chapter 1

Introduction

> It's the job that's never started as takes longest to finish.
>
> Samwise Gamgee

The blockchain and distributed ledger technology landscape has experienced significant growth and innovation in the past decade, spurred by the pursuit of systems offering enhanced throughput, reduced latency, and improved scalability while upholding the essential tenets of security and decentralization. The need for more comprehensive investigations becomes particularly pronounced within the realm of permissionless protocols, where constraints related to throughput and latency pose more significant challenges. As the evolution of cryptocurrencies and blockchain networks continues, researchers have embarked on explorations of various consensus protocols and structural frameworks to tackle these multifaceted challenges.

The genesis of this journey can be traced back to the moment when Nakamoto unveiled the Bitcoin protocol [88], laying the foundation for a permissionless consensus protocol that implements a robust decentralized payment system. Since that pivotal moment, a plethora of alternative protocols have emerged, each with the objective of enhancing Nakamoto consensus. At the heart of these decentralized payment

systems lies the maintenance of a distributed data structure known as the *blockchain*, maintained by participants often referred to as *miners*. Transactions are grouped into blocks and subsequently appended to the blockchain when specific conditions are fulfilled.

Most improvements to Nakamoto consensus are geared toward augmenting throughput without degrading security since Nakamoto consensus is severely limited in this sense [43]. One notable example is the GHOST protocol [104], which enables off-chain blocks to contribute to security by considering subtrees of blocks, in contrast to Nakamoto consensus, which relies only on chains. Thus allowing *forked* blocks, blocks with the same *parent* block, to enhance the security of their parent. Nevertheless, the approach of GHOST does not take into account the structure of blocks within a subtree, introducing a vulnerability to consensus, which can be exploited by an adversary with strong influence over the network, as exemplified in a balance attack [89].

Whenever a miner produces a block, the miner places it in some position with respect to the previously produced blocks (by including their hashes in the new block). Thus, a protocol execution constructs a tree, in which every node is a block b_i and an edge (b_i, b_j) denotes that b_i includes the hash of b_j. This tree can be used to understand the placement of newly mined blocks in Nakamoto consensus and in GHOST within a common framework. The chain that the miners extend is called the *main chain*. The key difference between such protocols lies in the way this main chain is selected. Nakamoto selects the longest chain, whereas GHOST selects a chain in which each block has the biggest subtree among its forked blocks. The chain selection of Nakamoto does not consider the number of blocks in each subtree, and GHOST does not consider the depth of each subtree. This common framework allows the study of chain-selection rules in which both the size and depth of the subtrees influence the chain selection. These rules can potentially obtain the performance of GHOST while maintaining the robustness of Nakamoto.

Amidst this ever-evolving environment, Avalanche [99] emerges as one of the alternatives to first-generation networks, such as those built on Nakamoto consensus (Bitcoin) or variants of GHOST (Ethereum). Avalanche offers a consensus protocol renowned for its exceptional speed and scalability, delivering high throughput, low latency, and a lightweight client. Unlike many established distributed ledgers, Avalanche departs from the proof of work paradigm, opting for a de-

liberately metastable mechanism that operates by recurrently sampling the network, guiding honest participants toward a common output. This unique approach enables Avalanche to achieve peak throughput of up to 20,000 transactions per second with a latency of less than half a second [99].

This novel mechanism, however, imposes more stringent security constraints on Avalanche compared to other networks. While traditional Byzantine fault-tolerant consensus tolerates corruption of up to a third of the participants [91], and proof-of-work protocols tolerate similar corruptions [56, 54], Avalanche can withstand only up to the square root of the number of participants behaving maliciously. Furthermore, transactions in Avalanche do not exhibit total ordering, setting it apart from most other cryptocurrencies that implement a form of *atomic broadcast* [27]. The structure of the protocol, organized around a *DAG* instead of a linear chain, introduces the potential for parallelism, and corresponding throughput improvement. Understanding the Avalanche consensus holds a dual significance; it not only guarantees the security of the protocol but also potentially unlocks a new dimension of scalability for permissionless protocols.

Furthermore, DAG protocols have demonstrated surprising promise in the permissioned setting. Traditional consensus protocols, exemplified by Paxos [91], PBFT [33], or Hotstuff [111], primarily follow a *leader-based* approach. In this model, a single party proposes a value, and the remaining participants validate this proposal. Consequently, the workload of the protocol becomes unevenly distributed among the participants, making the leader for a given instance a potential bottleneck. In contrast, Keidar et al. [63] initiated a line a work [45, 105, 64], allowing every party to create blocks forming a DAG, which is later ordered. DAG protocols have exhibited remarkable performance improvements in various implementations [45, 105]. The main issue of these protocols lies in their high latency when compared to traditional leader-based protocols.

A similar situation occurs in the permissionless setting, mostly proof of work, where every party is allowed to create a block. However, when blocks fork, all the computational resources spent in the blocks that do not end up in the main chain are wasted. Thus, DAG protocols could also revolutionize the permissionless setting.

Yet, it remains an open question whether the DAG approach represents the optimal direction for enhancing the performance of permissionless protocols. This very question serves as the central motivation behind this work, a question we aim to address comprehensively in the pages that follow.

Contributions

This dissertation is a compendium of five distinct works, four of them delving into diverse facets of blockchain scalability, with the fifth work adapting and extending the techniques unveiled in the preceding works to the prevention of *sandwich attacks*, a prevalent category of maximal extractable value attacks. These works contribute to the evolving landscape of blockchain technology and enhance the security and efficiency of blockchain networks.

In this overarching introduction, we provide an overview of the key themes and contributions of these works, drawing connections and highlighting the significance of the research they collectively represent.

Medium: A bridge from Nakamoto to GHOST [9]. The first work introduces the *Medium Protocol*, a consensus mechanism that takes inspiration from both Nakamoto consensus and the GHOST protocol. It leverages the structure of the block tree to create a weight-based approach for selecting the main chain. This approach generalizes the principles of Nakamoto consensus and GHOST, allowing for a fine-tuned balance between security and throughput. The work provides a detailed analysis of Medium's security properties, demonstrating its resilience against various attacks on consensus. By evaluating weight coefficients, the study illustrates how Medium can adapt to different network conditions while maintaining robust security and an improved understanding of the trade-offs between security and throughput in blockchain networks.

An analysis of Avalanche consensus [10]. The second work explores the Avalanche blockchain, an innovative and energy-efficient

alternative to first-generation protocols like Bitcoin and Ethereum. Avalanche distinguishes itself by its fast and scalable consensus protocol that forgoes the resource-intensive proof-of-work mechanism in favor of a deliberately metastable mechanism based on a directed acyclic graph (DAG). This mechanism repeatedly samples the network, guiding honest participants toward a common output, ensuring high throughput and low latency. However, Avalanche's unique approach imposes stricter security constraints, tolerating only a limited number of malicious parties. Notably, this work addresses the security and liveness of the Avalanche protocol, highlighting a vulnerability related to transaction dependencies. The study identifies a weakness that could potentially be exploited to delay transactions significantly, rendering the protocol impractical in real-world scenarios. To mitigate this vulnerability, the work suggests a modification known as *Glacier*.

DAG superiority [7]. The third work digs deeper into the potential of DAG consensus protocols, building upon the foundational idea that these protocols can significantly enhance the throughput and latency of blockchain networks. Traditional consensus protocols, represented by chain-based systems like Bitcoin, have made substantial progress in terms of throughput and latency. However, they face inherent limitations when block production rates increase. The work introduces a novel construction that takes a DAG or chain-based protocol and transforms it into a new DAG protocol while ensuring that every created block is eventually included in the blockchain. This transformation enhances throughput and latency without compromising the protocol's safety or liveness. The study offers a comprehensive analysis of the proposed construction and demonstrates that it provides an elegant improvement of chain-based protocols, showing that DAG protocols achieve better performance under the same assumption.

An atomic broadcast protocol[1]. The fourth work addresses a key issue with DAG protocols in the permissioned context: latency. The latency is primarily attributed to the usage of reliable broadcast primitive for the common core functionality. Cordial miners [64], reduced

[1]This work has been done during my internship at Google Cambridge under the supervision of abhi shelat and Matteo Frigo.

this latency by eliminating the broadcast requirement. However, Keidar et al. [64] still utilized the common core primitive, thereby limiting the reduction in latency. Our work introduces a protocol that employs a weaker form of the common core, eliminating the need for reliable broadcast. This weaker version proves sufficient for implementing atomic broadcast, thanks to the properties of the common coin. To the best of our knowledge, our novel protocol achieves the lowest best-case latency among all DAG protocols based on the common core while retaining their throughput characteristics.

Sandwich-attack prevention [6]. The fifth work tackles the critical problem of sandwich attacks, which exploit transaction order control within a block by miners. It presents a novel construction that takes an existing blockchain protocol as input and outputs a new blockchain protocol with identical security properties yet effectively mitigates the profitability of sandwich attacks. This construction accomplishes its goal by removing the miner's full control over transaction order within a block. Importantly, this approach is entirely decentralized and does not rely on trusted third parties or heavy cryptographic primitives. As a result, it introduces only a modest linear increase in latency and minimal computational overhead. The techniques employed in this work draw inspiration from various approaches developed previously to address scalability issues in consensus protocols.

The unifying thread. Collectively, these works contribute to the evolving field of blockchain technology by addressing fundamental challenges associated with consensus, throughput, and security. While the analysis of Avalanche, the DAG study, and the Medium Protocol each tackle different aspects of these challenges, they share a common objective: to improve the efficiency, scalability, and robustness of blockchain networks, culminating with an atomic broadcast protocol. This dissertation serves as a bridge between the theoretical underpinnings and practical implications of blockchain technology, offering a deeper understanding of the complex dynamics at play in these systems.

In the following chapters, we delve into each of these works in detail, presenting their methodologies, findings, and contributions. Through this exploration, we aim to obtain important insight into the evolving

landscape of blockchain technology and its critical role in shaping the future of decentralized systems.

Chapter 2

Preliminaries

> It was merely the substitution of
> one piece of nonsense for another.
>
> Ministry of Plenty's figures

Most of the work described in this thesis has been built upon a common set of primitives. This chapter captures these primitives. We start by describing the adversarial and communication models. Secondly, we introduce a set of primitives used in this thesis.

2.1 Parties and adversary

We consider a set of n parties, $\mathcal{P} = \{P_1, ..., P_n\}$ running a protocol Π that exchange messages through a *network*. In this work, we consider two different adversarial models and three network assumptions.

Byzantine model. In the Byzantine setting, parties are modeled as *interactive Turing machines* (*ITM*). An interactive Turing machine is a Turing machine with an input and an output tape that allows the

Turing machine to communicate with other Turing machines and make decisions based on the content of their input tape. The adversary is modeled as another ITM that corrupts up to f parties at the beginning of the execution. These corrupted parties obey the adversary; in other words, they may diverge from the normal execution of the protocol. Corrupted parties are often referred to as *Byzantine* and non-corrupted as *honest*.

Rational model. For $N \in \mathbb{N}$, we consider the N party game $\Gamma = (N, (S_i), (u_i))$ where S_i is a finite set of strategies for each party $i \in [N]$. Let $S := S_1 \times \cdots \times S_N$ denote the set of outcomes of the game. The utility function of each party i, $u_i : S \to \mathbb{R}$, evaluates the payoff of party i given an outcome of Γ. In this context, a *mixed strategy* for any party i is a probability distribution in $\mu(S_i)$. A *strategy profile* for Γ is $s := s_1 \times \cdots \times s_N$ where s_i is a mixed strategy of party i. The *expected utility* for a party i with respect to the mixed strategy profile s is defined as $u_i(s) = \mathbb{E}_{a_1 \leftarrow s_1, \cdots, a_N \leftarrow s_N}[u_i(a_1), \cdots, u_i(a_N)]$. In the rational model, we model the parties as agents behaving in a way that optimizes their expected utility. This model is used and explained in further detail in Chapter 7.

2.2 Communication models

In different chapters, we consider different network assumptions based on the nature of the respective protocols. We consider *synchronous-rounds* in the majority of the work presented here, an asynchronous model in Chapter 6, and an exponential delay model in Chapter 4.

2.2.1 Synchronous-rounds model

In this model, communication among the parties is implemented by a *diffusion functionality*, which is structured into *synchronous* rounds. The functionality keeps a distinct $RECEIVE_i$ string for each party P_i and makes it available to P_i at the start of every round. The purpose of the

string $RECEIVE_i$ is to serve as a repository for all the messages received by P_i.

When a party, say P_i, instructs the diffusion functionality to *BROADCAST* a set of messages, it signifies that P_i has *completed its round*. In response, the functionality marks P_i as having completed its operations for that specific round. The adversary, whose actions are described in detail below, possesses the ability to access the string of any party at any point during the execution. Additionally, the adversary can observe every message broadcast by any party instantaneously. Furthermore, the adversary has the capability to insert messages directly and selectively into $RECEIVE_i$ for any party P_i, ensuring that only P_i receives the message at the outset of the following round. This behavior models what is often termed a *rushing* adversary.

Once all non-corrupted parties have concluded their respective rounds, the diffusion functionality aggregates all messages that were broadcast by non-corrupted parties during that round. These aggregated messages are then appended to the $RECEIVE_i$ strings for all parties; this is the reason for the name *synchronous* rounds. Subsequently, each non-corrupted party updates its local view at the conclusion of every round. If a non-corrupted party sends a message in round r, all parties receive the message by round $r + 1$.

Furthermore, even if the adversary causes a message to be received selectively by only some non-corrupted parties in round r, the message is received by all non-corrupted parties by round $r + 2$. The update of the local view may also encompass the output of events when given criteria defined by the protocol are fulfilled.

2.2.2 Asynchronous model

In this model, communication among parties is again facilitated by a *diffusion functionality* under the control of an adversary. The functionality maintains a separate $RECEIVE_i$ string for each party P_i, accessible to parties at any point. When party P_i requests the diffusion functionality to *BROADCAST* a set of messages, the messages are stored. The adversary can read the stored messages and schedule their delivery *at will*, with the only constraint being that all messages must be delivered

within a finite time frame, in contrast with the model defined in Section 2.2.1. Additionally, the adversary can deliver a broadcast message to different parties at different points in time. There is no concept of rounds in this model.

2.2.3 Exponential-delay model

In this model, parties may access a low-level functionality for sending messages over authenticated point-to-point links between each pair of parties. In the protocol, this functionality is accessed through two events *send* and *receive*. Parties may also access a second low-level functionality for broadcasting messages through the network by gossiping, accessed by the two events *gossip* and *hear* in the protocol. Both primitives are subject to network and timing assumptions. Messages are delivered according to an exponential distribution; that is, the amount of time between the sending and the receiving of a message follows an exponential distribution with unknown parameter to the parties. However, messages from corrupted parties are not affected by this delay and will be delivered as fast as the adversary decides. This model differs from the traditional definition of *partial synchrony* [50] since the adversary does not possess the ability to influence the delay of honest messages.

2.3 Abstractions

Broadcast primitives are a fundamental building block in the protocols designed nowadays. In this section, we formalize these abstractions, as well as their interfaces. Our focus lies mostly on non-deterministic protocols; thus, the properties of all the abstractions introduced here are satisfied with all but negligible probability. Furthermore, we restrict the set of messages to transactions that are submitted by a set of *users*. However, since we do not impose restrictions on the content of transactions, this does not constitute a loss of generality.

2.3.1 Notation

The abstractions presented in this work interact through specific application programming interfaces. These interfaces are identified by a prefix denoting the particular abstraction. For instance a protocol implementing *reliable broadcast* contains a prefix *rb-* in the events of its interface.

In situations where multiple protocols, such as Π and Π', implementing the same abstraction coexist within the same context, we replace the prefix with the name of the protocol to avoid confusion, using prefixes such as Π- or Π'-. When the context is evident, we omit the prefix to lighten the notation.

2.3.2 Reliable broadcast

The *reliable broadcast* primitive is often used when parties want to *broadcast* transactions among each other. This primitive guarantees that every honest party delivers a transaction only if the other honest parties *eventually* deliver the transaction. However, the order in which every honest party delivers them may differ.

Our reliable broadcast primitive is accessed through the events *rb-broadcast* and *rb-deliver* and is equipped with an "external" validity predicate V that determines whether a transaction is valid [28].

Definition 1 (Reliable broadcast). A protocol solves *reliable broadcast* with validity predicate V if it satisfies the following conditions, except with negligible probability:

Validity: If a honest party *rb-broadcasts* a transaction tx, then it eventually *rb-delivers* tx.

Agreement: If a honest party *rb-delivers* a transaction tx, then all honest parties eventually *rb-deliver* tx.

Integrity: For any transaction tx, every honest party *rb-delivers* tx at most once, and only if tx was submitted by some user.

External validity: If an honest party *ab-delivers* a transaction tx, then $V(tx) = \text{TRUE}$.

A well-known algorithm implementing reliable broadcast with only three rounds of communication is Bracha broadcast [25]. The applications of reliable broadcast are limited by the lack of order in the delivered transactions.

2.3.3 Atomic broadcast

Atomic broadcast is an enhancement of reliable broadcast (Definition 1) in which every honest party delivers the transactions in the exact same order. This is crucial feature is needed in scenarios in which different order of delivery of transactions produces different results.

Our atomic broadcast primitive is accessed through the events *ab-broadcast* and *ab-deliver* and is equipped with an "external" validity predicate V that determines whether a block is valid.

Definition 2 (Atomic broadcast). A protocol solves *atomic broadcast* with validity predicate V if it satisfies the following conditions, except with negligible probability:

Validity: If a honest party *ab-broadcasts* a transaction tx, then it eventually *ab-delivers* tx.

Agreement: If a honest party *ab-delivers* a transaction tx, then all honest parties eventually *ab-deliver* tx.

Integrity: For any transaction tx, every honest party *ab-delivers* tx at most once, and only if it was submitted by some user.

Total order: If honest parties P_i and P_j both *ab-deliver* transactions tx and tx', then p *ab-delivers* tx before tx' if and only if P_j *ab-delivers* tx before tx'.

External validity: If an honest party *ab-delivers* a transaction tx, then $V(tx) = \text{TRUE}$.

It is worth mentioning that this primitive is equivalent to consensus [27]. In other words, an atomic broadcast protocol is sufficient to implement consensus, and conversely, a consensus protocol is also sufficient to implement atomic broadcast.

2.3.4 Generic Broadcast

Enforcing a total order of transactions can potentially negatively impact the performance of the protocol. Moreover, in certain scenarios, pairs of transactions do not require ordering, as their outcomes remain the same regardless of the order of delivery. The *generic broadcast* abstraction [93], defined by an equivalence relationship on the set of transactions, ensures a partial order of delivered transactions. When two transactions are related, an honest party delivers them in the same order. However, if they are not related, honest parties may deliver them in different orders.

Our generic broadcast primitive is accessed through the events *gb-broadcast*(tx) and *gb-deliver*(tx). Similar to other primitives, it defines an "external" validity property and introduces a predicate V that determines whether a transaction is valid.

Definition 3 (Generic broadcast). A protocol solves *generic broadcast* with validity predicate V and relation $\sim$ if it satisfies the following conditions, except with negligible probability:

Validity: If a honest party *g-broadcasts* a transaction tx, then it eventually *g-delivers*(tx).

Agreement: If a honest party *delivers* a transaction tx, then all honest parties eventually *deliver*(tx).

Integrity: For any transaction tx, every honest party *delivers*(tx) at most once, and only if it was submitted by some user.

Partial order: If honest parties P_i and Q_i both *deliver* transactions tx and tx' such that $tx \sim tx'$, then P_i *delivers*(tx) before it *delivers*(tx') if and only if P_j *delivers*(tx) before it *delivers*(tx').

External validity: If a honest party *delivers* a transaction tx, then $V(tx) = \text{TRUE}$.

Note that different instantiations of the relation $\sim$ transform the generic broadcast primitive into well-known primitives. For instance, when no pair of transactions are related, generic broadcast degenerates to reliable broadcast (Definition 1). Whereas when every two transactions are related, a generic broadcast transforms into an atomic broadcast (Definition 2).

2.3.5 Block-based atomic broadcast

In this work, we consider a variant of atomic broadcast (Definition 2) that includes the concept of a *block* in the interface and properties [6]. Parties broadcast transactions and deliver blocks using the events *bab-broadcast*(tx) and *bab-deliver*(b), respectively, where block b contains a sequence of transactions $[tx_1, \ldots, tx_m]$. The protocol outputs an additional event *bab-mined*(b, P), which signals that block b has been *mined* by party P_i, where P_i is defined as the *miner* of b. The event *bab-mined*(b, P) can be understood as the creation of block b by party P_i. Notice that *bab-mined*(b, P) signals only the creation of a block and not its delivery. In addition to predicate $VT()$ that determines the validity of a transaction, we also equip our protocol with a validity predicate $VB()$ to be applied to blocks. These predicates and functions are determined by the higher-level application or protocol.

Definition 4 (Block-based atomic broadcast). A protocol implements *block-based atomic broadcast* with validity predicates $VT()$ and $VB()$ and block-creation function $FB()$ if it satisfies the following properties, except with negligible probability:

Validity: If a correct party invokes a *bab-broadcast*(tx), then every correct party eventually outputs *bab-deliver*(b), for some block b that contains tx.

No duplication: No correct party outputs *bab-deliver*(b) for a block b more than once.

Integrity: If a correct party outputs *bab-deliver*(b), then it has previously output the event *bab-mined*($b, \cdot$) exactly once.

Agreement: If some correct party outputs *bab-deliver*(b), then eventually every correct party outputs *bab-deliver*(b).

Total order: Let b and b' be blocks, and P_i and P_j correct parties that output *bab-deliver*(b) and *bab-deliver*(b'). If P_i delivers b before b', then P_j also delivers b before b'.

External validity: If a correct party outputs *bab-deliver*(b), such that $b = [tx_1, \ldots, tx_m]$, then $VB(b) = \text{TRUE}$ and $VT(tx_i) = \text{TRUE}$, for $i \in 1, \ldots, m$. Moreover, if $FB(tx_1, \ldots, tx_m)$ returns b, then $VB(b) = \text{TRUE}$.

The block-based atomic broadcast primitive can be equipped with a

fairness property to guarantee that a potential adversary cannot insert an arbitrary number of consecutive blocks in the output.

Definition 5 (Fairness). A *block-based atomic broadcast* protocol is *fair* if it satisfies the following property, except with negligible probability:

Fairness: There exists $C \in \mathbb{N}$ and $\mu \in \mathbb{R}_{>0}$, such that for all $N \geq C$ consecutive delivered blocks, the fraction of the blocks whose miner is correct is at least μ.

The block-based atomic broadcast primitive serves as a fundamental model for protocols like blockchain protocols. These protocols prominently feature the concept of blocks as a core component. Specifically, we leverage this primitive when crafting constructions that alter the block content of a given blockchain protocol.

2.3.6 Common coin

The *common coin* primitive [27] can be understood as a weaker form of consensus [27]. A common coin allows parties to agree on a random value from an already predefined set of values with a given probability. Our common coin abstraction is accessed through the events *c-release*() and *c-output*().

Definition 6 (Common coin). A protocol solves *common coin* with domain $\mathcal{D}$ and bias ε if it satisfies the following conditions, except with negligible probability:

Termination: Every correct party eventually *c-output*() a coin value.

Unpredictability: The probability that an adversary predicts the *c-output*() value before at least one honest party invokes *c-release*() is at most $\frac{1}{|\mathcal{D}|} + \varepsilon$.

Matching: With probability at least δ, every correct party *c-outputs* the same value. If $\delta = 1$, the coin is called *perfect*

No bias: If all correct parties *c-output*() the value, the distribution of the coin is uniform over $\mathcal{D}$.

Note that the common coin primitive is not the same as consensus, as the output of the coin may not be a proposed value. However, one of the most prominent applications of a common-coin protocol is to implement consensus or atomic broadcast (Definition 2) in asynchrony, circumventing the FLP-Impossibility [55].

In this work we assume perfect coins with ε arbitrarily small. Such coins can be constructed efficiently in a distributed setting [106].

2.3.7 Common core

The *common core* is another weak form of consensus that has been introduced by Canetti [32]. This primitive has been recently reconsidered as building block for DAG-based consensus protocols [63, 45, 105, 64]. In this abstraction, every party has a value (P_i, v_i) as input and produces an output set U_i consisting of pairs of input values (P_j, v_j) with the condition that the output set of all the honest parties intersects contains a set U^* of at least $2f + 1$ different elements.

Our common core abstraction is accessed through the events *core-broadcast*(v) and *core-deliver*(U). Every party broadcasts its input value at the beginning.

Definition 7 (Common core). A protocol solves *common core* if it satisfies the following conditions with all but negligible probability:

Validity: Every honest party P_i eventually *delivers* a set U_i.

Common core: There exists a core set U^* of size at least $2f + 1$ that is included in the *delivered* set of every honest party.

Integrity: If honest party P_i includes (P_j, v_j) in its *delivered* set U_i, and j is honest, then v_j its input.

Agreement: If two honest parties include the pairs (P_j, v) and (P_j, v') in their *delivered* sets, then $v = v'$.

In the common core, honest parties agree on a common set that is part of their outputs. However, the common core is a weaker primitive than consensus [27] or atomic broadcast (Definition 2) because the core set is

unknown to the parties. In particular, the common core primitive only guarantees its existence.

The gather protocol [32] is a deterministic protocol implementing the common core in four rounds of communication. This algorithm is upgraded with a common coin (Definition 6) by Keidar et.al. [63] starting a new line of DAG-based protocols.

Chapter 3

Medium: A bridge from Nakamoto to GHOST

> It is so easy to remember our differences, Par'chin, I sometimes forget the similarities.
>
> Ahmann Jardir

3.1 Introduction

Since Nakamoto revealed the Bitcoin protocol [88] as a blueprint for a decentralized payment system, many other protocols have been introduced with the goal of improving Nakamoto consensus. The basic principle of these decentralized payment systems is that a distributed data structure, called the blockchain, is maintained by parties (also called *miners*) that run a distributed protocol. Transactions are grouped into blocks, which are later added to the blockchain when specific properties have been fulfilled. Most improvements to Nakamoto consensus aim at processing more transactions and achieving higher throughput

without degrading security because Nakamoto consensus is severely limited in this sense [43]. The GHOST protocol [104], for example, lets all mined blocks contribute to the security by considering subtrees of blocks, whereas Nakamoto consensus relies only on the blocks in the longest chain. GHOST, however, does not take into consideration how the blocks are structured and counts *all* blocks in a subtree in the same way. This introduces a potential vulnerability to consensus, which can be exploited by an adversary with strong influence over the network, as exemplified in a balance attack [89].

Whenever a miner produces a block, the miner places it in some position with respect to the previously produced blocks (by including their hashes in the new block). Thus, a protocol execution constructs a tree, in which every node is a block b_i and an edge (b_i, b_j) denotes that b_i includes the hash of b_j. This tree can be used to understand the placement of newly mined blocks in Nakamoto consensus and in GHOST within a common framework. The chain that the miners extend is called the *main chain*. The key difference between such protocols lies the way how this main chain is selected.

As a miner in Nakamoto consensus always selects the longest chain in the tree (technically, the one with the most work, but we ignore this subtlety here) and extends this chain by one block. The security relies intuitively on the rule that only the longest chain grows unless two parties mine concurrently and thereby create a fork. This may happen when a party mines without receiving the last block mined before. Forks limit the throughput of a network, and they typically occur more often when the block production rate increases compared to the message delay in the network.

On the other hand, GHOST determines the main chain by extracting more information from the tree. Starting from the genesis block, it iteratively selects the block with the heaviest subtree (defined by the number of blocks in the subtree of the block) until it reaches a leaf block. When a miner produces a new block, it appends this to the last block selected by this rule. The intuition is that also forked blocks (and their miners) contribute to the security of the blocks they point to. However, all blocks are counted in the same way regardless of their position in the subtree. This actually loses considerable information about the tree structure and may introduce vulnerabilities.

In this chapter, we introduce the *Medium protocol*[1] that takes into account the structure of the block tree in a way that generalizes both Nakamoto consensus and GHOST. Medium computes a *weight* for a subtree using a *polynomial* in a *weight coefficient* c, which determines the influence of the tree structure on chain selection. This results in a family of Medium protocols, each one uniquely defined by some c.

Specifically, we introduce a weight function

$$\omega : \mathcal{B} \times \mathcal{T} \to \mathbb{R}_{>0} \tag{3.1}$$

for a block $B \in \mathcal{B}$ in a tree $T \in \mathcal{T}$, defined by $\omega(b, T) = c^{d(b)}$, where $d(b)$ denotes the depth of b in T and $c \geq 1$. The selection rule of GHOST can be interpreted as the particular case of $c = 1$ (up to the way of breaking ties for trees with equal weight), and Nakamoto consensus results in the limit for $c \to \infty$. Thus, Medium generalizes GHOST and Nakamoto consensus, so that they can be compared in a comprehensive way to all protocols in the Medium family.

The weight function intuitively takes up the idea behind GHOST that every block contributes to the security and combines it with Nakamoto consensus' feature that deeper blocks are more relevant. Thus, forked blocks also influence the main chain selection process, but longer chains are still more desirable.

The weight coefficient determines the extent to which forks contribute to main chain selection in relation to the contribution of chain length.

We show that Medium is secure against well-known attacks on GHOST. In particular, a balance attack always fails after a finite number of rounds. We show that protocols with larger weight coefficients are, in general, safer from attacks, but may have lower throughput. There is, thus a continuum of weight coefficient values, leading to the ability to find a protocol with optimized throughput and safety, depending on the network and the user's requirements.

To analyze the security of Medium, we adopt the model of Kiayias and Panagiotakos [69], which allows us to prove security against attacks on consensus, such as double spending [88], block withholding [54], and

[1] Medium, in occultism, a person reputedly able to make contact with the world of spirits, especially while in a state of trance [52].

eclipse [60]. Specifically, we prove that the Medium protocol family satisfies three main properties in a synchronous network. Firstly, the *weight and length* of the main chain *increase* over time. This means that the protocol is live, adding ever more transactions to the blockchain, and also that the cost of reverting past transactions increases with time. Secondly, the main chain *contains* at least a *fraction of honest blocks*, i.e., blocks not mined by the adversary. This ensures that transactions of honest parties are eventually added to the main chain and executed. Lastly, the main chain of all the honest parties contains a *common prefix* that increases over time. This means that once a transaction has been in the main chain for long enough, it remains in the main chain. We use these properties to ultimately construct a decentralized payment system, where the blockchain is a robust public transaction ledger, following the notions of Kiayias *et al.* [56, 69].

The results illustrate how Medium forms a bridge between Nakamoto consensus and GHOST, allowing a deeper understanding of them; Medium can also improve other constructions that rely on Nakamoto consensus or GHOST.

Acknowledgement. The material contained in this chapter corresponds to the work 'Generalizing weighted trees: a bridge from Bitcoin to GHOST' [9] published at AFT21.

3.2 Related work

Garay *et al.*'s Bitcoin Backbone [56] is the first in-depth formalization of the Nakamoto consensus and represents an important step for understanding the security of blockchains. They analyze the protocol in synchronous and in partially synchronous networks. Kiayias and Panagiotakos [69] expand the model and demonstrate the security of Nakamoto consensus and GHOST against a variety of attacks.

In these security models, the adversary has only limited capability to prevent communication between honest parties. For instance, in the analysis of the eclipse attack [69], the adversary may only control the communication between a fraction of the miners. More powerful attacks,

however, could split the network in two and prevent any exchange between the parts. Such attacks threaten the security of Nakamoto consensus and have even more severe consequences for GHOST. In particular, Natoli and Gramoli [89] point out this issue under the name of a *balance attack*. Bagaria *et al.* [16] show that such an attack on GHOST can perpetuate a fork indefinitely, leading to miners splitting their power between the two sides of the fork and the network never reaching consensus. The difference between these attacks is that Bagaria *et al.* [16] assume the adversary has the ability to partition the network for a given amount of time. It is exactly such an attack that we aim to prevent by choosing a proper weight coefficient.

We note that Kiayias and Panagiotakos [69] present a unified description and security analysis of the GHOST protocol and Nakamoto consensus. This analysis relies on using a weighted norm, however, and their analysis only holds for linear weight functions. For blockchains, this means the weight of a subtree must increase linearly in relation to the number of blocks. This condition limits their analysis to boundary cases (e.g., Nakamoto consensus and GHOST); it cannot be applied to Medium's polynomial weight functions. We present a different approach, which adopts much of their notation and builds on their methodology and models. This should facilitate a comparison of the two protocols, including the spectrum between them.

The existence of protocols achieving a higher throughput than both Nakamoto consensus and GHOST is a well-known fact. Some of the most prominent examples are: BitcoinNG [53], Conflux [79], and Prism [16]. The reason for studying the spectrum between GHOST and Nakamoto consensus is that the previously mentioned protocols use either Nakamoto consensus or GHOST as a building block. Hence, given that Medium has either better security or better throughput than Nakamoto consensus or GHOST, these sophisticated protocols may inherit Medium's properties.

BitcoinNG [53] uses Nakamoto consensus to elect leaders. These leaders have the ability to generate many blocks. However, the security of the protocol depends completely on these leader-election blocks. Hence, a different rule for leader election at a higher security level, or with a higher ratio of leaders per unit of time, translates into an immediate upgrade of this protocol.

The main innovation behind Conflux [79] is its ability to include abandoned blocks in the ledger. Conflux uses the GHOST's rule to agree on a main chain. Consequently, Conflux uses a secondary set of references in order to topologically order the complete DAG. Conflux then purifies this DAG to eliminate all the possible double-spendings and builds the ledger. Once again, a better rule for the selection of the main chain improves the totality of the protocol.

With regard to Prism [16], finally, the situation is slightly more complex because its selection rule is more sophisticated. In this protocol, a block is not classified as valid or invalid depending on the value of its header. Instead, it is classified into several groups depending on the value of the hash function. One of these groups is invalid, another one allows the block to contribute with its transactions, but not to the chain selection, and another group contributes only to this chain selection. The security of this protocol relies exclusively on this last group; the chain selection inside this group follows a variation of Nakamoto consensus. Hence, Medium's chain-selection rule could again be exploited to upgrade Prism.

3.3 Model

3.3.1 General definitions

Similarly to the Bitcoin Backbone protocol [56], the execution of the protocol takes place in rounds. At the start of each round, parties receive the messages sent to them in the previous round, then the parties perform specific operations and finish the round by specifying the messages they want to broadcast.

A *block* is defined as a tuple of the form $b = [s, x, i, ctr]$ with $s \in \{0,1,\}^{\kappa}, x \in \{0,1\}^{*}$, $ctr \in \mathbb{N}$ and $i \in \{1, ..., n\}$ (where n is the total number of parties). Two cryptographic hash functions $G(\cdot)$ and $H(\cdot)$, which are modeled as random oracle functionalities [20] are used to define the validity of a block.

A block, mined by party P_i is defined as *valid* if it satisfies the condition

$$(H(ctr, G(s, x, i)) < D) \wedge (ctr \leq q),$$

where D is the difficulty parameter and q is the maximum number of hash queries in a round.

A *chain* C is a sequence of valid blocks, starting from the root block ($genesis(C)$) and extending to a final, head block ($head(C)$). For a chain to be valid each block in the chain must be valid and fulfill the condition that if a block $b = [s, x, i, ctr]$ extends block $b' = [s', x', i', ctr']$ in the chain, then $s = H(ctr', G(s', x', i'))$.

We say a new block has been *mined* if a valid block can be found that extends a chain in this valid manner. Since the difficulty parameter is D, the success probability of a single hashing query is $p = \frac{D}{2^\kappa}$, where κ is the length of the hash.

Miners that attempt to mine on the blockchain are referred to as *parties*.

There are a total of n mining parties, of these the adversary controls a maximum of f, the parties controlled by the adversary are called *corrupted*. Parties running the protocol are called *honest* and only communicate at the end of a round. When an honest party mines a block this block is referred to as an *honest block*. When an corrupted party, controlled by the adversary, mines a block it is referred to as a *corrupted block*.

A round is called *successful* if an honest party mines a block in that round, and *uniquely successful* if only one honest party mines in that round.

The length of the chain C is denoted by $\ell(C)$. When looking at a chain C, we say C *extends* another chain C' if C' is a prefix of C, we can then write $C' \preceq C$. The *depth* of a block b in the blockchain is the length of the path from that block to the genesis block. The tree of blocks mined by these parties is called the *block tree*, each party has a *local view* of the block tree which is comprised of all the valid mined blocks that it knows about. A *fork* in the tree occurs when two parties mine on the same block, extending the same chain. Which chain and therefore which block in the tree is mined on is decided by each party according to the protocol, this chain is referred to as the *main chain* and is chosen by the main chain selection algorithm. How the protocol handles forks must be defined in such an algorithm.

3.3.2 Communication and mining

We base our security analysis on the model used in the Bitcoin Backbone paper [56]. This model is an enhancement of the *Synchronous-rounds model* (Section 2.2.1) and *Byzantine model* (Section 2.1) in which parties have limited access to a *random oracle* functionality.

The *random oracle* is a functionality that can be queried in two different ways. If queried with input x as calculation, the random oracle returns a random string of a given length κ if it was not queried with x before. If was previously queried with input x it returns the same output as before. However, it can also be queried as verification with inputs (x, y), the random oracle outputs 1 if it was queried, for calculation, before with input x and the corresponding output was y. Otherwise it outputs 0. (The separate verification queries let this differ from the standard random-oracle model, but this is necessary in our context [56].)

Any party has access to q queries of the random oracle for calculation, the adversary has q queries per corrupted party. The number of queries for verification is unbounded for honest parties, however the adversary has no access to verification queries. This has been called the *q-bounded flat model* [56].

3.4 The Medium protocol

The *Medium* protocol proceeds roughly like the Nakamoto consensus and GHOST protocols [69] by arranging the received blocks into a tree, as also formalized by the Bitcoin Backbone protocol [56]. Nakamoto consensus then selects the longest branch in the tree as its main chain, and GHOST constructs its main chain by greedily selecting the block with the heaviest subtree by number of blocks. In Medium, the main chain is determined by always following the heaviest *weighted* subtree, using the Medium weight function introduced here.

Definition 8 (Weight). The *weight of a block* b in a tree T is given by

$$\omega_c(b, T) = c^{d(b)},$$

where $d(b)$ denotes the depth of b in T when the Medium protocol is instantiated with weight coefficient c.

Definition 9 (Tree Weight). The *weight of a tree* T is the sum of the weights of all blocks of T,

$$\omega_c(T) = \sum_{b' \in T} \omega_c(b', T).$$

Notice that the contribution of each block to the tree weight depends on the position of the block in the tree. We define $T(b)$ to be the subtree rooted at a block b and refer to the weight of $T(b)$ as the *tree weight* b.

3.4.1 Detailed description

In more detail, each party starts a round with a local view of the block tree and its current main chain C. To determine the new main chain, the protocol recursively iterates over the block tree, starting from the genesis block. At each block, the protocol extends the main chain with the child that has the heaviest tree weight, that is, by choosing the (polynomially weighted) heaviest subtree. Ties are broken by choosing the root of the subtree that results in the longest main chain, or if this would be the same, then by selecting the block that has been received earlier. Extending the main chain through proof-of-work (POW) occurs similarly to the Bitcoin Backbone protocol.

The miner starts the round and checks the input string $RECEIVE_i$ for new blocks. The miner then runs *update*() to extend its local tree and validate any received blocks. Then it runs the *Medium*() algorithm, as illustrated in Algorithm 1, to determine its main chain. If *update*() has added a new block to the local tree, the miner broadcasts this new block again at the end of the round.

After this is completed the miner can start running the POW algorithm to try to mine a new block that can extend the main chain and fulfill the needed properties for validity. If the party mines such a block it uses the diffusion functionality to send a message with the block information to all parties at the end of the round, we call this *broadcasting* the block. By broadcasting the blocks the party has accepted during a round again

Algorithm 1 Main chain selection algorithm

```
1: function Medium(T, ω_c)                    // a tree T and a weight function ω_c
2:     b ← root(T)
3:     if desc(b) = ∅ then
4:         return b
5:     else                                   // break ties by larger depth of trees
6:         b ← argmax{ω_c(T(b')) : b' ∈ desc(b)}
7:         return b‖Medium(T(b), ω_c)         // concatenate blocks

8: function ω_c(T)                            // weight function ω_c with coefficient c
9:     b ← root(T)
10:    sum ← 0
11:    for b' ∈ desc(b) do
12:        sum ← sum + ω_c(T(b'))
13: return c · sum + 1
```

at the end of the round the protocol ensures other honest parties also receive the same blocks and can update their own trees accordingly. This ensures that if an adversary broadcasts in round r to an honest party by the end of round $r+1$ all other parties also receive the block. A formal description is included in Appendix 3.9.

3.4.2 Choice of the weight coefficient

To make it harder for the adversary to perpetrate the balance attack. we may choose weight coefficients $c > 1$ of a particular shape. Given a tree T, we can express its tree weight $\omega_c(T)$ as a polynomial in c of degree ℓ,

$$\omega_c(T) = a_0 c^0 + a_1 c^1 + ... + a_\ell c^\ell,$$

where ℓ is the depth of the tree and the coefficient a_i expresses how many blocks there are at level i in the tree. We observe that $a_i \geq 1$ for $i \in \{0, ..., \ell\}$; furthermore the total number of blocks in the tree is $N = \sum_i a_i$. We can use these polynomials to compare the weight of two different trees, T_1 and T_2. Two trees have equal weight whenever

$$\begin{aligned} 0 &= \omega_c(T_1) - \omega_c(T_2) \\ &= (a_{0,1} - a_{0,2}) + ... + (a_{max\{\ell_1,\ell_2\},1} - a_{max\{\ell_1,\ell_2\},2}) c^{max\{\ell_1,\ell_2\}} \end{aligned}$$

Clearly, the weight of the two trees is the same if c is a root of the polynomial resulting from their difference. If we want two trees of given depth $\leq \ell$ to have the same weight if and only if they have the same structure, we need to consider a weight coefficient c that it is not a root of any polynomial of degree ℓ or less.

Consider the polynomial $f_{n,p}(X) = X^n - p$ with p a prime number and $n \geq 1$, by Eisenstein's criteria [51], this polynomial is irreducible on $\mathbb{Z}$. We define the set

$$\mathcal{S}_\ell = \{c : f_{n,p}(c) = 0 | c \in \mathbb{R}, n \geq \ell, p \text{ prime}\}, \tag{3.2}$$

any constant taken from this set is a root of an irreducible polynomial of degree at least ℓ. Hence, to make sure that two trees of depth $\leq \ell$ have the same weight if and only if they have the same structure, it is enough to consider any element from $\mathcal{S}_\ell$.

3.4.3 Relation with Nakamoto consensus and GHOST

Above we explained how to select the weight coefficient to guarantee that trees of some bounded depth have the same weight if and only if they have the same structure. However, there are different choices of c that are interesting to study.

If we select $c = 1$, our protocol reduces to the GHOST protocol. Additionally, the polynomial associated to the tree structure reduces to the number of blocks. In other words, we lose a huge amount of information regarding tree structure.

In the other extreme, if we consider increasing values of c, the weight of a block in the tree is the same as the weight of c blocks in the previous level. This difference increases with c, thus, when c is large, we need a large number of blocks in the previous level to match the weight of a single block. This shows, intuitively, that the Medium protocol behaves like Nakamoto consensus for $c \to \infty$ because the longest path in a subtree dominates its weight.

An execution that illustrates differences between Nakamoto consensus, GHOST, and Medium is shown in Figure 3.1.

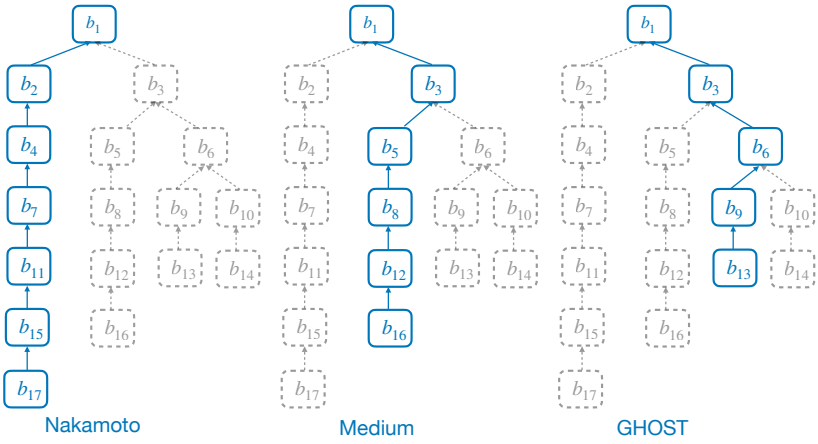


Figure 3.1. In this example, where every blocks represents an equal amount of hashing work, different chains are selected by Nakamoto consensus (left), GHOST (right), and Medium (center) with $c = 2$. Full (blue) block denote the main chain and dashed (grey) blocks denote off-chain blocks. Nakamoto consensus simply selects the longest chain, but much less hashing power may have gone into this than into the other subtree. GHOST, however, selects the blocks in the larger subtree. One drawback of GHOST is the big loss of information about the structure of the ignored subtrees. Medium selects a chain that represents more hashing power, than the chain chosen by Nakamoto consensus; at the same time, more structural information about the tree is taken into account by Medium than by GHOST.

3.5 Security analysis

The aim of this security analysis is to show that Medium is a robust transaction ledger, in other words, Medium satisfies liveness and persistence. To do this, we shall show that if a block is in the main chain and a sufficient number of blocks have been mined on this main chain after that block, so that these subsequent blocks weigh a predefined amount, then that block can be added to the ledger. This means, the block remains in the main chain of any honest party except with negligible

probability. We also show that in sufficiently many consecutive rounds there is always one honest block that enters the main chain and becomes stable.

We show this by establishing that the weight of the block tree increases in a specific manner during the execution of the protocol. This is done with the help of a *typical execution*. This denotes an execution in which for any set of enough consecutive rounds, the random variables do not diverge form the expected value in a significant quantity. An execution is not typical with negligible probability. We also analyze how the tree produced by running the Medium protocol behaves, which permits us to specify the corresponding increase in weight. We determine upper and lower bounds for this weight increase, which hold except with negligible probability. We use these bounds to establish our version of the common prefix property. If we remove blocks according to a specific weight condition from the main chains of two honest parties, the resulting chains are a prefix of each other. Furthermore, if we remove blocks according to this weight condition for one honest party at round r, this chain is a prefix of the main chain of any honest party in all later rounds. By determining the minimal number of rounds needed to let the block tree grow by a specific amount, we can also show how the implied main chain becomes stable. With this knowledge, we will finally show that a minimal number of honest blocks are produced in every consecutive subset of these rounds, that they are in the main chain, and that they remain stable.

3.5.1 Typical execution

We shall now introduce the formal notion of a *typical execution* [56], the idea is that if we have enough consecutive rounds, we can show that they fulfill certain properties with a high probability. Furthermore, we note that if we have have a set of consecutive rounds of a certain size, we can show that every subset of consecutive rounds within it, if large enough, also fulfills these properties. To define these properties we introduce the following notation, aligned with the Bitcoin Backbone paper [56].

We define X_{ijk} to be a Boolean random variable that denotes whether in round i the j-th query of the k-th honest party is successful. Furthermore, let Z_{ijk} be a Boolean random variable for the same case but for

Overview of Parameters and Variables	
q	Number of POW calls in a round for each party
p	Probability of POW call to be successful and block mined
κ	Length of hash, determines difficulty parameter $D = p2^{\kappa}$
n	Number of mining parties (we assume a flat setting)
f	Maximum number of parties controlled by the adversary
δ	Honest Majority Parameter, $\delta \in (0,1)$ with $f \leq (1-\delta)(n-t)$
β	Hashing power of the adversary per round, $\beta = tpq$
α	Hashing power of the honest parties per round, $\alpha = (n-t)pq$
h	Total hashing power per round, $h = \alpha + \beta$
γ	Probability that a round is successful $\gamma = 1-(1-p)^{(n-t)q}$
γ_u	Probability that a round is uniquely successful, $\gamma_u > (1-\frac{\gamma}{3}f)$
ε	Typical execution parameter, $\varepsilon \in (0,1)$
λ	Consecutive rounds needed for a typical execution
c	Weight coefficient $c > 1$ and $c \in \mathbb{R}$
K	Weight parameter for the common weighted prefix property, $K \in \mathbb{R}$

the k-th corrupted party mining. We also let Y_i denote whether or not *exactly one* honest party mines in round i, and let $\tilde{X}_i$ represent whether or not any *honest* party mines in round i. A round with $Y_i = 1$ is called *uniquely successful*. Given these, we define $X_i = \sum_{k=1}^{n-t} \sum_{j=1}^{q} X_{ijk}$ and $Z_i = \sum_{k=1}^{t} \sum_{j=1}^{q} Z_{ijk}$. For a set S of (consecutive) rounds, we define $X(S) = \sum_{r \in S} X_r$ and similarly for $Z(S)$, $\tilde{X}(S)$ and $Y(S)$. In summary, we obtain the following:

$X(S)$	Total number of blocks mined by an honest party in consecutive rounds S.
$\tilde{X}(S)$	Total number of times an honest party mines in a round, for consecutive rounds S.
$Z(S)$	Total number of blocks an adversary mines in consecutive rounds S.
$Y(S)$	Number of rounds in S that are uniquely successful.

We make the same *honest majority assumption* as in the Bitcoin Backbone [56], that there exists $\delta \in (0,1)$ such that $f \leq (1-\delta)(n-f)$. Let also

$$\alpha = \mathrm{E}[X_i] = pq(n-f)$$
$$\beta = \mathrm{E}[Z_i] = fpq$$
$$\gamma = \mathrm{E}[\tilde{X}_i]$$

from which it follows that $\mathrm{E}[Y] = \gamma_u = q(n-f)p(1-p)^{q(n-f)-1} > (1-\frac{\gamma}{3}h)$. We assume that $3\gamma + 3\varepsilon < \delta \leq 1$, where γ is the probability that a round is successful and $\varepsilon \in (0,1)$.

We also use Garay *et al.*'s notions of insertions, predictions, and copies [56]. In particular, an *insertion* occurs when, given a tree T with two consecutive blocks b and b' a block b^* created after b' so that b, b^*, and b' form three consecutive blocks of a valid chain inside the tree. A *copy* occurs if the same block exists in two different positions in the tree. A *prediction* occurs when a block extends one which was computed at a later round.

Definition 10. An *(ε, λ)-typical execution* for $\varepsilon \in (0,1)$ and $\lambda \geq 2/\gamma$, over a set S of at least λ consecutive rounds satisfies:

1. $(1-\varepsilon)\mathrm{E}[X(S)] < X(S) < (1+\varepsilon)\mathrm{E}[X(S)]$
2. $(1-\varepsilon)\mathrm{E}[\tilde{X}(S)] < \tilde{X}(S) < (1+\varepsilon)\mathrm{E}[\tilde{X}(S)]$
3. $(1-\varepsilon)\mathrm{E}[Y(S)] < Y(S)$
4. $Z(S) < Y(S)$ and $Z(S) < (1+\varepsilon)\mathrm{E}[Z(S)]$
5. There are no insertions, predictions, or copies.
6. $\sum_{j=1}^{q} X_{ijk} \leq 1$ for every honest party P_k.

We note that the points (2)–(5) correspond to the conditions for a typical execution as defined by Garay *et al.* [56].

Theorem 1. *An execution is (ε, λ)-typical with probability* $1 - e^{-\Omega(q\varepsilon^2\gamma\lambda+\kappa+q)}$.

Proof. The proof is analogous to the proof in the Bitcoin Backbone paper [56]. It follows directly from applying a Chernoff bound to $X(S)$, $Z(S)$ and $\tilde{X}(S)$. We note X_{ijk}, $\tilde{X}_i$ and Z_{ijk} are all independent Bernoulli trials. In all trials the probability that one of these is outside the given range is at most $2e^{-\mu\varepsilon^2/3}$, where μ is the respective expected value. Garay *et al.* [56] show that the expected values of these variables can all be rewritten to have an upper bound that is a factor of γ. Thus, an execution fulfills the first four criteria with probability $1 - e^{-\Omega(q\varepsilon^2\gamma\lambda)}$. They further showed that insertions, deletions and copies occur with probability bounded by $e^{-\Omega(\kappa)}$, as insertions and copies happen if a

block extends two distinct blocks, which means a collision has occurred and a prediction occurs at an equally small likelihood.

The final condition is directly influenced by the choice of q, and as p is already small, a Chernoff bound can be used to show that this occurs with probability bounded by $e^{-\Omega(q)}$.

Using the Union bound, we combine the previous three bounds to finish the proof. □

From now on, unless explicitly noted otherwise, all statements we make assume the conditions of a typical execution hold. In other words, we can find parameters ε, γ, λ, q and κ so that the properties hold with probability $1 - e^{-\Omega(q\varepsilon^2\gamma\lambda+\kappa+q)}$.

3.5.2 Properties of Medium

For analyzing the protocol, we define some of its main properties in the model of Garay *et al.* [56].

Definition 11 (Normalized tree weight). For a block b in tree T, we define the *normalized tree weight of* b, or $\bar{\omega}_c(T(b))$, to be the weight of the subtree on b (or the tree weight of b) divided by the weight of b, or

$$\bar{\omega}_c(T(b)) = \frac{\omega_c(T(b))}{c^{d(b)}}.$$

Definition 12 (k-dominant prefix). We define the *k-dominant* prefix of the chain C, or $C^{\lceil k}$, as the chain C without any blocks b for which $\omega_c(T(b)) < k$, with the parameter $k \in \mathbb{R}$. If there is no block b in chain C with $\tau_c(T(b)) \geq k$, $C^{\lceil k}$ is defined to be the genesis block.

We note that blocks are always removed from the head of the chain when computing the k-dominant prefix of a chain. We can now come to the properties.

Definition 13 (Normalized tree weight growth). For parameters $\tau \in \mathbb{R}, s \in \mathbb{N}$, for any honest block b mined in round r, and for a set of consecutive rounds S with size $|S| = s$ starting just after round r it

holds that when b is in the main chain of every honest party P_i during S, then the normalized weight of b increases by at least weight τ in the local view of every honest party P_i.

Definition 14 (Chain growth). There exist parameters $g > 0$ and $r_0 \in \mathbb{N}$ such that in round $r \geq r_0$, every honest party adopts a chain of length at least $g \cdot r$.

Definition 15 (Common weighted prefix). There exists a parameter $K \in \mathbb{R}$ so that for any pair of honest parties P_1 and P_2 that adopt main chains C_1, C_2 at rounds $r_1 \leq r_2$ in their respective local views, it holds $C_1^{\lceil K} \preceq C_2$.

Definition 16 (Fresh block). At round r there exists a parameter $u \in \mathbb{N}$ so that for any subset u consecutive rounds, there is at least one block mined by an honest party which is in the main chain of all honest parties in every round $r' \geq r$.

In the remainder of this section, we establish the chain growth, weight growth, common weighted prefix, and fresh block properties. From these, it is possible to show that a robust public transaction ledger exists on top of our protocol, which satisfies liveness and persistence; we do this in the next section.

3.5.3 Foundation lemmas and chain growth

We use *block trees* as defined by Kiayias and Panagiotakos [69]. $\mathcal{T}_r^P$ is the tree formed from the blocks that honest party P_i has received up to round r. $\mathcal{T}_r$ is the tree containing *all* blocks broadcast by any party up until round r. $\hat{\mathcal{T}}_r$ is the tree that contains $\mathcal{T}_r$ and also includes all blocks mined by honest parties at round r. This means that for any honest party P_j, we have

$$\mathcal{T}_r^P \subseteq \mathcal{T}_r \subseteq \hat{\mathcal{T}}_r \subseteq \mathcal{T}_{r+1}^P.$$

This follows intuitively from the fact that each honest party has a subtree of all broadcast blocks up to round r in their local view at the start of round r, thus $\mathcal{T}_r^P \subseteq \mathcal{T}_r$. This subtree always contains all honest blocks broadcast in the previous round. As honest parties broadcast all newly mined blocks and blocks they received before round r at the end of round r, $\mathcal{T}_r \subseteq \mathcal{T}_{r+1}^P$ must hold.

It is important to note the adversary can choose to only broadcast its blocks to certain honest parties, so two honest parties P_1 and P_2 may have received different blocks in round $r-1$, which means $\mathcal{T}_r^{P_1} \neq \mathcal{T}_r^{P_2}$. Thus the main chains of two honest parties may also differ in length. $\mathcal{T}_r$ is the tree containing *all* blocks broadcast by any party up until round r, the length of the main chain of this tree is unique, as there can only be multiple main chains in $\mathcal{T}_r$ if each has the same length and weight.

We define $\ell_{mc}(\mathcal{T}_r)$ to be the length of the main chain in $\mathcal{T}_r$. The length of the main chain in $\hat{\mathcal{T}}_r$ is also unique (as honest parties extend the main chain by at most one block in a typical execution). As in $\mathcal{T}_r^P$, the length of the main chain in $\mathcal{T}_{r+1}^P$ is not necessarily unique.

The next remark introduces a different perspective that simplifies the upcoming proofs.

Remark 1. Given two chains C_1, C_2 in the local view of some honest party P_i, such that one of them is the main chain, w.l.o.g. C_1. The fact that C_1 is the main chain means that at some point in the chain C_2 there is a block b_2 that has a sibling $b_1 \in C_1$ that has a heavier subtree. This follows directly from the fact that all the chains start with the genesis block and in every interaction the algorithm selects the block with the heaviest subtree.

We shall start our analysis by discussing chain length growth behavior during a typical execution.

Lemma 2. *If an honest party mines in round r and the adversary does not broadcast in round $r-1$ it holds that*

$$\ell_{mc}(\hat{\mathcal{T}}_r) = \ell_{mc}(\mathcal{T}_r) + 1.$$

Additionally, if this is an uniquely successful round all parties have the same local view and have the same main chain in $\hat{\mathcal{T}}_r$.

Proof. This is clear from the protocol, honest parties always mine on the main chain, which is chosen by recursively selecting the block with the heaviest subtree and heaviest subtree resulting in the longest main chain if there are ties. Unless an adversary broadcasts in round $r-1$ all honest parties mine on the same main chain unless there was a block with more than one descendant that had a subtree of the same weight,

resulting in two different main chains of the same length. Thus, if any honest parties are successful in round r, they extend the chain they are mining on by length 1 (only by length 1, due to point 5 of a typical execution (Definition 10)). As there can only be multiple main chains in the local views of honest parties if they all have the same length any chain that is mined on in round r by an honest party has the same length. Furthermore, the block that was mined in that round adds to the weight of the subtrees of all the previous blocks in the chain, thus a main chain in $\hat{\mathcal{T}}$ is a chain that was mined on, which now has length $\ell_{mc}(\mathcal{T}_r) + 1$.

Furthermore, it is clear that if only one party mines, only one main chain is extended and thus there cannot be another main chain in the local view of an honest party as we have assumed the adversary has not broadcast in the round before. □

With this we can prove the following lemma.

Lemma 3. *Assume that an honest block b_0, mined in round r_0, stays in the main chain of all the honest parties for a set of consecutive rounds S starting at round $r_0 + 1$, then the length increase of the main chain of a given party P_i, at the beginning of the first round just after S, is lower and upper bounded ($l(S)$ is the increase in length of the main chain during the set of rounds S) by:*

$$Y(S) - Z(S) \leq l(S) \leq \tilde{X}(S) + Z(S).$$

In other words, the length increase is lower bounded by the number of uniquely successful rounds minus the number of adversarial blocks released in S, and, upper bounded by the number of successful rounds plus the number of adversarial blocks released in S.

Proof. First of all, notice that only the blocks releases in the subtree of b are relevant. Since all the honest parties agree that b is in the main chain during all the execution, this means that blocks releases by the adversary mined previously to b can safely be ignored. We analyze first the lower bound, $Y(S) - \hat{Z}(S) \leq l(S)$. The result follows by induction over the number of uniquely successful rounds $Y(S)$. First of all, notice that any adversarial block produced before round r_0 is completely irrelevant since the assumption is that b_0 remains in the main chain. In other words it is

enough to analyze the structure of the subtree of b_0 and the adversarial blocks produced after or in rounds r_0.

- Case $Y(S) = 0$, the bound is trivially satisfied.
- Case $Y(S) = 1$. Since the hypothesis is that block b_0 stays in the main chain of any honest party, the unique uniquely successful block mined is a descendant of b_0. This implies that the main chain, which before the set of rounds S finished in b_0, no longer finishes with b_0 (b_0 is no longer a leaf).
- Case $Y(S) = 2$. This follows from the fact that b_0 stays in the main chain during all the execution and the existence of a chain of length two.
- Assume that the statement holds up to $n - 1$. However, assume that the statement is not true for n. Precisely, denote by r_1 the round in which the last uniquely successful block of S was mined, and define a set of rounds

$$S' := \{r \in \mathbb{N} | r_0 < r < r_1\}.$$

We obtain a system of two equations,

$$\begin{cases} l(S) < Y(S) - \hat{Z}(S) \\ l(S') \geq Y(S') - \hat{Z}(S'). \end{cases}$$

By definition of S', we observe that $Y(S') = Y(S) - 1$ and $\hat{Z}(S') = \hat{Z}(S) - k$, where k is the number of adversarial blocks released after the last uniquely successful block in S. Then we get

$$\begin{cases} l(S) < Y(S) - \hat{Z}(S) \\ l(S') \geq Y(S) - 1 - \hat{Z}(S) + k. \end{cases}$$

Since the minimum increase in length is one, and negating the second inequality, this means

$$\begin{cases} l(S) + 1 \leq Y(S) - \hat{Z}(S) \\ -l(S') \leq -Y(S) + 1 + \hat{Z}(S) - k. \end{cases}$$

Adding both equations gives

$$l(S) + 1 - l(S') \leq 1 - k.$$

and

$$k \leq l(S) - l(S').$$

This is a contradiction since $k \geq 0$. Thus, $l(S) \geq l(S')$ and we see that

$$\begin{cases} l(S) < Y(S) - \hat{Z}(S) \\ l(S) \geq l(S') \geq Y(S) - 1 - \hat{Z}(S) + k. \end{cases}$$

Taking into consideration that the minimum increase in length is one and that k is non-negative, it holds

$$\begin{cases} l(S) < Y(S) - \hat{Z}(S) \\ l(S) > Y(S) - \hat{Z}(S) \end{cases}$$

We conclude that the statement holds for $Y(S) = n$. This completes the inductive step and proves the lower bound in the lemma.

The upper bound follows trivially from the fact that the best case for length growth is when the adversary collaborates with the honest parties. □

Lemma 4. *Assume a set of consecutive rounds S after an honest b_0 is mined in round r_0, with $|S| \geq \lambda$. Assume b_0 is part of the main chain during the set of rounds S, then the length increase of the main chain f any honest party P_i is lower bounded by*

$$(1-\varepsilon)\mathrm{E}[Y(S)] - (1+\varepsilon)\mathrm{E}[Z(S).]$$

Proof. Follows from Lemma 3 and the properties of an (ε, λ) -typical execution. □

Corollary 5 (Chain growth). *The chain growth property (Definition 14) holds with parameters $r_0 = \lambda$ and $g = (1-\varepsilon)\gamma_u - (1+\varepsilon)\beta$.*

Proof. We apply Lemma 3 together with the fact that the genesis block is always part of the main chain for every honest party and $S = \{r \in \mathbb{N} | r' \leq r\}$ satisfies that $|S| \geq \lambda$. Thus, the conditions of an (ε, λ)-typical execution hold. These conditions also imply that $((1-\varepsilon)\gamma_u - (1+\varepsilon)\beta) \cdot r = (1-\varepsilon)\mathrm{E}[Y(S)] - (1+\varepsilon)\mathrm{E}[Z(S)] > 0$, where S is the set of rounds until r. □

3.5.4 Weight growth property

In this section we commence the full analysis to prove that Medium satisfies the normalized tree weight growth property.

We introduce notation needed to formalize bounds on the weight increase of the blocks in the main chain.

Definition 17. Given a block b and a round r such that b is in the main chain of some honest party P_i we define

$$\ell_b := \min_{P_i \text{honest}} \{\ell_{mc}(\mathcal{T}_r^P)\} - d(b).$$

ℓ_B is the minimum distance from block b to the head of the main chain in the local view of party P_i. We define ℓ_L as the maximal distance from a block b in the main chain of any honest party to any head of a chain in its subtree.

Lemma 6. *For any honest block b mined in round r and any set of consecutive rounds S starting just after r, consisting of at least λ rounds, the normalized weight increase $\Delta\bar{\omega}_c(T(b))$ respects*

$$\Delta\bar{\omega}_c(T(b)) < \sum_{i=1}^{\lceil(1+\varepsilon)(\gamma+\beta)|S|\rceil} k(i,|S|)\ c^i.$$

Where

$$k(i,|S|) = \begin{cases} 1 & \text{if} \quad i < \lceil(1+\varepsilon)(\gamma+\beta)|S|\rceil - \lceil\frac{(1+\varepsilon)(\alpha-\gamma)|S|}{(1+\varepsilon)\gamma}\rceil \\ (1+\varepsilon)\alpha & \text{if} \quad i \geq \lceil(1+\varepsilon)(\gamma+\beta)|S|\rceil - \lceil\frac{(1+\varepsilon)(\alpha-\gamma)|S|}{(1+\varepsilon)\gamma}\rceil \end{cases}.$$

If b is in the main chain of every honest party during S, then the normalized weight increase is also lower bounded by

$$\sum_{i=1}^{\lfloor((1-\varepsilon)\gamma_u-(1+\varepsilon)\beta)|S|\rfloor} c^i < \Delta\bar{\omega}_c(T(b)).$$

Both bounds in the local view of any honest party P_i.

Proof. On the one hand, the maximum weight increase occurs when the adversary collaborates with the honest parties and the honest parties mine in the subtree of b, and both the adversary and the honest parties succeed as often as possible. This respects the conditions of an (ε, λ)-typical execution and we can apply Lemma 3.

From Lemma 3, there is an upper bound in the length increase of the main chain of any honest party that considers b as part of the main chain: $l(S) \leq \tilde{X}(S) + \hat{Z}(S) < (1+\varepsilon)(\gamma+\beta)|S|$, using the conditions of an (ε, λ)-typical execution. Furthermore, the weight is maximized when all the forked blocks occur as deep as possible in the tree. Again, by the properties of an (ε, λ)-typical execution, the number of blocks in the tree is bounded by $(1+\varepsilon)(\alpha+\beta)|S|$, and the number of honest blocks mined per round is upper bounded by $(1+\varepsilon)\alpha$. We have at most $(1+\varepsilon)(\alpha-\gamma)|S|$ forked blocks, and in every level of the tree up to $(1+\varepsilon)\gamma$ blocks. We conclude that the best case for weight increase occurs when the last $\lceil \frac{(1+\varepsilon)(\alpha-\gamma)|S|}{1+\varepsilon)\gamma} \rceil$ levels of the tree contain all the forked blocks. Defining $k(i, |S|)$ as in the statement,

$$k(i,|S|) = \begin{cases} 1 & \text{if} \quad i < \lceil (1+\varepsilon)(\gamma+\beta)|S| \rceil - \lceil \frac{(1+\varepsilon)(\alpha-\gamma)|S|}{1+\varepsilon)\gamma} \rceil \\ (1+\varepsilon)\alpha & \text{if} \quad i \geq \lceil (1+\varepsilon)(\gamma+\beta)|S| \rceil - \lceil \frac{(1+\varepsilon)(\alpha-\gamma)|S|}{1+\varepsilon)\gamma} \rceil \end{cases}.$$

Hence,

$$\Delta\bar{\omega}_c(T(b)) < \sum_{i=1}^{\lceil (1+\varepsilon)(\gamma+\beta)|S| \rceil} k(i,|S|)\ c^i.$$

On the other hand, the minimum weight increase occurs when the main chain of some party P_i whose local view includes b in the main chain, is as short as possible. From Lemma 3, we observe that $l(S) \geq Y(S) - Z(S)$, and using the conditions of an (ε, λ)-typical execution, it follows $l(S) > ((1-\varepsilon)\gamma_u - (1+\varepsilon)\beta)|S|$. Furthermore, the worst case for the weight increase is when the adversary achieves this with allowing any superfluous block to the subtree of b. Hence

$$\Delta\bar{\omega}_c(T(b)) > \sum_{i=1}^{\lfloor ((1-\varepsilon)\gamma_u - (1+\varepsilon)\beta)|S| \rfloor} c^i.$$

□

We can now apply these bounds to achieve the normalized tree weight growth property.

Theorem 7 (Normalized tree weight growth). *The normalized tree weight growth property holds with parameters $s = |S| \geq \lambda$ and*

$$\tau = \sum_{i=1}^{\lfloor((1-\varepsilon)\gamma_u - (1+\varepsilon)\beta)|S|\rfloor} c^i.$$

Proof. This follows directly from Lemma 6. □

3.5.5 Common weighted prefix property

Lemma 6 has further applications than the ones discussed in Section 3.5.4.

Remark 2. Any block b requires at least λ consecutive rounds to get a normalized subtree-weight of at least

$$\sum_{i=1}^{\lceil(1+\varepsilon)(\gamma+\beta)\lambda\rceil} k(i,\lambda)\ c^i.$$

This is a direct consequence of Lemma 6.

This inequality constitutes the baseline to our proof of the common weighted prefix property. Before we go into this, we introduce two complementary lemmas.

Lemma 8. *Assume there exists a fork in $\mathcal{T}_r^P$, where P_i is any honest party. Denote by C_1 and C_2 two unique chains produced by this fork, which have the same prefix prior to this fork; assume that the last block in this common prefix was mined in round $r' \leq r$. Denote by b_1 and b_2 the first block in each chain after the fork. Further assume $\omega_c(\mathcal{T}_r^P(b_1)) < \omega_c(\mathcal{T}_r^P(b_2))$, $\ell(C_1) = \ell(C_2) + s$ for $s > 0$. Then, the adversary had to release s blocks from round $r' - 1$ to $r - 1$.*

Proof. To prove this statement we assume that there are no blocks in the subtree $\mathcal{T}_r^P(b_2)$ that are at a greater depth than the length of C_2. Still assuming this assumption holds let now assume the statement of the lemma does not hold and find a contradiction.

Take any block b_i in C_1 at depth $\ell(C_2)+i$, for $i \in [1,s]$, this block was mined at $r' \leq r_i \leq r$. We shall show that for each of the blocks b_i the adversary had to release at least one block for there to exist a tree of such a shape at round r.

If b_i is corrupted we do not have to show anything, thus we assume b_i is honest. This means for at least one honest party P'

$$\omega_c(\mathcal{T}_{r_i}^{P'}(b_1)) \geq \omega_c(\mathcal{T}_{r_i}^{P'}(b_2)).$$

Now, since we know that in round r

$$\omega_c(\mathcal{T}_r^P(b_2)) > \omega_c(\mathcal{T}_r^P(b_1)) > \omega_c(\mathcal{T}_r^P(b_1)) \geq \omega_c(\mathcal{T}_{r_i}^{P'}(b_1)) + \sum_{j=i}^{s} c^{\ell(C_2)+j}$$

we know that between rounds $r_i - 1$ and r there must have been at least $s-i+1$ blocks mined on C_2. (This follows from $\sum_{j=0}^{s-i} c^{s-j} > s-i+1$).

We examine two cases.

1. The first is, if the adversary did not broadcast in round $r_i - 1$. This means all parties have the same local view of $\mathcal{T}_{r_i}$. Thus, an honest party can only mine on C_2 (to produce the missing $s-i+1$ blocks on C_2) at round r_i, if $\mathcal{T}_{r_i}(b_1)$ and $\mathcal{T}_{r_i}(b_2)$ have the same weight and result in main chains of the same length, but this contradicts b_i being the only block at this depth . Thus, the blocks must have been mined after round r_i, but after round r_i, b_1 is the sibling with the heaviest tree weight, thus no honest party would have mined on C_2. Thus, the adversary must have released $s-i+1$ blocks on the subtree of b_2 for this fork to occur, or, if $i \neq s$ it can switch the local view of another honest party before further blocks are released on C_1. If it does this after round r_i it needs to compensate the weight produced on C_1 in this round and has to release more than one block (as blocks in the subtree on b_2 have a strictly lower depth). Otherwise the adversary could have changed

the local view of another honest party before r_i, this is the second case.

2. If the adversary did broadcast in round $r_i - 1$ it is possible to create two different local views by only releasing blocks to certain honest parties and have honest parties mine blocks on C_2 in that round. As the adversary must expend at least a block for this it just remains for us to show it cannot 'compensate' multiple b_i in this manner. (We note that under the conditions of an (ε, λ)-typical execution honest parties do not mine on their own blocks during a round and can only extend the length of a chain by 1 block).

   If honest parties release enough blocks for b_2 to have a heavier tree weight than b_1 after round r_i, i.e. $\omega_c(\hat{\mathcal{T}}_{r_i}(b_1)) < \omega_c(\hat{\mathcal{T}}_{r_i}(b_2))$ we note honest parties do not mine on C_1 without the adversary releasing further blocks. If we would like the following b_{i+1} to be honest we come back to this case, if $i = s$ we are done. Alternatively, the honest parties could mine enough weight for b_1 to have equal or heavier tree weight than b_2 after round r_i, if there are less than $k < s - i$ corrupted blocks needed for b_2 to have a higher tree weight than b_1 in round r than the adversary has won. As we assumed blocks on the subtree of b_2 cannot weigh more than $c^{\ell(C_1)}$ this leads us to the condition that $k > \sum_{j=i+1}^{s} c^j$ (as further blocks must be released on C_1), and thus $k > s - i - 1$, which would be a contradiction.

It remains to show this still holds if there are blocks in the subtree $\mathcal{T}_r^P(b_2)$ that are at a greater depth than the length of C_2. We show this follows recursively from our statement. If there was a block in $\mathcal{T}_r^P(b_2)$ at a greater depth there would be a fork in this subtree with blocks b_3 and b_2^* broadcast in round $r^* \geq r'$ resulting in two chains, C_2 and another C_3 with $\ell(C_3) = \ell(C_2) + s^*$, $s^* > 0$ and $\mathcal{T}_r^P(b_2^*) > \mathcal{T}_r^P(b_3)$. We apply this until there are no blocks in the heavier subtree with a longer length than the length of the main chain and then apply our proof. Then, such a subtree is only possible if the adversary broadcast at least s^* blocks from round $r^* - 1$ to $r - 1$. Using this our proof still holds.

Thus, for every block b_i in C_1 at depth $\ell(C_1) + i$, for $i \in [1, s]$ there is a corresponding corrupted block and the adversary must broadcast at least s blocks to produce such a fork.

□

We are now able to discuss the behavior of chains when removing blocks of a specific tree weight, we shall show the weighted common prefix property must hold by proving the following lemma. As we must take multiple cases of different possible tree structures into account the proof is quite lengthy.

Lemma 9. *Suppose at round r of an (ε, λ)-typical execution, an honest party has a chain C_1 and a chain C_2 is adopted by an honest party, such that C_2 differs from C_1 in a block b_2 with $\omega(T(b_2)) \geq \omega(T(b_1))$. That is, the blocks before b_2 in C_2 are the same as in C_1 and C_1 has b_1 in the place of b_2. Then $C_1^{\lceil K} \preceq C_2$ and $C_2^{\lceil K} \preceq C_1$ for weight*

$$K = \sum_{i=1}^{\lceil (1+\varepsilon)(\gamma+\beta)\lambda \rceil} k(i, \lambda)\ c^i.$$

Proof. We assume by contradiction, either $C_1^{\lceil K} \not\preceq C_2$ or $C_2^{\lceil K} \not\preceq C_1$. Consider the last block of the common prefix of C_1 and C_2 that was computed by an honest party at round r^* and at depth ℓ (this block could be genesis). We define $S = \{i : r^* \leq i \leq r\}$ and note that $|S| \geq \lambda$, due to Lemma 6. We shall show that this implies $Z(S) \geq Y(S)$ which is a contradiction. (We note that $Z(S) \geq Y(S)$ is only dependent on the size of S, thus $Z(S') \geq Y(S)$ holds for S' if $|S'| = |S|$, here we define $S' = \{i : r^* - 1 \leq i \leq r - 1\}$.)

To do this we shall examine an injection between the uniquely successful rounds in S (their number given by $Y(S)$) and the blocks needed to "balance them" on the other chain, so that at round r two different honest parties can have two different local views of the main chain.

We look at a uniquely successful round r_i, where the honest party who mined b_i at this round mines on the main chain in their current local view. We look at 3 different cases for the main chain C at $\mathcal{T}_{r_i}^P$ that the honest party P_i mines on. We first assume the honest party mines on chain C_1 or C_2, without loss of generality we assume that the honest party is mining on chain C_1, by $\ell_i(C_1)$ we denote the length of the chain at that round in the local view of the honest party mining in that round.

- **Case 1:** $\ell_i(C_1) > \ell_i(C_2)$ to balance this block on C_2 the adversary must release more than one block on the other side, due to blocks at lower depths having more weight.
- **Case 2:** $\ell_i(C_1) = \ell_i(C_2)$ to balance this block on C_2 the adversary must release one block at that level or more (if there is already a weight difference, or it cannot mine a block at the depth)
- **Case 3:** $\ell_i(C_1) < \ell_i(C_2)$ from Lemma 8 we know that for $\mathcal{T}_{r_i}^P(b_1)$ to weigh more than $\mathcal{T}_{r_i}^P(b_2)$ (and have a common root produced in round r^*) but be shorter by a length of $s > 0$ the adversary must have already broadcast s parties, even if in the best case the

$$c^{s+1} > \sum_{i=1}^{s} c^i = \frac{(c^s - 1)c}{(c-1)}$$

which holds for $c < 2$ and the adversary can 'balance' s blocks with 1 block, he must still broadcast s blocks to produce this kind of subtree, and thus still needs at least as many corrupted blocks as uniquely successful rounds to create this fork.

In all these cases we see that $Z(S) \geq Y(S)$ must occur for the adversary to win, which contradicts the assumption of a typical execution.

We still need to review what happens when the the honest party P_i mines on a different chain than C_1 or C_2, we call this chain C_3, for this to happen C_3 must be the main chain in its local view at the uniquely successful round r_i and there are blocks $b_{3,1}$ and $b_{3,2}$ so that $\omega_c(\mathcal{T}_{r_i}^P(b_{3,1})) \geq \omega_c(\mathcal{T}_{r_i}^P(b_1'))$ and $\omega_c(\mathcal{T}_{r_i}^P(b_{3,2})) \geq \omega_c(\mathcal{T}_{r_i}^P(b_2'))$ where $b_{3,1}$ is the first block on C_3 after it forks from C_1 and $b_{3,2}$ the first after C_3 forks from C_2, the blocks b_1' and b_2' are the first blocks in C_1 and C_2 after the fork of their respective chains from C_3.

- **Case 1:** $\ell_i(C_3) \geq \ell_i(C_2)$ and $\ell_i(C_3) \geq \ell_i(C_1)$ to balance this block on C_2 and C_1 the adversary must release at least one block on *both* C_2 and C_1.
- **Case 2:** $\ell_i(C_3) \geq \ell_i(C_2)$ and $\ell_i(C_3) < \ell_i(C_1)$, or $\ell_i(C_3) \geq \ell_i(C_1)$ and $\ell_i(C_3) < \ell_i(C_2)$. For $\ell_i(C_3) \geq \ell_i(C_2)$ resp. $\ell_i(C_3) \geq \ell_i(C_1)$ at least one block has to be released on C_2 resp. C_1 to balance the block on C_3. How many blocks were needed to produce $\ell_i(C_3) < \ell_i(C_1)$ while $\omega_c(\mathcal{T}_{r_i}^P(b_{3,1})) > \omega_c(\mathcal{T}_{r_i}^P(b_1'))$ is slightly more

complicated. To apply Lemma 8 we must go back to the round where the root of C_3 and C_1 entered the block tree, which could be before r^*. Therefore, we observe the following, C_1 is at least $s_1 > 0$ longer than C_3, as we are in an (ε, λ)-typical execution insertions do not occur, therefore the rounds these blocks were mined in must have been after round r^*. We know the adversary must have broadcast at least s_1 blocks from rounds $r^* - 1$ to $r_i - 1$ for such a tree structure to exist, the same is true for the case where $\ell_i(C_3) < \ell_i(C_2)$.

- **Case 3:** $\ell_i(C_3) < \ell_i(C_2)$ and $\ell_i(C_3) < \ell_i(C_1)$ as already shown this means that for both sides the length difference $s > 0$ must be produced earliest at round $r^* - 1$ by at least an equal number of blocks.

In all these cases we deduce that $Z(S') \geq Y(S)$ this contradicts the assumption of an (ε, λ)-typical execution and we have proved the weighted common prefix lemma. □

From this, the common weighted prefix property follows directly.

Theorem 10 (Common weighted prefix). *Let*

$$K = \sum_{i=1}^{\lceil (1+\varepsilon)(\gamma+\beta)\lambda \rceil} k(i, \lambda)\ c^i \tag{3.3}$$

be the normalized tree weight. Then, for any pair of honest parties P_1 and P_2 adopting chains C_1 and C_2 at rounds $r_1 \leq r_2$ in their respective local views, respectively, it holds $C_1^{\lceil K} \preceq C_2$.

Proof. We assume the theorem is not true and find a contradiction. This means that there are rounds $r_1 \leq r_2$ where honest parties P_1 and P_2 adopt chains C_1 and C_2 as their main chains respectively but $C_1^{\lceil K} \not\preceq C_2$. From Lemma 9 we know that for all chains $\tilde{C}_i$ in the local view of an honest party in round r_1 it must hold that $C_1^{\lceil K} \preceq \tilde{C}_i$ and $\tilde{C}_i^{\lceil K} \preceq C_1$. It follows that $\tilde{C}_i \not\preceq C_2$, which means that C_2 is not an extension of a chain that was in the local view of an honest party at round r_1.

This means that there must exist a round $r \geq r_1$ where an honest party adopted a chain C' over a chain C s.t. $C_1^{\lceil K} \preceq C$ but $C_1^{\lceil K} \not\preceq C'$ (implied

by $C_1^{\lceil K} \not\preceq C_2$) , however in this round we can again apply Lemma 9 that $C^{\lceil K} \preceq C'$, furthermore, as C is an extension of a chain that was in the main chain of an honest party at round r_1 we know that since blocks in the chain could not decrease weight, thus $C_1^{\lceil K} \preceq C^{\lceil K}$ must hold, which implies $C_1^{\lceil K} \preceq C'$ and is a contradiction. Thus we have proved the common weighted prefix property. □

It still remains for us to show the fresh block property, which we shall prove with our own version of the Chain Quality Lemma.

3.5.6 Fresh block property

Finally we prove that honest blocks eventually enter the ledger.

Theorem 11 (Fresh block). *The fresh block property is satisfied with parameter*

$$u = \frac{\hat{R}^2 + 2\hat{R}}{(1-\varepsilon)2\gamma} + \lambda R \tag{3.4}$$

where $\hat{R} = \lfloor \frac{R}{2} \rfloor$ *and* R *is the maximal constant that fulfills the equation*

$$\sum_{i=1}^{R+1} c^i \leq \sum_{i=1}^{\lceil (1+\varepsilon)(\gamma+\beta)\lambda \rceil} k(i,\lambda)\ c^i.$$

Proof. We use a similar proof strategy as the one applied for the common weighted prefix property 10. We analyze the honest blocks produced in a successful round during these u consecutive rounds, and show that an honest block mined in a successful round enters the main chain and remains there in all subsequent rounds. This is proven thanks to the constrains of the number of corrupted blocks produced by the adversary in an (ε, λ)-typical execution.

However, the structure of the tree at the start of these u rounds plays a role in how many honest blocks can be 'balanced' by the adversary. Assume the best case for the adversary that there exists another chain that is R blocks longer than the current main chain, this chain differs from the main chain at a fork produced in round r (this could be prior

to the start of the u rounds), so that the weight of the subtree at b_{mc} (that results in the main chain) must be greater than the weight of the subtree on b_R (that results in the R-blocks-longer chain). By releasing a block at depth $R+1$, the adversary can compensate up to H honest blocks, where H is the largest constant that satisfies

$$\sum_{i=1}^{H} c^i \leq c^{R+1}.$$

We note H is at maximum R. Whenever such a chain exists it is a vulnerability. However, we can show the length of such a chain is bounded in an (ε, λ)-typical execution, if

$$\sum_{i=0}^{R} c^i > K = \sum_{i=1}^{\lceil (1+\varepsilon)(\gamma+\beta)\lambda \rceil} k(i,\lambda)\ c^i$$

the common weighted prefix property no longer holds, as if the adversary released a block at depth $R+1$ only to certain honest parties then they would adopt this longer chain as their main chain. However, the K-prefix of this chain is not an extension of the other main chain, which is a contradiction. Thus, the maximal possible length of such a chain is bounded by the maximal constant R that solves the equation $\sum_{i=0}^{R} c^i \leq K$. Furthermore in an (ε, λ)-typical execution at the start of these u consecutive rounds there can be no chain that is longer than R.

After compensating H honest blocks, mined in uniquely successful rounds by one corrupted block, in an (ε, λ)-typical execution the adversary has at most $H-2$ blocks left, which it can use to build a fork with a chain that is $H-2$ longer than the current main chain and compensates a certain number of honest blocks H' (at maximum $R-2$), mined in uniquely successful rounds. Using the additional corrupted blocks mined during these rounds the adversary can build a fork with a $H'-2$ longer chain, this can continue until the adversary has used all the 'additional' blocks. In the worst case it takes

$$\sum_{i=0}^{\hat{R}} (2i+1) = \hat{R}^2 + 2\hat{R}$$

uniquely successful rounds, with $\hat{R} = \lfloor \frac{R}{2} \rfloor$, for all the adversary's additional blocks to be used. After this point the adversary always has to

release strictly more than one block to compensate the weight produced in each uniquely successful round to prevent the block mined in that round from entering the main chain. From the properties of an (ε, λ)-typical execution, we know that in $u - R\lambda$ rounds, there are at least $\hat{R}^2 + 2\hat{R}$ uniquely successful rounds, as

$$(1 - \varepsilon)(u - R\lambda)\gamma = \hat{R}^2 + 2\hat{R}.$$

Moreover, there is at least one honest block entering the main chain.

Enough blocks must now be mined on this honest block for it to have a large enough tree weight for it to be in the K-prefix of the main chain. As the adversary has no 'additional' blocks in every successful round it must release at least one block to prevent that block from entering the main chain. We consider groups of λ blocks.

- In the first group one honest block (more specifically $A := \alpha\lambda(1 - \epsilon) - \beta\lambda(1 + \epsilon)$ blocks) enters the main chain.
- In the second group, if the adversary removes the extra block(s) from the previous group, $2A$ honest blocks enter.
- In the $(R + 1)$-th group, $(R + 1)A$ blocks enter the main chain, thus the first of these blocks has a normalized tree weight of at least $\sum_{i=0}^{(R+1)A-1} c^i$, which is greater than K and thus enters in the K-prefix and is stable.

□

3.6 Robust public transaction ledger

The content x of each block b has not played any role in the security analysis of the Medium protocol described in Section 3.5. The properties achieved by the block in the Medium protocol are independent of their content. However, transactions submitted by users are the main component of the ledger. In this section, we model Medium as atomic broadcast (Definition 2), recalled below.

Definition 2 (Atomic broadcast). A protocol solves *atomic broadcast* with validity predicate V if it satisfies the following conditions, except with negligible probability:

Validity: If a honest party *ab-broadcasts* a transaction tx, then it eventually *ab-delivers* tx.

Agreement: If a honest party *ab-delivers* a transaction tx, then all honest parties eventually *ab-deliver* tx.

Integrity: For any transaction tx, every honest party *ab-delivers* tx at most once, and only if it was submitted by some user.

Total order: If honest parties P_i and P_j both *ab-deliver* transactions tx and tx', then p *ab-delivers* tx before tx' if and only if P_j *ab-delivers* tx before tx'.

External validity: If an honest party *ab-delivers* a transaction tx, then $V(tx) = \text{TRUE}$.

In this work, we do not specify an external validity predicate V for the transactions. The protocol can be instantiated with several validity predicates. A party running the Medium protocol *ab-broadcasts*(tx) when it mines a block $b = [s, x, i, ctr]$ such that $tx \in x$. Furthermore, we assume that parties update their state based on the transactions submitted by the users. A party *ab-delivers*(tx) each valid non-delivered transaction contained in a block in the K-prefix of its main chain. The order in which the *ab-delivery* occurs is according to the natural order defined by blocks in the chain and transactions inside the blocks. The validity of a transaction is performed according to the external validity predicate V.

Theorem 12. *The Medium protocol implements atomic broadcast.*

Proof. We assume an (ε, λ)-typical execution. We structure the proof property by property:

Validity: Assume that an honest party P_i *ab-broadcasts*(tx). By definition of *ab-broadcast*(tx), P_i mined a block $b = [s, x, i, ctr]$ such that $tx \in x$. If b eventually becomes part of the K-prefix of the main chain of P_i, it *ab-delivers*(tx), and the validity property is satisfied. However, b may not become part of such K-prefix. However, The Fresh block property (Definition 16, Theorem 11) implies that at any round r, there exists an honest block mined after round r that becomes part of the K-prefix of every party, in particular P_i. Since honest parties reinsert transactions included in the blocks

that are not part of the main chain, transaction tx is eventually included in a block that becomes part of the K-prefix of the main chain of P_i. Thus, P_i eventually *ab-delivers*(tx).

Agreement: Assume that an honest party *ab-delivers*(tx) in round r. By definition, there exists a block $b = [s, x, i, ctr]$ such that $tx \in x$ in the K-prefix of the main chain C_i of P_i, let us denote this prefix by C. Consider an honest party P_j with main chain C_j. The common prefix property (Definition 15, Theorem 10) guarantees that $C = C_i^{\lceil K} \preceq C_j$ for C_j the main chain of party P_j in any round $r' \geq r$. The normalized tree weight growth property (Definition 13, Theorem 7) implies the weight of any block b' that remains in C increases at least $\tau \cdot s$ after s rounds. Apply the normalized tree weight growth property to block b, note that the common prefix property implies that b remains in the main chain of party P_j. The weight of block b in the view of P_j eventually surpasses K. Thus, P_j eventually *ab-delivers* transaction tx, and the agreement property is satisfied.

Integrity: Given a transaction tx in a block in the K-prefix of the main chain of party P_i, transactions tx is only *ab-delivered* if it has not previously been *ab-delivered* and it has been submitted by a user. We conclude that the integrity property is satisfied.

Total order: Given two honest parties P_i, P_j such that both *ab-deliver* transactions tx and tx'. Party P_i *ab-deliver* tx and tx' because they are contained in a block in the K-prefix of its main chain. Assume that P_i *ab-delivers*(tx) in round r_i and P_j does in round $r_j \geq r_i$, denote C_i and C_j to be the main chain of P_i and P_j in the respective rounds. The common prefix property (Definition 15, Theorem 10) implies that $C_i^{\lceil K} \preceq C_j$. Furthermore, since transactions are *ab-delivered* according to the natural order defined by the chain, P_i delivers tx' before tx if and only if tx' occurs before tx in $C_i^{\lceil K}$. Since $C_i^{\lceil K} \preceq C_j$ the same applies to P_j. We conclude that both P_i and P_j *ab-deliver* tx and tx' in the same order. Thus, the total order property is satisfied.

External validity: Honest parties only *ab-deliver* transactions that satisfy the validity predicate V. Thus, the external validity property is satisfied.

Since an execution (ε, λ)-typical with all but negligible probability (The-

orem 1), we conclude that Medium implements atomic broadcast. □

Note that since both Nakamoto consensus and GHOST can be considered special cases of the Medium protocol, Theorem 12 serves as a proof that Nakamoto consensus and GHOST implement atomic broadcast.

3.7 Throughput

The particular characteristics of Nakamoto consensus, GHOST, and Medium allow to compare the throughput of these protocols in a unified and simplified manner. Namely, all protocols select one main chain as the correct one and ignore every block that is not part of it.

Bagaria *et al.* [16] show that for Nakamoto consensus, throughput is bounded by a security constraint which ensures that the malicious chain cannot grow faster in expectation than the honest main chain,

$$\alpha(1 - \psi_f) > \beta. \tag{3.5}$$

The variable ψ_f stands for the probability that a block forks, i.e. the probability that a successful round is not uniquely successful. Without this constraint,, an adversary would be able to build a secret chain that eventually becomes longer than the main chain of any honest party. It is clear that the throughput of the Nakamoto consensus protocol is limited when this probability is small. A low forking probability is correlated with a low mining ratio (number of blocks mined per unit of time). Therefore, Nakamoto consensus' throughput is limited by this constraint.

In the case of Medium, since the weight of a block increases exponentially with its depth in the tree, one might suspect that the security constraint is the same as in Nakamoto consensus. However, this is not exactly the case.

On the one hand, if a hypothetical adversary had access, for unlimited time, to some set of corrupted parties, the above constraint (3.5) still applies. The reason for this is that despite the contribution of the forked blocks, the secret chain of the adversary becomes at some point long enough to compensate for this.

On the other hand, if we consider a more realistic scenario, in which the adversary is allowed to perform this attack for some set of consecutive rounds rounds S only, the result is slightly different.

Lemma 13. *The expected weight of a subtree with N blocks and depth ℓ, starting at depth ℓ_0 produced by honest parties running the Medium protocol is $\frac{N}{\ell}\sum_{i=1}^{\ell} c^{i+\ell_0}$.*

Proof. When honest parties run the protocol, the main chain is always the longest and they only split mining power when a fork occurs. In a given round, the probability of multiple honest parties mining is constant. If all parties mine on a chain at depth d, the probability that there are multiple blocks mined at depth d is given by this constant. When only honest parties mine, after a successful round parties always increase the depth they are mining at, thus at every depth the probability of there being multiple blocks is constant.

A subtree of depth ℓ has at least weight $\sum_{i=1}^{\ell} c^{i+\ell_0}$, the tree is rooted at depth ℓ_0. The rest of the blocks, in total $N-\ell$, can be at any depth in the subtree, therefore they follow a uniform distribution. This means that the expected weight is $\frac{N-\ell}{\ell}\sum_{i=1}^{\ell} c^{i+\ell_0}$ for these blocks, adding this to the weight of the previous blocks gives us the expected weight of the subtree. □

From now on, assume that the adversary builds a secret chain after some honest block b_0, and all the weights are normalized by $c^{depth(b_0)}$. Writing $s = |S|$, the expected value of honest (malicious) blocks in a set of consecutive rounds S is αs (βs, respectively). The relative weight of this secret chain (C_s) is

$$\sum_{i=1}^{\lceil \beta s \rceil} c^i = \frac{c(c^{\lceil \beta s \rceil}-1)}{c-1} \simeq \frac{c(c^{\beta s}-1)}{c-1}.$$

Regarding the honest subtree, its expected number of blocks is αs and its depth $\alpha s(1-\psi_f)$, since a block does not fork, and increases the depth of the subtree, with probability $1-\psi_f$). Using Lemma 13, the expected weight of this subtree can be written as

$$\frac{\lfloor \alpha s \rfloor}{\lfloor \alpha s(1-\psi_f) \rfloor} \sum_{i=1}^{\lfloor \alpha s(1-\psi_f) \rfloor} c^i \geq \frac{\alpha s}{\alpha s(1-\psi_f)} \sum_{i=1}^{\lfloor \alpha s(1-\psi_f) \rfloor} c^i$$
$$\geq \frac{1}{(1-\psi_f)} \sum_{i=1}^{\lfloor \alpha s(1-\psi_f) \rfloor} c^i$$

$$> \sum_{i=1}^{\lfloor \lfloor \alpha s(1-\psi_f) \rfloor \rfloor} c^i \geq \frac{c(c^{\lfloor \alpha s(1-\psi_f) \rfloor} - 1)}{c-1}$$
$$\simeq \frac{c(c^{\alpha s(1-\psi_f)} - 1)}{c-1} > \frac{c(c^{\beta s} - 1)}{c-1}.$$

The last inequality follows from (3.5). Hence, we conclude that even given some probability of a fork ψ_f, an adversary is more likely to succeed attacking Nakamoto consensus than Medium. This implies that, at the same level of security, Medium can tolerate higher mining ratio.

Furthermore, we run simulations of the throughput of Medium for different values of c and compare this with Nakamoto consensus and GHOST. The results are shown in Figure 3.2. GHOST achieves a higher ratio of honest blocks than both Medium and Nakamoto consensus; however, this has to be contrasted with GHOST's susceptibility to a balance attack, as discussed in the next section.

3.8 Analysis of a balance attack

We first describe the details of the attack that we consider. It is structured as follows, with details shown in Algorithm 2:

1. The adversary cuts the communication between two sets of parties $\mathcal{P}_1$ and $\mathcal{P}_2$ with approximately equal hashing power. This partitions the network in two.
2. The honest parties continue running the protocol for τ rounds, but only receive blocks produced within their own partition. The parties build independent subtrees in each partition.

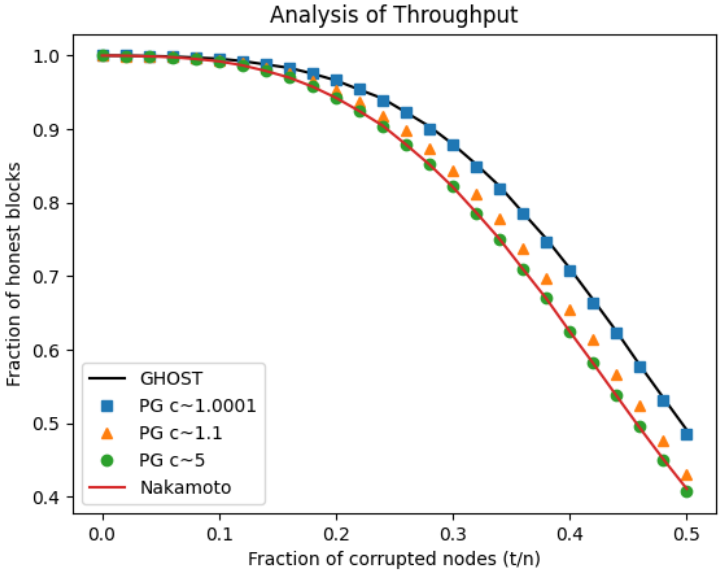


Figure 3.2. The fraction of honest blocks in the main chain depending on the number of corrupted parties, with fixed mining ratio such that $npq = 1$. The adversary's strategy is to build a heavier secret chain during eight rounds and to release this afterwards. The throughput of Medium (M) is shown for coefficients c of about 5, 1.1, and 1.0001 (the exact values are $\sqrt[10]{10001521}$, $\sqrt[100]{10001521}$, and $\sqrt[100000]{10001521}$, i.e., n-th roots of a prime according to Section 3.4.2). Throughput is higher with smaller values of c, and the black line corresponds to GHOST (almost overlapping Medium for $c \approx 5$, and the red line corresponds to Nakamoto consensus (almost overlapping with Medium for $c \approx 1.0001$). As expected, Medium lies between GHOST and Nakamoto consensus.

3. During these τ rounds, the adversary divides its hashing power between the partitions. Every block produced by the adversary is added to a bank of reserve blocks, $\mathcal{B}_1$ or $\mathcal{B}_2$, in the corresponding partition.
4. After τ rounds, the adversary enables communication among all parties again and tries to balance the two trees. This means that it releases blocks from the banks (or freshly mined blocks) with the goal of preventing that the parties agree on the same main chain across the former partitions. Notice that every block released like this may be broadcast selectively, so that it is only received by some honest parties initially. Even if the adversary may not be able to perfectly balance the trees with this strategy, it can release blocks to make one tree heavier than the other only in the local view of the parties in one partition.
5. Once the adversary runs out of blocks in the banks, the attack is over and the adversary cannot further balance the trees. Eventually, the honest parties converge on one subtree and on a single chain.

Simulations of the resistance of the Medium protocol against this attack are shown in Figure 3.3. The figure shows for how long the adversary can keep the fork alive and thus prevent the parties from agreeing after the partition has healed. Since deeper blocks weigh more, the adversarial strategy is to mine as deeply as possible in each partition. The duration of the fork in Medium can be almost an order of magnitude lower than in GHOST and comparable with Nakamoto consensus.

Theorem 14. *Under the assumptions of an (ε, λ)-typical execution*[2]*, the duration of the balance attack on Medium is bounded by $R\lambda$ rounds, where where R is the solution of*

$$b_\tau = \frac{(c^{R(1+\varepsilon)\lambda\beta} - 1)}{(c^{(1+\varepsilon)\lambda\beta} - 1)}.$$

b_τ is the sum of the number of blocks the adversary has in the banks after τ rounds, before he has released blocks to balance the fork.

Furthermore, if $\tau \geq \lambda$ then $b_\tau < (1+\varepsilon)pqt\tau$ and the bound can be

[2]The properties showed in Section 3.5 may not hold due to the partition of the network. However, the conditions of (ε, λ)-typical execution still hold.

Algorithm 2 DBLP:conf/dsn/natolig17 attack of τ rounds

Partition the network in two parts for τ rounds.
Denote the trees in each partition by T_1 and T_2.
Assume T_1 and T_2 are rooted at blocks b_1 and b_2.
The adversary splits his mining power between partitions
Adversary creates banks $\mathcal{B}_1$ and $\mathcal{B}_2$.
ℓ denotes the length of the main chain in subtree T_i
$n \leftarrow 0$ // number of rounds after the first τ
while TRUE **do**
  if $\exists i, j : [\omega(b_i) > \omega(b_j)] \vee [\omega(b_i) = \omega(b_j) \wedge \ell(T_i) > \ell(T_j)]$ **do**
    $\Delta \leftarrow \omega(b_i) - \omega(b_j)$
    if $\exists \mathcal{B}' \subseteq \mathcal{B}_j : [\omega(\mathcal{B}') \geq \Delta] \wedge [\ell(T_j \cup \mathcal{B}') > \ell(T_i \cup \mathcal{B}')]$ **do**
    Release subset $\mathcal{B}'$ of minimal weight to partition j
    else //adversary lost
      return
  $n \leftarrow n + 1$
  Honest parties and adversary mine on their respective local view

rewritten as

$$R < \frac{\log_c [(1+\varepsilon)pqt\tau(c^{(1+\varepsilon)\lambda\beta} - 1) + 1]}{(1+\varepsilon)\lambda\beta}.$$

Proof. The best case for the adversary is when after each round the two subtrees are equally balanced, in weight and length. It is clear that after each uniquely successful round the tree becomes unbalanced, and thus the adversary is forced to release at least one block to balance the subtrees. For any set of consecutive round of size at least λ, it holds that $Z(\lambda) < Y(\lambda)$, thus for every λ consecutive rounds there is at least one round where the adversary has to broadcast blocks from the bank.

We shall further show that the blocks in an adversary's bank always loose weight over time. We know that the length of the main chain of a subtree i increases by one after a uniquely successful round, to balance the other subtree j the adversary has to release blocks of equivalent or greater weight than the weight of the newly mined block in subtree i. If the adversary were to release blocks that balance this block but decrease the length of the main chain in subtree j it has to compensate a greater weight next time an honest party mines on i as the honest parties in j

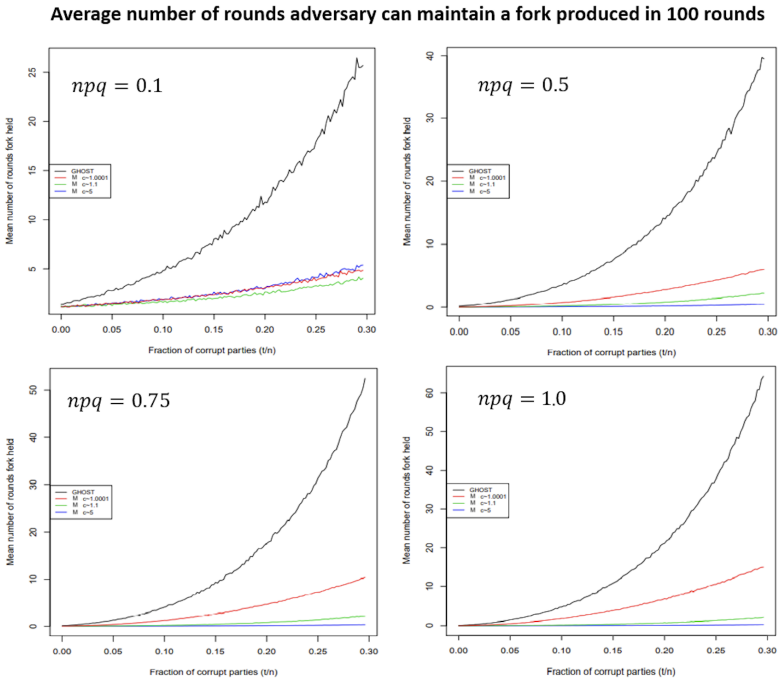


Figure 3.3. Simulations of how long a fork can be perpetuated by the adversary in the different protocols. A fork is created and maintained for 100 rounds, during which two partitions of the network are isolated from each other. The adversary compensates the weight of the heavier fork greedily, using the smallest number of blocks from its bank. This means that it more likely releases "heavier" blocks. The simulation shows four different values of the total mining ratio pqn. Again, we show Medium for c of about 5, 1.1, and 1.0001 (as in Figure 3.2). Notice that with c close to 1, the duration of the fork in Medium is almost an order of magnitude lower than in GHOST and comparable with Nakamoto consensus.

mine at a much lower depth. (Furthermore, if c is chosen so that blocks of a lower depth cannot fully balance blocks of a higher depth, as soon as the adversary releases blocks from the bank that are less deep in the chain it can longer fully balance the two chains and starts to be force to

release blocks in each subsequent round, regardless if an honest mines, meaning an even faster decrease of the bank and the attack failing even earlier.) Thus, we assume the main chains increase in length in any round that the adversary can balance them without using the bank, and the chain thus grow by $Z(\lambda)$ is λ rounds. Furthermore, we assume that the adversary can use it's bank in the first uniquely successful round before it has lost weight. After $R\lambda$ rounds the adversary have to release

$$\sum_{i=0}^{R-1} c^{(1+\varepsilon)\lambda\beta i} = \frac{(c^{R(1+\varepsilon)\lambda\beta} - 1)}{(c^{(1+\varepsilon)\lambda\beta} - 1)}$$

blocks from the bank.

If $\tau \geq \lambda$, the new bounds follow from the conditions of (ε, λ)-typical execution applied to the first τ rounds. □

Theorem 14 shows that the duration of the attack is bounded. However, the bound may not be tight in almost every execution.

3.9 Protocol details

We follow the approach of Kiayias and Panagiotakos [69] and use three external functions to describe our protocol, $V(\cdot), I(\cdot)$ and $R(\cdot)$. We call these functions the input validation predicate, the input contribution function and the chain reading function respectively. As in GHOST and Nakamoto consensus, $V(\cdot)$ controls that the content of a block fulfills specific criteria. We recall that a block is represented in the form $[s, x, i, ctr]$. $V(\cdot)$ only returns TRUE if all criteria hold for a block (the contents of a block are given in the x variable). The $I(\cdot)$ function in its simplest form tells a party what contents should be inserted into the next block to be mined. It receives as input a tuple, $(state, \mathcal{M}, C, round, RECEIVE_i)$, where *state* stands for state data, $\mathcal{M}$ for a set of transactions inputed by the users of the protocol and maintained by the party, C for the main chain, and messages received $RECEIVE_i$. Finally, the chain reading function $R(\cdot)$ reads the contents of the main chain C. The *BROADCAST*() function is the way a party P_i can send a message via the diffusion functionality to all other parties.

Algorithm 3 Medium protocol, as run by honest party i.

$T \leftarrow genesis$
$state \leftarrow \varepsilon$
for $round = 1, 2, 3...$ **do**
    $[T_{new}, b] \leftarrow update(T, \text{blocks found in } RECEIVE_i)$
    $C \leftarrow Medium(T_{new}, \omega_c)$ // ω_c is the global weight function
    $[state, x] \leftarrow I(state, \mathcal{M}, C, round, RECEIVE_i)$
    $C_{new} \leftarrow POW(x, i, C)$
    if $T \neq T_{new}$ **then**
        $BROADCAST(b)$
        $T \leftarrow T_{new}$
    if $C \neq C_{new}$ **then**
        $T \leftarrow update(T_{new}, head(C_{new}))$
        $BROADCAST(head(C_{new}))$
    output $R(C)$ // outputs the list of transactions in the chain

Algorithm 4 PoW function, with input (x, i, C), or block content x, party i and main chain C. This function parameterized by q, D, and cryptographic hash functions $G(\cdot)$ and $H(\cdot)$.

function $PoW(x, i, C)$
    if $C = \varepsilon$ **then**
        $s \leftarrow 0$
    else
        $[s', x', i', ctr'] \leftarrow head(C)$
        $s \leftarrow H(ctr', G(s', x', i'))$
    $ctr \leftarrow 1$
    $b \leftarrow \varepsilon$
    $h \leftarrow G(s, x, i)$
    while $(ctr \leq q)$ **do**
        if $(H(ctr, h) < D)$ **then**
            $b \leftarrow [s, x, i, ctr]$
            break
        $ctr \leftarrow ctr + 1$
    return $C||B$

Algorithm 5 Tree update function, with input a block tree T and a set of blocks b. Further parameters are q, D, $G(\cdot)$ and $H(\cdot)$.

```
52: function update(T, b)
53:     (b', b*) ← (∅, ∅)
54:     for [s', x', i', ctr'] in b do
55:         if V(x') then          // input x fulfills validation criteria
56:             b' ← b' ∪ [s', x', i', ctr']
57:     for [s, x, i, ctr] in T do
58:         for [s', x', i', ctr'] in b' do
59:             if s' = H(ctr, G(s, x, i))
60:             ∧ H(ctr', G(s', x', i')) < D ∧ ctr' ≤ q then
61:                 // [s', x', i', ctr'] is valid and extends the tree
62:                 insert [s', x', i', ctr'] into T
63:                  as descendent of [s, x, i, ctr]
64:                 b* ← b* ∪ [s', x', i', ctr']
65:     return [T, b*]
```

3.10 Conclusion

Medium is a family of protocols that implement a robust transaction ledger. Medium shares interesting properties with the well-known Nakamoto consensus and GHOST protocols. More precisely, Medium achieves better throughput than Nakamoto consensus, but not better than GHOST. However, with a proper choice of the weight coefficient c, Medium tolerates a balance attack some orders of magnitude better than GHOST. We conclude that Medium is a protocol that lies between GHOST and Nakamoto consensus and inherits the good properties from either side.

Future work may refine the security analysis of Medium, as the properties established here may not be tight. Alternatively, a Markov-chain based analysis [72] could be used. Another extension would be to consider dynamic sets of parties [35].

Chapter 4

An analysis of Avalanche consensus

Losto Caradhras, sedho, hodo,
nuitho i 'ruith!

Gandalf the Grey

4.1 Introduction

The Avalanche blockchain with its fast and scalable consensus protocol is one of the most prominent alternatives to first-generation networks like Bitcoin and Ethereum that consume huge amounts of energy. Its AVAX token is ranked 22nd according to market capitalization in October 2023 [41]. Avalanche offers a protocol with high throughput, low latency, excellent scalability, and a lightweight client. In contrast to many well-established distributed ledgers, Avalanche is not backed by proof of work. Instead, Avalanche bases its security on a deliberately metastable mechanism that operates by repeatedly sampling the network, guiding the honest parties to a common output. This allows Avalanche to reach

a peak throughput of up to 20'000 transactions per second with a latency of less than half a second [99].

This novel mechanism imposes stricter security constraints on Avalanche compared to other networks. Traditional Byzantine fault-tolerant consensus tolerates up to a third of the parties to be corrupted [91] and proof-of-work protocols make similar assumptions in terms of mining power [56, 54]. Avalanche, however, can tolerate only up to square root of the parties behaving maliciously. Furthermore, the transactions in the "exchange chain" of Avalanche (see below) are not totally ordered, in contrast to most other cryptocurrencies, which implement a form of atomic broadcast [27]. As the protocol is structured around a directed acyclic graph (DAG) instead of a chain, it permits some parallelism. Thus, the parties may output the same transactions in a different order, unless these transactions causally depend on each other. Only the latter must be ordered in the same way.

The consensus protocol of a blockchain is of crucial importance for its security and for the stability of the corresponding digital assets. Analyzing such protocols has become an important topic in current research. Although Bitcoin appeared first without formal arguments, its security has been widely understood and analyzed meanwhile. The importance of proving the properties of blockchain protocols has been recognized for a long time [30].

However, there are still protocols released today without the backing of formal security arguments. The Avalanche whitepaper [99] introduces a family of consensus protocols and offers rigorous security proofs for some of them. Yet the Avalanche protocol itself and the related Snowman protocol, which power the platform, are not analyzed. Besides, several key features of this protocol are either omitted or described only vaguely.

In this chapter, we explain the Avalanche consensus protocol in detail. We describe it abstractly through pseudocode and highlight features that may be overlooked in the whitepaper (Sections 4.3–4.4). Furthermore, we use our insights to formally establish safety properties of Avalanche. Per contra, we also identify a weakness that affects its liveness. In particular, Avalanche suffers from a vulnerability in how it accepts transactions that allows an adversary to delay targeted transactions by several orders of magnitude (Section 4.5), which may render the protocol useless in practice. The problem results from dependencies that exist among the

votes on different transactions issued by honest parties; the whitepaper does not address them. The attack may be mounted by a single malicious party with some insight into the network topology. Finally, we suggest a modification to the Avalanche protocol that would prevent our attacks from succeeding and reinstantiate liveness of the protocol (Section 4.6). This version, which we call *Glacier*, restricts the sampling choices in order to break the dependencies, but also eliminates the parallelism featured by Avalanche.

The vulnerability has been acknowledged by the Avalanche developers. The deployed version of the protocol differs however from the protocol in the whitepaper in a crucial way. It implements another measure that prevents the problem, as we explain as well (in Section 4.8).

4.2 Related work

Despite Avalanche's tremendous success, there is no independent research on its security. Recall that Avalanche introduces the "snow family" of consensus protocols based on sampling [99, 14]: Slush, Snowflake, and Snowball. Detailed proofs about liveness and safety for the snow-family of algorithms are given. The Avalanche protocol for asset exchange, however, lacks such a meticulous analysis. The dissertation of Yin [110] describes Avalanche as well, but does not analyze its security in more detail either.

Recall that Nakamoto introduced Bitcoin [88] without any formal analysis. This has been corrected by a long line of research, which established the conditions under which it is secure (e.g., by Garay, Kiayias, and Leonardos [56, 57] and by Eyal and Sirer [54]).

The consensus mechanisms that stand behind the best-known cryptocurrencies are meanwhile properly understood. Some of them, like the proof-of-stake protocols of Algorand [58] and the Ouroboros family that powers the Cardano blockchain [70, 46], did apply sound design principles by first introducing and analyzing the protocols and only later implementing them.

Many others, however, have still followed the heuristic approach: they released code first and were confronted with concerns about their secu-

rity later. This includes Ripple [12, 8] and NEO [108], in which several vulnerabilities have been found, or Solana, which halted multiple times in 2021–2022. Stellar comes with a formal model [81], but it has also been criticized [73].

Protocols based on DAGs have potentially higher throughput than those based on chains. Notable examples include PHANTOM and GHOSTDAG [103], the Tangle of IOTA [95], Conflux [80], and others [63]. However, they are also more complex to understand and susceptible to a wider range of attacks than those that use a chain. Relevant examples of this kind are the IOTA protocol [83], which has also failed repeatedly in practice [107] and PHANTOM [103], for which a vulnerability has been shown [79] in an early version of the protocol.

Acknowledgement. The material contained in this chapter corresponds to the work 'When Is Spring Coming? A Security Analysis of Avalanche Consensus' [10] published at Opodis22.

4.3 Model

4.3.1 Avalanche platform

We briefly review the architecture of the Avalanche platform [14]. It consists of three separate built-in blockchains, the *exchange* or *X-Chain*, the *platform* or *P-Chain*, and the *contract* or *C-Chain*. Additionally there are a number of subnets. In order to participate in the protocols and validate transactions, a party needs to stake at least 2'000 AVAX (about 24'000 USD in November 2023 [41]).

The *exchange chain* or *X-Chain* secures and stores *transactions* that trade digital assets, such as the native AVAX token. The *X-Chain* is also used for cross-chain operations between *P-Chain* and *C-Chain* . This chain implements a variant of the *Avalanche consensus protocol* that only partially orders the transactions and that is the focus of this work. All information given here refers to the original specification of Avalanche [99].

The *platform chain* or *P-Chain* secures *platform primitives*; it manages all other chains, designates parties to become validators or removes them again from the validator list, and creates or deletes wallets. The P-Chain implements the *Snowman* consensus protocol: this is a special case of Avalanche consensus that always provides total order, like traditional blockchains. It is not explained in the whitepaper and we do not describe it further here.

The *C-Chain* hosts *smart contracts* and runs transactions on an Ethereum Virtual Machine (EVM). It also implements the Snowman consensus protocol of Avalanche and totally orders all transactions and blocks.

4.3.2 Communication and adversary

We consider a Byzantine adversary (Section 2.1) in the exponential delay communication model (Section 2.2.3), which we briefly recall below.

Parties have access to two low-level primitives: *point to point links* to *send* and *receive* a messages. And a *gossip* primitive to *gossip* and *hear* messages. Messages are delivered according to an exponential distribution, that is, the amount of time between the sending and the receiving of a message follows an exponential distribution with unknown parameter to the parties. However, messages from corrupted parties are not affected by this delay and will be delivered as fast as the adversary decides.

4.3.3 Abstractions

Since Avalanche, as introduced in its whitepaper [99] does not batch transactions into blocks, in this chapter we make a distintion between *payload transaction* and *transaction*. The *payload transactions* of Avalanche are submitted by users and built according to the *unspent transaction output* (*UTXO*) model of Bitcoin [88]. A payload transaction *tx* contains a set of *inputs*, a set of *outputs*, and a number of digital signatures. Every input refers to a position in the output of a transaction executed earlier; this output is thereby *spent* (or *consumed*)

and distributed among the outputs of *tx*. The balance of a user is given by the set of unspent outputs of all transactions (UTXOs) executed by the user (i.e., assigned to public keys controlled by that user). A payload transaction is valid if it is properly authenticated and none of the inputs that it consumes has been consumed yet (according to the view of the party executing the validation).

Blockchain protocols are generally formalized as *atomic broadcast*, since every party running the protocol outputs the same ordered list of transactions. However, the transaction sequences output by two different parties running Avalanche may not be exactly the same because Avalanche allows more flexibility and does not require a total order. Avalanche only orders transactions that causally depend on each other. Thus, we abstract Avalanche as a *generic broadcast* (Definition 3) according to Pedone and Schiper [93], we recall the definition below.

Definition 3 (Generic broadcast). A protocol solves *generic broadcast* with validity predicate V and relation $\sim$ if it satisfies the following conditions, except with negligible probability:

Validity: If a honest party *g-broadcasts* a transaction *tx*, then it eventually *g-delivers*(*tx*).

Agreement: If a honest party *delivers* a transaction *tx*, then all honest parties eventually *deliver*(*tx*).

Integrity: For any transaction *tx*, every honest party *delivers*(*tx*) at most once, and only if it was submitted by some user.

Partial order: If honest parties P_i and Q_i both *deliver* transactions *tx* and *tx*′ such that $tx \sim tx'$, then P_i *delivers*(*tx*) before it *delivers*(*tx*′) if and only if P_j *delivers*(*tx*) before it *delivers*(*tx*′).

External validity: If a honest party *delivers* a transaction *tx*, then $V(tx) = \text{TRUE}$.

The instantiation of the *equivalence relationship* and the *validity predicate* are specified in the following definitions.

Definition 18 (Related). Two payloads *tx* and *tx*′ are said to be *related*, denoted by $tx \sim tx'$, if *tx* consumes an output of *tx*′ or vice versa.

Definition 19 (External validity). A payload *tx* satisfies the validity predicate of Avalanche if all the cryptographic requirements are fulfilled

and there is no other delivered payload with any input in common with tx.

For the remainder of this work, we fix the external validation predicate V to check the validity of payloads according to the logic of UTXO mentioned before.

In our context, broadcasting corresponds to submitting a payload transaction to the network, whereas delivering corresponds to accepting a payload and appending it to the ledger.

The Avalanche protocol augments payload transactions to *protocol transactions*. A protocol transaction additionally contains a set of *references* to previously executed protocol transactions, together with further attributes regarding the execution. A protocol transaction in the implementation contains a batch of payload transactions, but this feature of Avalanche is ignored here, since it affects only efficiency. Throughout this chapter, *transaction* refers to a protocol transaction, unless the opposite is indicated, and *payload* means simply a payload transaction. Protocol transactions are denoted by T and payload by tx.

A transaction references one or multiple previous transactions, unlike longest-chain protocols, in which each transaction has a unique parent [88]. An execution of the Avalanche protocol will therefore create a directed acyclic graph (DAG) that forms its ledger data structure.

Given a protocol transaction T, all transactions that it references are called the *parents* of T and denoted by $parents(T)$. The parents of T together with the parents of those, recursively, are called the *ancestors* of T, denoted by $ancestors(T)$. Analogously, the transactions that have T as parent are called the *children* of T and are denoted by $children(T)$. Finally, the children of T together with their recursive set of children are called the *descendants* of T, denoted by $descendants(T)$.

Note that two payload transactions tx_1 and tx_2 in Avalanche that consume the same input are not related, unless the condition of Definition 18 is fulfilled. However, two Avalanche payloads consuming the same output *conflict*. For each transaction T, Avalanche maintains a set $conflictSet[T]$ of transactions that conflict with T.

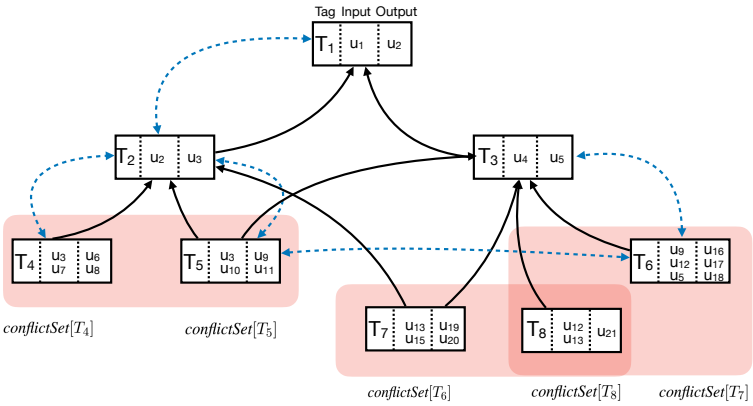


Figure 4.1. The UTXO model, conflicting transactions, and related transactions in Avalanche. The eight transactions are labeled $T_1, ..., T_8$. Each transaction is divided into three parts: the left part is a tag T_i to identify the transaction, the middle part is its set of inputs, and the right part is its set of outputs. The solid arrows indicate the references added by the protocol, showing the parents of each transaction. For instance, T_5 references T_2 and T_3 and has them as parents. The dashed double-arrows indicate related transactions. For example, T_5 and T_2 are related because u_3 is created by T_2 and consumed by T_5. The conflict sets are denoted by the shaded (red) rectangles. As illustrated, conflict sets can be symmetric, as for T_4 and T_5, where the conflict sets are identical ($conflictSet[T_4] = conflictSet[T_5]$) or asymmetric, as for T_6, T_7, and T_8 where $conflictSet[T_6] \cup conflictSet[T_7] = conflictSet[T_8]$.

4.4 A description of the Avalanche protocol

Avalanche's best-known quality is its efficiency. Permissionless consensus protocols, such as those of Bitcoin and Ethereum, are traditionally slow, suffer from low throughput and high latency, and consume large amounts of energy, due to their use of proof-of-work (PoW). Avalanche substitutes PoW with a random sampling mechanism that runs at network speed and that has every party adjust its preference to that of a (perceived) majority in the system. Avalanche also differs from more traditional blockchains by forming a DAG of transactions instead of a chain.

4.4.1 Overview

Avalanche is structured around its *polling* mechanism. In a nutshell, party P_i repeatedly selects a transaction T and sends a *query* about it to k randomly selected parties in the network. If a majority of them send a positive reply, the query is successful and the transaction contributes to the security of other transactions. Otherwise, the transaction is still processed but does not contribute to the security of any other transactions. Then the party selects a new transaction and repeats the procedure. A bounded number of such polls may execute concurrently. Throughout this work the terms "poll" and "query" are interchangeable.

In more detail, the protocol operates like this. Through the *gossip* functionality, every party is aware of the network membership $\mathcal{N}$. A party locally stores all those transactions processed by the network that it knows. The transactions form a DAG through their references as described in the previous section.

Whenever a user submits a payload transaction tx to the network, the user actually submits it through a party P_i. Then, P_i randomly selects a number of leaf nodes from a part of the DAG known as the *virtuous frontier*; these are the leaf nodes that are not part of any conflicting set. Party P_i then extends tx with references to the selected nodes and thereby creates a transaction T from the payload transaction tx. Next, P_i sends a QUERY message with T to k randomly, according to stake, chosen parties in the network and waits for their replies in the

form of VOTE messages. When a party receives a query for T and if T and its ancestors are *preferred*, then the party replies with a positive vote. The answer to this query depends exclusively on the status of T and its *ancestors* according to the local view of the party that replies. Moreover, the definition of *preferred* is non-trivial and will be explained further below. If the polling party receives more than $\alpha > \frac{k}{2}$ positive votes, the poll is defined to be successful.

Every party P_i running the Avalanche protocol sorts transactions of its DAG into conflict sets.

Definition 20 (Conflict set). The *conflict set* $conflictSet[T]$ of a given transaction T is the set of transactions that have an input in common with T (including T itself).

Note that even if two transaction T and T' consume one common transaction output and thus conflict, their conflict sets $conflictSet[T]$ and $conflictSet[T']$ can differ, since T may consume outputs of further transactions. (In Figure 4.1, for example, T_8 conflicts with T_6 and T_7, although T_7 conflicts with T_8 but not with T_6.)

Decisions on accepting transactions are made as follows. For each of its conflict sets, a party selects one transaction and designates it as *preferred*. This designation is parametrized by a *confidence value* $d[T]$ of T, which is updated after each transaction query. If the confidence value of some conflicting transaction T^* surpasses $d[T]$, then T^* becomes the preferred transaction in the conflict set.

It has been shown [59, 99] that regardless of the initial distribution of such confidence values and preferences of transactions, this mechanism converges. For the transactions of one conflict set considered in isolation, this implies that all honest parties eventually prefer the same transaction from their local conflict sets. (The actual protocol has to respect also dependencies among the transactions; we return to this later.)

To illustrate this phenomenon, assume that there exist only two transactions T and T' and that half of the parties prefer T, whereas the other half prefers T'. This is the worst-case scenario. Randomness in sampling breaks the tie. Without loss of generality, assume that parties with preferred transaction T are queried more often. Hence, more parties consider T as preferred as a consequence. Furthermore, the next

time when a party samples again, the probability of hitting a party that prefers T is higher than hitting one that prefers T'. This is the "snowball" effect that leads to ever more parties preferring T until every party prefers T.

This preferred transaction is the candidate for acceptance and incorporation into the ledger. The procedure is parametrized by a confidence counter for each conflict set, which reflects the probability that T is the preferred transaction in the local view of the party. The party increments the confidence counter whenever it receives a positive vote to a query on a descendant of T; the counter is reset to zero whenever such a query obtains a negative vote. When this counter overcomes a given threshold, T is accepted and its payload is added to the ledger. We now present a detailed description of the protocol and refer to the pseudocode in Algorithm 6–9.

4.4.2 Data structures

The information presented here has been taken from the whitepaper [99], the source code [15], or the official documentation [14].

Notation. We introduce the notation used in the remaining sections including the pseudocode. For a variable a and a set $\mathcal{S}$, the notation $a \xleftarrow{R} \mathcal{S}$ denotes sampling a uniformly at random from $\mathcal{S}$. We frequently use hashmap data structures: A hashmap associates keys in a set $\mathcal{K}$ with values in $\mathcal{V}$ and is denoted by $\mathit{HashMap}[\mathcal{K} \to \mathcal{V}]$. For a hashmap $\mathcal{F}$, the notation $\mathcal{F}[K]$ returns the entry stored under key $K \in \mathcal{K}$; referencing an unassigned key gives a special value $\perp$.

We make use of timers throughout the protocol description. Timers are created in a stopped state. When a timer has been started, it produces a *timeout* event once after a given duration has expired and then stops. A timer can be (re)started arbitrarily many times. Stopping a timer is idempotent.

Global parameters. We recall that we model Avalanche as run by an immutable set of parties $\mathcal{N}$ of size n. There are more three global

parameters: the number k of parties queried in every poll, the majority threshold $\alpha > \frac{k}{2}$ for each poll, and the maximum number *maxPoll* of concurrent polls.

Local variables. Queried transactions are stored in a set $\mathcal{Q}$, the subset $\mathcal{R} \subset \mathcal{Q}$ is defined to be the set of *repollable* transactions, a feature that is not explained in the original whitepaper [99]. The number of active polls is tracked in a variable *conPoll*. The parents of a transaction are selected from the *virtuous frontier*, $\mathcal{VF}$, defined as the set of all *non-conflicting* transactions that have no known descendant and whose ancestors are preferred in their respective conflict sets. A transaction is non-conflicting if there is no transaction in the local DAG spending any of its inputs. For completeness, we recall that conflicting transactions are sorted in *conflictSet[T]* formed by transactions that conflict with T, i.e., transactions which have some input in common with T.

Transactions bear several attributes related to queries and transaction preference. A *confidence value* $d[T]$ is defined to be the number of positive queries of T and its descendants. Given a conflict set $conflictSet[T]$, the variable $pref[conflictSet[T]]$, called *preferred transaction*, stores the transaction with the highest confidence value in $conflictSet[T]$. The variable $last[conflictSet[T]]$ denotes which transaction was the preferred one in $conflictSet[T]$ after the most recent update of the preferences. The preferred transaction is the candidate for acceptance in each conflict set, the acceptance is modeled by a counter $cnt[conflictSet[T]]$. Once accepted, a transaction remains the preferred one in its conflict set forever.

4.4.3 Detailed description

Each transaction does through three phases during the consensus protocol: query of transactions, reply to queries, and update of preferences. All of the previous phases call the same set of functions.

Functions. The function *updateDAG*(T) sorts the transactions in the corresponding conflict sets. The function *preferred*(T) (L 163) outputs

TRUE if T is the preferred transaction in its conflict set and FALSE otherwise. The function *stronglyPreferred(T)* (L 165) outputs TRUE if and only if T, and everyone of its ancestors is the preferred transaction in its respective conflict set.

The function *acceptable*(T) (L 167) determines whether T can be accepted and its payload added to the ledger or not. Transaction T is considered accepted when one of the two following conditions is fulfilled:

- T is the unique transaction in its conflict set, all the transactions referenced by $\mathcal{T}$ are considered accepted, and *cnt*[*conflictSet*[T]] is greater or equal than β_1.
- *cnt*[*conflictSet*[T]] is greater or equal than β_2.

Finally, the function *updateRepollable*() (L 171) updates the set of repollable transactions. A transaction T is repollable if T has already been accepted; or all its ancestors are preferred, a transaction in its conflict set has not already been accepted, and no parent has been rejected

Transaction query. A party in Avalanche progresses only by querying transactions. In each of these queries, party P_i selects a random transaction T (L 98), from the set of transactions that P_i has not previously queried by P_i. Then, it samples a random subset $\mathcal{S}[T] \subset \mathcal{N}$ of k parties from the set of parties running the Avalanche protocol and sends each a [QUERY, T] message. In the implementation of the protocol, P_i performs *numPoll* simultaneous queries. The repoll functionality (L 98–113) consists of performing several simultaneous transactions. When P_i does not know of any transaction that has not been queried, P_i queries a transaction that has not been accepted yet. The main idea behind this functionality is to utilize the network when this is not saturated. The repoll functionality (L 98–113) constitutes one of the most notable changes from Avalanche's whitepaper [99].

Query reply. Whenever P_i receives a query message with transaction T, party P_i replies with a message [VOTE, u, T, *stronglyPreferred(T)*] containing the output of the binary function *stronglyPreferred(T)* according to its local view (L 165).

Update of preferences. Party P_i collects the reply messages $[\text{VOTE}, v, T, \textit{stronglyPreferred}(T)]$, and counts the number of positive votes. On the one hand, if the number of positive votes overcomes the threshold α (L 118), the query is considered successful. In this case party P_i loops over T and all its ancestors T', increasing the confidence level $d[T']$ by one. If T' is the preferred transaction in its conflict set, then party P_i increases the counter for transaction $\textit{cnt}[\textit{conflictSet}[T']]$ by one. Subsequently, P_i checks whether T' has also previously been the preferred transaction in its conflict set. And when T' is not the preferred transaction according to the most recent query, party P_i will set the counter to one (L 118–132), in order to ensure that $\textit{cnt}[\textit{conflictSet}[T']]$ correctly reflects the number of consecutive successful queries of descendants of T'.

On the other hand, if P_i receives more than $k - \alpha$ negative votes, party P_i loops also over T and its ancestors, and sets their counters $\textit{cnt}[\textit{conflictSet}[T']]$ to zero as if to indicate that T' and the other transactions should not be accepted yet. (L 133–138).

Acceptance of transactions. Party P_i accepts transaction T when its counter $\textit{cnt}[\textit{conflictSet}[T]]$ reaches a certain threshold β_1 or β_2. If T is the only transaction in its conflicting set and all its parents have already been accepted, then P_i accepts T if $\textit{cnt}[\textit{conflictSet}[T]] \geq \beta_1$, otherwise P_i waits until the counter overcomes a higher value β_2.

No-op transactions. The local DAG is modified whenever a poll is finalized. In particular, only the queried transaction ans its ancestors are modified. Avalanche makes use of *no-op transactions* to modify all the transactions in the DAG. After finalizing a poll, party P_i queries the network with all the transactions in the virtuous frontier whose state has not been modified, in a sequential manner.

4.4.4 Life of a transaction

We follow an honest transaction T through the protocol. The user submits the payload transaction *tx* to some party P_i, then P_i adds references *refs* to the payload transaction, creating a transaction $T =$

(*tx*, *refs*). These references point to transactions in the virtuous frontier $\mathcal{VF}$. Transaction T is then *gossiped* through the network and added to the set of known transactions $\mathcal{T}$(L 87–93). Party P_i may also *hear* about new transactions through this gossip functionality. Whenever this is the case, P_i add the transaction to its set of known transactions $\mathcal{T}$ (L 94–97).

Party P_i eventually selects T to be processed. When this happens, P_i *samples* k random parties from the network and stores them in $\mathcal{S}[T]$. Party P_i *queries* parties in $\mathcal{S}[T]$ with T and starts a timer *timeout*$[T]$. T is added to $\mathcal{Q}$ (L 98–113).

Parties queried with T reply with the value of the function *stronglyPreferred*(T) (L 165). This function answers positively

Algorithm 6 Avalanche (party P_i), state

Global parameters and state

66: $\mathcal{N}$ // set of parties
67: $maxPoll \in \mathbb{N}$ // maximum number of concurrent polls, default 4
68: $k \in \mathbb{N}$ // number of parties queried in each poll, default value 20
69: $\alpha \in \{\lceil \frac{k+1}{2} \rceil, ..., k\}$ // majority threshold for queries, default 15
70: $\beta_1 \in \mathbb{N}$ // threshold for early acceptance, default value 15
71: $\beta_2 \in \mathbb{N}$ // threshold for acceptance, default value 150
72: $\mathcal{T} \leftarrow \emptyset$ // set of known transactions
73: $\mathcal{Q} \subset \mathcal{T} \leftarrow \emptyset$ // set of queried transactions
74: $\mathcal{R} \subset \mathcal{Q} \leftarrow \emptyset$ // set of repollable transactions
75: $\mathcal{D} \subset \mathcal{T} \leftarrow \emptyset$ // set of no-op transactions to be queried
76: $\mathcal{VF} \subset \mathcal{Q} \leftarrow \emptyset$ // set of transactions in the virtuous frontier
77: $conPoll \in \mathbb{N} \leftarrow 0$ // number of concurrent polls performed
78: $conflictSet : HashMap[\mathcal{T} \to 2^{\mathcal{T}}]$ // conflict set
79: $\mathcal{S} : HashMap[\mathcal{T} \to \mathcal{N}]$ // set of sampled parties to be queried
80: $votes : HashMap[\mathcal{T} \times \mathcal{N} \to \{\text{FALSE}, \text{TRUE}\}]$ // replies of queries
81: $d : HashMap[\mathcal{T} \to \mathbb{N}]$ // confidence value of a transaction
82: $pref : HashMap[2^{\mathcal{T}} \to \mathcal{T}]$ // preferred transaction among conflicts
83: $last : HashMap[2^{\mathcal{T}} \to \mathcal{T}]$ // preferred transaction in the last query
84: $cnt : HashMap[2^{\mathcal{T}} \to \mathbb{N}]$ // counter for acceptance
85: $accepted : HashMap[\mathcal{T} \to \{\text{FALSE}, \text{TRUE}\}]$ // transaction accepted
86: $timer : HashMap[\mathcal{T} \to \{timers\}]$ // timer for a query

Algorithm 7 Avalanche (party P_i), part 1

87: **upon** *broadcast*(tx) **do**
88: **if** $V(tx)$ **then**
89: $T \leftarrow (tx, \mathcal{VF})$ // up to a maximum number of parents
90: $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
91: *accepted*$[T] \leftarrow$ FALSE
92: *updateDAG*(T)
93: gossip message [BROADCAST, T]

94: **upon** hearing message [BROADCAST, T] **do**
95: **if** $T \notin \mathcal{T}$ **do**
96: $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
97: *accepted*$[T] \leftarrow$ FALSE

98: **upon** $conPoll < maxPoll$ **do**
99: $conPoll \leftarrow conPoll + 1$
100: **if** $\mathcal{D} \neq \emptyset$ **then** // prefer no-op transactions
101: $T \leftarrow$ least recent transaction in $\mathcal{D}$
102: **else if** $\mathcal{T} \setminus \mathcal{Q} \neq \emptyset$ **then** // any not yet queried transaction
103: $T \overset{R}{\leftarrow} \mathcal{T} \setminus \mathcal{Q}$
104: $d[T] \leftarrow 0$
105: **else** // take one queries transaction
106: *updateRepollable*()
107: $T \overset{R}{\leftarrow} \mathcal{R}$
108: $\mathcal{S}[T] \leftarrow sample(\mathcal{N} \setminus \{P_i\}, k)$ // according to stake
109: *send* message [QUERY, T] to all parties $v \in \mathcal{S}[T]$
110: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\bot, \mathcal{VF} \setminus \{T\})\}$ // create a no-op transaction
111: **start** *timer*$[T]$ // duration Δ_{query}
112: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{T\}$
113: *updateDAG*(T)

114: **upon** receiving message [QUERY, T] from party P_j **do**
115: *send* message [VOTE, u, T, *stronglyPreferred*(T)] to party P_j

116: **upon** receiving message [VOTE, v, T, w] such that $v \in \mathcal{S}[T]$ **do**
117: *votes*$[T, v] \leftarrow w$ // $w \in \{$FALSE, TRUE$\}$

Algorithm 8 Avalanche (party P_i), part 2

118:**upon** $\exists T \in \mathcal{T} : \left|\{v \in \mathcal{S}[T] \mid votes[T, v] = \text{TRUE}\}\right| \geq \alpha$ **do**
  //query of T is successful
119: **stop** $timer[T]$
120: $votes[T, *] \leftarrow \perp$ // remove all entries in $votes$ for T
121: $S[T] \leftarrow [\,]$ // reset S for T
122: $d[T] \leftarrow d[T] + 1$
123: **for** $T' \in ancestors(T)$ **do** // all ancestors of T
124: $d[T'] \leftarrow d[T'] + 1$
125: **if** $d[T'] > d[pref[conflictSet[T']]]$ **then**
126: $pref[conflictSet[T']] \leftarrow T'$
127: **if** $T' \neq last[conflictSet[T']]$ **then**
128: $last[conflictSet[T']] \leftarrow T'$
129: $cnt[conflictSet[T']] \leftarrow 1$
130: **else**
131: $cnt[conflictSet[T']] \leftarrow cnt[conflictSet[T']] + 1$
132: $conPoll \leftarrow conPoll - 1$

133:**upon** $\exists T \in \mathcal{T} : \left|\{v \in \mathcal{S}[T] \mid votes[T, v] = \text{FALSE}\}\right| > k - \alpha$ **do**
  // query of T failed
134: **stop** $timer[T]$
135: $votes[T, *] \leftarrow \perp$ // remove all entries in $votes$ for T
136: $S[T] \leftarrow [\,]$ // reset S for T
137: **for** $T' \in ancestors(T)$ **do** // all ancestors of T
138: $cnt[conflictSet[T']] \leftarrow 0$

139:**upon** $\exists T \in \mathcal{T}$ such that $acceptable(T) \wedge \neg accepted[T]$ **do**
  // T can be accepted
140: $(tx, parents) \leftarrow T$
141: **if** $V(tx)$ **then**
142: $accepted[T] \leftarrow \text{TRUE}$
143: *deliver* tx

145:**upon** timeout from $timer[T]$ **do** // not enough votes received
146: $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{T\}$
147: $votes[T, *] \leftarrow \perp$ // remove all entries in $votes$ for T
148: $S[T] \leftarrow [\,]$ // do not consider more votes from this query

Algorithm 9 Avalanche, auxiliary functions

149: **function** *updateDAG*(T)
150: $\mathcal{VF} \leftarrow$ set of non-conflicting leaves in the DAG
151: $conflictSet[T] \leftarrow \emptyset$
152: **for** $T' \in \mathcal{T}$ with a common input with T **do**
153: $conflictSet[T] \leftarrow conflictSet[T] \cup \{T'\}$
154: $conflictSet[T'] \leftarrow conflictSet[T'] \cup \{T\}$
155: **if** $conflictSet[T] = \{T\}$ **then** // T is non-conflicting
156: $pref[conflictSet[T]] \leftarrow T$
157: $last[conflictSet[T]] \leftarrow T$
158: $cnt[conflictSet[T]] \leftarrow 0$
159: $conflictSet[T] \leftarrow conflictSet[T] \cup \{T\}$

160: **function** *getParents*(T)
161: $(tx, parents) \leftarrow T$
162: **return** *parents* // set of parents stored in T

163: **function** *preferred*(T)
164: **return** $T \stackrel{?}{=} pref[conflictSet[T]]$

165: **function** *stronglyPreferred*(T)
166: **return** $\bigwedge_{T' \in ancestors(T)} preferred(T')$

167: **function** *acceptable*(T)
168: **return** $\big(|conflictSet[T]| = 1 \wedge cnt\big[conflictSet[T]\big] \geq \beta_1\big)$
$\bigwedge_{T' \in parents(T)} acceptable(T') \wedge \vee cnt\big[conflictSet[T]\big] \geq \beta_2$

169: **function** *isRejected*(T)
170: **return** $\exists T' \in \mathcal{T}$ such that *acceptable*(T')

171: **function** *updateRepollable*()
172: $\mathcal{R} \leftarrow \emptyset$
173: **for** $T \in \mathcal{T}$ **do**
174: **if** $acceptable(T) \vee \bigwedge_{T' \in parents(T)} stronglyPreferred(T')$
$\wedge \neg isRejected(T')$ **then**
175: $\mathcal{R} \leftarrow \mathcal{R} \cup \{T\}$

(TRUE) if T is *strongly preferred*, i.e., if T and all of its ancestors are the preferred transaction inside each respective conflict set. A negative answer (FALSE) is returned if either T or any of its ancestors fail to satisfy these conditions.

Party P_i then stores the answer from party P_j to the query in the variable $votes[T][v]$.

- If P_i receives more than α positive votes, P_i runs over all the ancestors of T. If the ancestor T' was the most recent (or "last") preferred transaction in its conflict set, its counter is increased by one. Otherwise, T' becomes the most recent preferred transaction and its counter is reset to one (L 118–132).
- If P_i receives at least $k - \alpha$ FALSE votes, P_i resets the counter for acceptance of all its ancestors $cnt[T'] \leftarrow 0$ (L 133–138).
- If timer $timeout[T]$ is triggered before the query is completed, the query is aborted instead. The votes are reset and every party is removed from the set $\mathcal{S}[T]$, so no later reply can be considered (L 145–148).

In parallel to the previous procedure, party P_i may perform up to *conPoll* concurrent queries of different transactions.

Once T has been queried, it awaits in the local view of party P_i to be accepted. Since by assumption T is honest, $conflictSet[T] = \{T\}$. Hence T is accepted when $cnt\big[conflictSet[T]\big]$ reaches β_1, if all the ancestors of T are already accepted, or β_2 otherwise (L 167–168). We recall that $cnt[conflictSet[T]]$ is incremented whenever a query involving a descendant of T is successful. However, when a non-descendant of T is queried, it may trigger a no-op transaction (L 100) that is a descendant of T.

If there is no new transaction waiting to be queried, i.e., $\mathcal{T} \setminus \mathcal{Q}$ is empty, the party proceeds with a *repollable* transaction (L 105–107). A *repollable* transaction is one that has not been previously accepted but it is a candidate to be accepted (L 171–174).

4.5 Security analysis

Avalanche deviates from the established PoW protocols and uses a different structure. Its security guarantees must be assessed differently. The bedrock of security for Avalanche is random sampling.

4.5.1 From Snowball to Avalanche

The Avalanche protocol family includes Slush, Snowflake, and Snowball [99] that implement single-decision Byzantine consensus. Every party *proposes* a value and every party must eventually *decide* the same value for an instance. The Avalanche protocol itself provides a "payment system" [99, Sec. V]; we model it here as generic broadcast.

The whitepaper [99] meticulously analyzes the three consensus protocols. It shows that as long as $f = O(\sqrt{n})$, the consensus protocols are live and safe [99] based on the analysis of random sampling [100]. On the other hand, an adversary controlling more than $\Theta(\sqrt{n})$ parties may have the ability to keep the network in a bivalent state.

However, the Avalanche protocol itself is introduced without a rigorous analysis. The most precise statement about it is that *"it is easy to see that, at worst, Avalanche will degenerate into separate instances of Snowball, and thus provide the same liveness guarantee for virtuous transactions"* [99, p. 9]. In fact, it is easy to see that this is *wrong* because every vote on a transaction in Avalanche is linked to the vote on its ancestors. The vote on a descendant T' of T depends on the state of T.

We address this situation here through the description in the previous section and by giving a formal description of Snowball in Appendix 4.7. We notice that one can isolate single executions of Snowball that occur inside Avalanche as follows. Consider an execution of Avalanche and a transaction T and define an *equivalent* execution of Snowball as the execution in which every party P_i proposes the value 1 if T is preferred in their local view, proposes 0 if another transaction is, and does not propose otherwise. Every party also selects the same parties in each round of snowball and for a query with T, for a query with a transaction that conflicts with T, or for any query with a descendant of these two.

Lemma 15. *If party P_i delivers an honest transaction in Avalanche, then P_i decided 1 in the equivalent execution of Snowball with threshold β_1. Furthermore, P_i delivers a conflicting transaction in Avalanche, then P_i decides 1 in Snowball with threshold β_2.*

Proof. By construction of the Avalanche and Snowball protocols in the whitepaper [99], the counter for acceptance of value 1 in Snowball is always greater or equal than the counter for acceptance in Avalanche. Since a successful query in Avalanche implies a successful query in Snowball, if an honest transaction in Avalanche is delivered, the counter in the equivalent Snowball instance is at least β_1. Analogously, if a conflicting transaction in Avalanche is delivered, then the counter in Snowball is at least β_2. Hence, a party in Snowball would decide 1 with the respective thresholds. □

Looking ahead, we will introduce a modification of Avalanche that ensures the complete equivalence between Snowball and Avalanche. We first assert some safety properties of the Avalanche protocol.

Theorem 16. *Avalanche satisfies integrity, partial order, and external validity of a generic broadcast for payload transactions under relation $\sim$ and UTXO-validity.*

Proof. The proof is structured by property:

Integrity: We show that every payload is delivered at most once. A payload *tx* may potentially be delivered multiple times in two ways: different protocol transactions that both carry *tx* may be accepted or *tx* is delivered multiple times as payload of the same protocol transaction.

First, we consider the possibility of accepting two different transactions T_1 and T_2 carrying *tx*. Assume that party P_i accepts transaction T_1 and party P_j accepts transaction T_2. By definition, T_1 and T_2 are *conflicting* because they spend the same inputs. Using Lemma 15, party P_i and P_j decide differently in the equivalent execution in Snowball, which contradicts agreement property of the Snowball consensus [99].

The second option is that one protocol transaction T that contains *tx* is accepted multiple times. However, this is not possible

either because tx is delivered only if $accepted[T]$ = FALSE; variable $accepted[T]$ is set to TRUE when transaction T is accepted (L 139–143).

Furthermore, a transaction that has not been submitted by a user in invalid, thus never delivered.

Partial order: Avalanches satisfies partial order because no payload is valid unless all payloads creating its inputs have been delivered (L 139–143). Transactions T and T' are related according to Definition 18 if and only if T has as input (i.e., spends) at least one output of T', or vice versa. This implies that related transactions are delivered in the same order for any party.

External validity: The external validity property follows from L 139, as a payload transaction can only be delivered if it is valid, i.e., its inputs have not been previously spent and the cryptographic requirements are satisfied.

□

Theorem 16 shows that Avalanche satisfies the safety properties of a generic broadcast in the presence of an adversary controlling $O(\sqrt{n})$ parties. A hypothetical adversary controlling more than $\Omega(\sqrt{n})$ parties could violate safety. It is not completely obvious how an adversary could achieve that. Such an adversary would broadcast two conflicting transactions T_1 and T_2. As we already discussed, and also explained in the whitepaper of Avalanche [99], such an adversary can keep the network in a bivalent state, so the adversary keeps the network divided into two parts: parties in part $\mathcal{P}_1$ consider T_1 preferred, and parties in part $\mathcal{P}_2$ prefer T_2. The adversary behaves as preferring T_1 when communicating with parties is $\mathcal{P}_1$ and as preferring T_2 when communicating with parties in $\mathcal{P}_2$. Eventually, a party $u \in \mathcal{P}_1$ will query only parties in $\mathcal{P}_1$ or the adversary for β_2 queries in a row. Thus, P_i will accept transaction T_1. Similarly, a party $v \in \mathcal{P}_2$ will eventually accept transaction T_2. Party P_i will *deliver* the payload contained in T_1 and P_j the payload contained in T_2, hence violating agreement.

An adversary controlling at most $O(\sqrt{n})$ can also violate agreement, but the required behavior is more sophisticated, as we explain next.

4.5.2 Delaying transaction acceptance

An adversary aims to prevent that a party P_i accepts an honest transaction T. A necessary precondition for this is $cnt[conflictSet[T]] \geq \beta_1$. Note that whenever a descendant of T is queried, $cnt[conflictSet[T]]$ is modified. If the query is successful (L 118), then $cnt[conflictSet[T]]$ is incremented by one. If the query is unsuccessful, $cnt[conflictSet[T]]$ is reset to zero. Remark, however, $cnt[conflictSet[T]]$ cannot be reset to one as a result of another transaction becoming the preferred in $conflictSet[T]$ (L 127) because T is honest, as there exist no transaction conflicting with T.

Our adversary thus proceeds by sending to P_i a series of cleverly generated transactions that reference T. We describe these steps that will delay the acceptance of T.

1. **Preparation phase.** The adversary submits conflicting transactions T_1 and T_2. For simplicity, we assume that she submits first T_1 and then T_2, so the preferred transaction in both conflict sets will be T_1. The adversary then waits until the target transaction T is submitted.
2. **Main phase.** The adversary repeatedly sends malicious transactions referencing the target T and T_2 to P_i. These transactions are valid but they reference a particular set of transactions.
3. **Searching phase.** Concurrently to the main phase, the adversary looks for transactions containing the same payload as T. If some are found, she references them as well from the newly generated transactions.

For simplicity, we assume that the adversary knows the acceptance counter of T at P_i, so she can send a malicious transaction whenever T is close to being accepted. In practice, she can guess this only with a certain probability, which will degrade the success rate of the attack. We also assume that the query of an honest transaction is always successful, which is the worst case for the adversary.

After P_i submits T, the adversary starts the main phase of the attack. If P_i queries an honest transaction $\hat{T}$, and if $\hat{T}$ references a descendant of T, then $cnt[conflictSet[T]]$ increases by one. If it does not, then $\hat{T}$ may cause P_i to submit a no-op transaction referencing a descendant of T.

Algorithm 10 Liveness attack: Delaying transaction T

Initialization

176: create two conflicting transactions T_1 and T_2
177: gossip two messages [BROADCAST, T_1] and [BROADCAST, T_2]
178: $\mathcal{A} \leftarrow \emptyset$

179:**upon** hearing message [BROADCAST, T] **do** // target transaction
180: $\mathcal{A} \leftarrow \{T\}$

181:**upon** $cnt[conflictSet[T]] = \lfloor \frac{\beta_1}{2} \rfloor$ in the local view of P_i **do**
182: create $\hat{T}$ such that $T_2 \in ancestors(\hat{T})$ and for all $T' \in \mathcal{A}$, also $T' \in ancestors(\hat{T})$
183: *send* message [BROADCAST, $\hat{T}$] to party P_i// pretend to gossip

184:**upon** hearing message [BROADCAST, $\tilde{T}$]**do** // $\tilde{T}$ resubmission
185: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\tilde{T}\}$

Hence, honest transactions always increase $cnt[conflictSet[T]]$ by one, this is the worst case for an adversary aiming to delay the acceptance of T.

If P_i queries a malicious transaction $\hat{T}$, honest parties reply with their value of $stronglyPreferred(\hat{T})$. Since T_2 is an ancestor of $\hat{T}$ and not the preferred transaction in its conflict set (as we have assumed that T_1 is preferred), all queried parties return FALSE. Thus, P_i sets acceptance counter of every ancestor of $\hat{T}$ to zero (L 133), in particular, $cnt[conflictSet[T]] \leftarrow 0$. However, since $\hat{T}$ does not reference the virtuous frontier, P_i submits a no-op transaction that references a descendant of T, thus increasing $cnt[conflictSet[T]]$ to one.

We show that when the number of transactions is low, in particular when $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ for every party, then Avalanche may lose liveness.

Theorem 17. *Avalanche does not satisfy validity nor agreement of generic broadcast with relation $\sim$ with one single malicious party if $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ for every party.*

Proof. We consider again the adversary described above that targets T

and P_i.

- **Validity.** Whenever $cnt[conflictSet[T]]$ in the local view of P_i reaches $\lfloor \frac{\beta_1}{2} \rfloor$, the adversary sends a malicious transaction to party P_i, who immediately queries it (since $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$). It follows that P_i sets $cnt[conflictSet[T]]$ to zero and increases it intermediately afterwards, due to a no-op transaction. This process repeats indefinitely over time and prevents P_i from delivering the payload in T.
- **Agreement.** Assume that an honest party broadcasts the payload contained in T. The adversary forces a violation of agreement by finding honest parties P_i and P_j such that $cnt[conflictSet[T]] = \beta_1 - 1$ at P_j and $cnt[conflictSet[T]] < \beta_1 - 1$ at P_i (such parties exist because in the absence of an adversary, as $cnt[conflictSet[T]]$ increases monotonically over time). The adversary then sends an honest transaction T_h that references T to P_j and a malicious transaction T_m, as described before, to P_i. On the one hand, party P_j queries T_h, increments $cnt[conflictSet[T]]$ to β_1, accepts transaction T, and delivers the payload. On the other hand, party P_i queries T_m and sets $cnt[conflictSet[T]]$ to one. After that, the adversary behaves as discussed before. Notice that P_j has delivered the payload within T but P_i will never do so.

□

An adversary may thus cause Avalanche to violate validity and agreement. For this attack, however, the number of transactions in the network must be low, in particular, $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$. In July 2022, the Avalanche network processed an average of 647238 transactions per day[1]. Assuming two seconds per query, four times the value observed in our local implementation, the recommended values of 30 transactions per batch, and four concurrent polls, the condition $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ is satisfied 88% of the time.

However, the adversary still needs to know the value of the counter for acceptance of the different parties.

[1] https://subnets.avax.network/stats/network

4.5.3 A more general attack

We may relax the assumption of knowing the acceptance counters and also send the malicious transaction to more parties through gossip. After selecting a target transaction, the adversary continuously gossips malicious transactions to the network instead of sending them only to one party. For analyzing the performance of this attack, our figure of merit will be the number of transactions to be queried by an honest party (not counting no-ops) for confirming the target transaction T. The larger this number becomes, the longer it will take the party until it may accept T. We assume that $\mathcal{T} \setminus \mathcal{Q} \neq \emptyset$ and that a fraction γ of those transactions are malicious at any point in time[2]. A non-obvious implication is that the repoll function never queries the same transaction twice.

Lemma 18. *Avalanche requires every party to query at least β_1 transactions before accepting transaction T in the absence of an adversary.*

Proof. The absence of an adversary carries several simplifications. Firstly, there are no conflicting transactions, thus every transaction is the preferred one in its respecting conflict set and every query is successful. Secondly, due to the no-op transactions, the counter for acceptance of every transaction in the DAG is incremented by one after each query. Finally, a transaction T is accepted when its counter for acceptance reaches β_1, since the counter of the parent of any transaction reaches β_1 strictly before T (L 167). □

Lemma 19. *The average number of queried transactions before accepting transaction T in the presence of the adversary, as described in the text, is at least*

$$\beta_1 + \frac{1 + (2 + \beta_1\gamma)(1-\gamma)^{\beta_1} - (1-\gamma)^{2\beta_1}(1 + \beta_1\gamma)}{\gamma(1-\gamma)^{\beta_1}(1 - (1-\gamma)^{\beta_1})}.$$

Proof. We recall that in the worst-case scenario for the adversary, the query of an honest transaction increments the counter for acceptance of

[2] Avalanche may impose a transaction fee for processing transactions. However, since the malicious transactions cannot be delivered, this mechanism does not prevent the adversary from submitting a large number of transactions.

the target transaction T by one, while the query of a malicious transaction, effectively, resets the counter for acceptance to one, as a result of a no-op transaction.

Let a random variable W denote the number of transactions queried by P_i until T is accepted, and let $X \in \{0, 1\}$ model the outcome of the following experiment. Party P_i samples transactions until it picks a malicious transaction or until it has sampled $\beta_1 - 1$ honest transactions. In the first case, X takes the value zero, and otherwise, X takes the value one. By definition, X is a Bernoulli variable with parameter $p = (1-\gamma)^{(\beta_1-1)}$. Thus, the number of attempts until X returns one is a random variable Y with geometric distribution, $Y \sim \mathcal{G}(p)$, with the same parameter p. We let W_a be the random variable denoting the number of queried transactions per attempt of this experiment. The expected number of failed attempts is $\mathrm{E}[Y] = \frac{1}{(1-\gamma)^{\beta_1}}$. Furthermore, the probability that an attempt fails after sampling exactly k transactions, for $k \leq \beta_1$, is

$$\mathsf{P}[W_a = k | X = 0] = \frac{\gamma(1-\gamma)^{k-1}}{1-(1-\gamma)^{\beta_1}}.$$

Thus, the expected number of transactions per failed attempt can be expressed as

$$\mathrm{E}[W | X = 0] = \frac{1-(1-\gamma)^{\beta}_1(1+\beta_1\gamma)}{\gamma(1-(1-\gamma)^{\beta_1})}. \tag{4.1}$$

The expected number of transaction queried during a successful attempt is at least β_1 by Lemma 18. Finally, the total expected number of queried transactions can be written as the expected number of transaction per failed attempt multiplied by the expected number of failed attempts plus the expected number of transactions in the successful attempt,

$$\mathrm{E}[W] = \mathrm{E}[W_a | X = 0] \cdot (\mathrm{E}[Y] - 1) + \mathrm{E}[W_a | X = 1] \cdot 1. \tag{4.2}$$

From equations (4.1) and (4.2) and basic algebra, we obtain

$$\mathrm{E}[W] = \beta_1 + \frac{1 + (2+\beta_1\gamma)(1-\gamma)^{\beta_1} - (1-\gamma)^{2\beta_1}(1+\beta_1\gamma)}{\gamma(1-\gamma)^{\beta_1}(1-(1-\gamma)^{\beta_1})}.$$

□

This expression is complex to analyze. Hence, a graphical representation of this bound is given in Figure 4.2. It shows the expected smallest

number of transactions to be queried by an honest party (not counting no-ops) until it can confirm the target transaction T. The larger this gets, the more the protocol loses liveness. It is relevant that this bound grows proportional to $\frac{1}{(1-\gamma)^{\beta_1}}$, i.e., exponential in acceptance threshold β_1 since $(1-\gamma) < 1$.

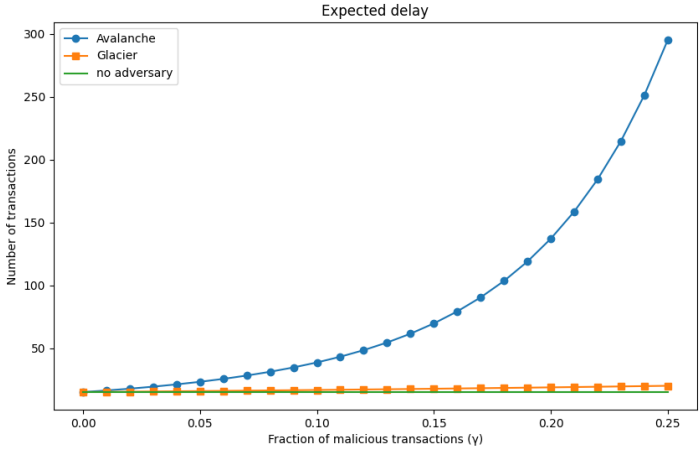


Figure 4.2. Expected delay in number of transactions needed to confirm a given transaction with acceptance threshold $\beta_1 = 15$, the recommended value [14], and assuming that the queries of honest transactions are successful. The (green) horizontal line shows β_1, the expected delay without attacker. The (blue) dotted line represents the expected confirmation delay in Avalanche depending on the fraction of malicious transactions. The (orange) squared line denotes the delay in Glacier (Section 4.6).

The Avalanche team has acknowledged our findings and the vulnerability in the abstract protocol. The protocol deployed in the actual network, however, differs from our formalization in a way that should prevent the problem. We describe the deployed version of Avalanche in Section 4.8. The next section describes another variation of Avalanche that provably eliminates the problem.

4.6 Fixing liveness with Glacier

The adversary is able to delay the acceptance of an honest transaction T because T is directly influenced by the queries of its descendants. Note the issuer of T has no control over its descendants according to the protocol. A unsuccessful query of a descendant of T carries a negative consequence for the acceptance of T, regardless of the status of T inside its conflict set. This influence is the root of the problem described earlier. An immediate, but inefficient remedy might be to run one Snowball consensus instance for each transaction. However, this would greatly degrade the throughput and increase the latency of the protocol, as many more messages would be exchanged.

We propose here a modification, called *Glacier*, in which an unsuccessful query of a transaction T carries negative consequences *only* for those of its ancestors that led to negative votes and caused the query to be unsuccessful. Our protocol is shown in Algorithm 11. It specifically modifies the voting protocol and adds to each VOTE message for T a list L with all ancestors of T that are not preferred in their respective conflict sets (L 188–192). When party P_i receives a negative vote like $[\text{VOTE}, v, T, \text{FALSE}, L]$, it performs the same actions as before. Additionally, it increments a counter for each ancestor T^* of T to denote how many parties have reported T^* as not preferred while accepting T (L 199). If P_i receives a positive vote, the protocol remains unchanged.

If the query is successful because P_i receives at least α positive votes on T, then it proceeds as before (Algorithm 8, L 118). But before P_i declares the query to be unsuccessful, it furthermore waits until having received a vote on T from all k parties sampled in the query (L 200). When this is the case, P_i only resets the counter for acceptance of those ancestors T^* of T that have been reported as non-preferred by more than $k - \alpha$ queried parties (L 204–206). If T^* is preferred by at least α parties, however, then P_i increments its confidence level as before (L 208).

Considering the adversary introduced in Section 4.5.3, a negative reply to the query of a descendant of the target transaction T does not carry any negative consequence for the acceptance of T here. In particular, the counter for acceptance of transaction T is never reset, even when a query is unsuccessful, because T is the only transaction in its conflicting

set, then always preferred. Thus, transaction T will be accepted after β_1 successful queries, if all its parents are accepted, or β_2 successful queries if they are not accepted. Assuming that queries of honest transactions are successful, on average $\frac{\beta}{1-p}$ transactions are required to accept T for $\beta \in [\beta_1, \beta_2]$ depending on the state of the parents of T. For simplicity we assume that the parents are accepted, thus, the counter needs to achieve the value β_1. If this were not the case, then it is sufficient to substitute β_1 with β in the upcoming expression. Avalanche requires on average $\beta_1 + \frac{1+(2+\beta_1\gamma)(1-\gamma)^{\beta_1}-(1-\gamma)^{2\beta_1}(1+\beta_1\gamma)}{\gamma(1-\gamma)^{\beta_1}(1-(1-\gamma)^{\beta_1})}$ transactions to accept T by Lemma 19. The assumption that the query of honest transactions is always successful is more beneficial to Avalanche than to Glacier, since in Avalanche such a query resets the counter for acceptance of T. But in Glacier, the query simply leaves the counter as it is. The value of the acceptance threshold β_1 is also more beneficial for Avalanche since the number of required transactions increases linearly in Glacier and exponentially in Avalanche. Figure 4.2 shows a comparison of both expressions.

In Glacier, the vote for a transaction is independent of the vote of its descendant and ancestors, even if a query of a transaction carries an implicit query of all its ancestors. Thus, Lemma 15 can be extended.

Lemma 20. *Party P_i delivers a transaction T with counter for acceptance $cnt[conflictSet[T]] \geq \beta_1$ in Glacier if and only if P_i decides 1 in the equivalent execution of Snowball with threshold $cnt[conflictSet[T]]$.*

Proof. Consider a transaction T in the equivalent execution of Snowball. The counter for acceptance of the value 1 in Snowball is always the same as the counter for acceptance of transaction T in Glacier because of the modifications introduced by Glacier. Thus, following the same argument as in Lemma 15, transaction T is accepted in Glacier with counter $cnt[conflictSet[T]]$ if and only if 1 is decided with counter $cnt[conflictSet[T]]$ in the equivalent execution of Snowball. □

Theorem 21. *The Glacier algorithm satisfies the properties of generic broadcast in the presence of an adversary that controls up to $\mathcal{O}(\sqrt{n})$ parties.*

Proof. Lemma 15 is a a special case of Lemma 20. Theorem 16 shows that Lemma 15 and the properties of Snowball (Section 4.7) guarantee

that Avalanche satisfies integrity, partial order, and external validity. In the same way, Lemma 20 guarantees that Glacier satisfies these same properties. Thus, it is sufficient to prove that Glacier satisfies validity and agreement.

Validity: Assume that an honest party broadcasts a payload *tx*. Because the party is honest, the transaction T containing *tx* is valid and non-conflicting. In the equivalent execution of Snowball, every honest party that proposes a value proposes 1. Hence, using the validity and termination properties of Snowball, every honest party eventually decides 1. Using Lemma 20, every honest party eventually delivers *tx*.

Agreement: Assume that an honest party delivers a payload transaction *tx* contained in transaction T. Using Lemma 20, an honest party decides 1 in the equivalent execution of Snowball. Because of the termination and agreement properties of Snowball, every honest party decides 1. Using Lemma 20 again, every honest party eventually delivers payload *tx*

We conclude that Glacier satisfies the properties of generic broadcast. □

With the modification to Glacier, Avalanche can be safely used as the basis for a payment system. The only possible concern with Glacier could be a decrease in performance compared to Avalanche. However, Glacier does not reduce the performance but rather improves it. Glacier only modifies the update in the local state of party P_i after a query has been unsuccessful. The counter of acceptance of a given transaction T in Glacier implementation is always greater or equal than its counterpart in Avalanche. This follows because a reset of $cnt[conflictSet[T]]$ in Glacier implies the same reset in Avalanche. Such a reset in Glacier occurs if the query of a descendant of T fails and T was reported as non-preferred by more than $k - \alpha$ parties, whereas in Avalanche it is enough if the query of the descendant failed. In Avalanche, $cnt[conflictSet[T]]$ is incremented if the query of a descendant of T succeeds, and the same occurs in Glacier. Thus, $cnt[conflictSet[T]]$ in Glacier is always greater or equal than in Avalanche. We recall that a transaction is accepted when $cnt[conflictSet[T]]$ reaches a threshold depending on some conditions of the local view of the DAG, but these are identical for Glacier and

Avalanche. Hence, every transaction that is accepted in Avalanche is accepted in Glacier with equal or smaller latency. This implies not only that the latency of Glacier is smaller than the latency of Avalanche, but also that the throughput of Glacier is at least as good as the throughput of Avalanche.

4.7 The Snowball protocol

The Snowball protocol is the Byzantine-resistant protocol introduced together with Avalanche in the whitepaper [99]. We shortly summarize this protocol in Algorithm 12.

Snowball possesses almost the same structure as Avalanche, but it is a protocol for consensus, not for broadcast. This Byzantine consensus primitive is accessed through the events *propose*(b) and *decide*(b). Any party is allowed to propose a value $b \in \{0, 1\}$. Since Snowball is a probabilistic algorithm, the properties of our abstraction need to be fulfilled only with all but negligible probability.

Definition 21. A protocol solves *Byzantine consensus* if it satisfies the following conditions, except with negligible probability:

Validity: If all parties are honest and *propose* the same value P_j, then no honest party decides a value different from P_j; furthermore, if some party decides P_j, then P_j was proposed by some party.

Termination: Every honest party eventually *decides* some value.

Integrity: No honest party *decides* twice.

Agreement: No two honest parties *decide* differently.

At the beginning of the protocol, party P_i may propose a value b, if P_i does not propose $b = \perp$. Snowball is structured in rounds around the same sample mechanism as Avalanche. Party P_i starts a round by sampling k parties at random and querying them for the value they are currently considering for b. If the value of a queried party is undefined, the queried party adopts the value that it is been queried with and replies accordingly. If more than α votes for value b' are collected, the query is finalized and P_i updates its local state by incrementing $d[b']$.

If b' is the same value as the value b in the local view of P_i, the counter for acceptance is incremented. However, if $b' \neq b$ and $d[b'] > d[b]$, the party updates its local value b and resets the counter to zero. Party P_i finishes consensus when the acceptance counter reaches the value β.

Considering the adversary introduced in Section 4.3.2 controlling up to $O(\sqrt{n})$ parties, it can be shown that Snowball satisfies the properties of Byzantine consensus.

Theorem 22. *Snowball satisfies Byzantine consensus.*

Proof. The proof is provided in the Appendix A of the whitepaper of Avalanche [99]. □

Algorithm 11 Modifications to Avalanche (Algorithm 6–9) for Glacier (party P_i)

State

186: $nonpref : HashMap[\mathcal{T} \times \mathcal{T} \to \mathbb{N}]$
// votes on T reporting that T' is not preferred

// replaces L 114
187: **upon** receiving message [QUERY, T] from party P_j **do**
188: $L \leftarrow [\,]$ // contains the non-preferred ancestors of T
189: **for** $T' \in ancestors(T)$ **do**
190: **if** $\neg preferred(T')$ **then**
191: append T' to L
192: send message [VOTE, v, T, $stronglyPreferred(T)$, L] to party P_j

// replaces code at L 116
193: **upon** receiving message [VOTE, v, T, w, L] from a party $v \in \mathcal{S}[T]$ **do**
194: $votes[T, v] \leftarrow w$
195: **for** $T' \in L$ **do**
196: **if** $nonpref[T, T'] = \perp$ **then**
197: $nonpref[T, T'] \leftarrow 1$
198: **else**
199: $nonpref[T, T'] \leftarrow nonpref[T, T'] + 1$

// replaces code at L 133
200: **upon** $\exists T \in \mathcal{T}$ such that $\left|votes[T, v]\right| = k$
$\wedge \left|\left\{v \in \mathcal{S}[T] \mid votes[T, v] = \text{FALSE}\right\}\right| > k - \alpha$ **do**
201: **stop** $timer[T]$
202: $votes[T, *] \leftarrow \perp$ // remove all entries in $votes$ for T
203: $S[T] \leftarrow [\,]$ // reset the HashMap S
204: **for** T' such that $nonpref[T, T'] \neq \perp$ **do**// all ancestors of T
205: **if** $nonpref[T, T'] > k - \alpha$ **then**
206: $cnt[conflictSet[T']] \leftarrow 0$
207: **else** // $nonpref[T, T'] \leq \alpha$
208: $cnt[conflictSet[T']] \leftarrow cnt[conflictSet[T']] + 1$
209: $nonpref[T, *] \leftarrow \perp$

Algorithm 12 Snowball (party P_i)

Global parameters and state

210: $\mathcal{N}$ // set of parties
211: $newRound \in \{\text{FALSE}, \text{TRUE}\}$ // when to start a round
212: $decided \in \{\text{FALSE}, \text{TRUE}\}$ // when to finish the protocol
213: $k \in \mathbb{N}$ // number of parties queried in each poll
214: $\alpha \in \mathbb{N}$ // majority threshold for queries
215: $cnt \in \mathbb{N}$ // counter for acceptance
216: $\beta \in \mathbb{N}$ // threshold for acceptance
217: $\mathcal{S} : HashMap[\mathcal{T} \to \mathcal{N}]$ // set of sampled parties to be queried
218: $d : HashMap[\{0,1\} \to \mathbb{N}]$ // confidence value of a transaction
219: $votes : HashMap[\{0,1\} \to \mathbb{N}]$ // number of votes for a value

Algorithm

220: **upon** $propose(b')$ **do**
221: $decided \leftarrow$ FALSE
222: $newRound \leftarrow$ TRUE
223: $b \leftarrow b'$

224: **upon** $newRound \wedge \neg decided$ **do** // still not decided
225: $newRound \leftarrow$ FALSE
226: $votes[*] \leftarrow 0$
227: **if** $b \neq \perp$ **then**
228: $\mathcal{S} \leftarrow sample(\mathcal{N} \setminus \{P_i\}, k)$ // sample k random parties
229: *send* message [QUERY, b] to all parties $k \in \mathcal{S}$

Algorithm 13 Snowball (party P_i)

230:**upon** $votes[b'] \geq \alpha$ **do** // $b' = 0$ or $b' = 1$
231: $d[b'] \leftarrow d[b'] + 1$
232: **if** $b = b'$ **then** // the outcome is our proposal
233: $cnt \leftarrow cnt + 1$
234: **else**
235: **if** $d[b'] > d[b]$ **then**
236: $b \leftarrow b'$
237: $cnt \leftarrow 1$
238: $newRound \leftarrow$ TRUE

239:**upon** $n = k \wedge votes[0] < \alpha \wedge votes[1] < \alpha$ **do** // no majority
240: $cnt \leftarrow 0$
241: $newRound \leftarrow$ TRUE

242:**upon** receiving message [QUERY, b'] from party k **do**
243: **if** $b = \perp$ **then**
244: $decided \leftarrow$ FALSE
245: $b \leftarrow b'$
246: *send* message [VOTE, b] to party k // reply with b

247:**upon** receiving message [VOTE, b'] from a party $k \in \mathcal{S}$ **do**
248: $votes[b'] \leftarrow votes[b'] + 1$

249:**upon** $cnt = \beta \wedge \neg decided$ **do** // enough confidence for b
250: $decide(b)$
251: $decided \leftarrow$ TRUE

4.8 The Avalanche protocol as implemented

The actual implementation of Avalanches addresses the liveness problems differently from Glacier and works as follows. Consider the protocol in Section 4.4. In the implementation, a party P_i queries k parties with transaction T as before. When some party P_j is queried with T, then P_j first adds T to its local view. Then it replies with a VOTE message, but instead of including a binary vote, it sends the whole virtuous frontier according to its local view. Party P_i collects the responses as in Section 4.4 and stores the virtuous frontiers received in the VOTE messages. Then it counts how many queried parties have reported some transaction T', or a descendant of T', as part of their virtuous frontier in the variable $ack[T, T']$. If more than α parties have reported T', then P_i adds T' to the set $\mathcal{G}[T']$ and updates its state, as for a successful query in the original protocol (L 271–277). For the remaining transactions, i.e., all the transactions in $\mathcal{Q}$ outside $\mathcal{G}[T']$, the counter is set to zero (L 279–281); this is equivalent to the effect of a negative query in the original protocol.

This fix addresses the liveness issue shown in Section 4.5.2 since a potential adversary loses the ability to submit a transaction that causes a reset of the acceptance counter of an honest transaction. As explained in Section 4.3.2, the adversary could reset the counter for acceptance of honest transactions by simply submitting a transaction T referring to the target transaction T and both transactions of a double spending T_1 and T_2 because T is not strongly preferred. However, in the implemented protocol, the parties reply with the virtuous frontier regardless of the queried transaction. Due to this, the adversary cannot influence the reply of the queried parties by creating malicious transactions.

Since the Cortina update (April 6th, 2023)[3], the Avalanche platform uses Snowman[4], a linearization of the Avalanche consensus, as basis for their distributed ledger.

A detailed analysis of this protocol is beyond the scope of this work.

[3] https://www.avax.network/blog/cortina-x-chain-linearization
[4] https://github.com/ava-labs/avalanchego/blob/master/snow/README.md

Algorithm 14 Modifications to Avalanche (Algorithm 6–9) in the Avalanche implementation (party P_i)

State

252: $ack : \mathit{HashMap}[\mathcal{T} \times \mathcal{T} \to \mathbb{N}]$
// number of votes on T reporting that T' is not preferred

253: $\mathcal{G} : \mathit{HashMap}[\mathcal{T} \times \mathcal{N} \to 2^{\mathcal{T}}]$
// ancestors of positively reported transactions

// replaces code at L 114

254: **upon** receiving message [QUERY, T] from party P_j **do**
255: **if** $T \not\in \mathcal{T}$ **then** // party P_i sees T for the first time
256: *updateDAG*(T)
257: send message [VOTE, u, T, $\mathcal{VF}$] to party P_j

// replaces code at L 116

258: **upon** receiving message [VOTE, v, T, $\mathcal{VF}'$] from a party $v \in \mathcal{S}[T]$ **do**
// $\mathcal{VF}'$ is the new vote
259: $votes[T, v] \leftarrow \mathcal{VF}'$
260: **for** $T' \in votes[T, v]$ **do**// ancestors of the reported transactions
261: $\mathcal{G}[T, v] \leftarrow \mathcal{G}[T, v] \cup ancestors(T')$
262: **for** $T' \in \mathcal{G}[T, v]$ **do** // number of parties that reported T'
263: **if** $ack[T, T'] = \perp$ **then**
264: $ack[T, T'] \leftarrow 1$
265: **else**
266: $ack[T, T'] \leftarrow ack[T, T'] + 1$

Algorithm 15 Modifications to Avalanche (Algorithm 6–9) in the Avalanche implementation (party P_i)

```
    // replaces code at L 118 and L 133
267:upon ∃T ∈ 𝒯 such that |{votes[T, v]}| = k    // all parties replied
268:     stop timer[T]
269:     votes[T, ∗] ← ⊥                  // remove all entries in votes for T
270:     for T′ ∈ 𝒬 do
271:         if ack[T, T′] ≥ α then
272:             d[T′] ← d[T′] + 1
273:             if d[T′] > d[pref[conflictSet[T′]]] then
274:                 pref[conflictSet[T′]] ← T′
275:             if T′ ≠ last[conflictSet[T′]] then
276:                 last[conflictSet[T′]] ← T′
277:                 cnt[conflictSet[T′]] ← 1
278:             else
279:                 cnt[conflictSet[T′]] ← cnt[conflictSet[T′]] + 1
280:         else
281:             cnt[conflictSet[T′]] ← 0
282:     ack[T, ∗] ← ⊥                    // remove all entries in ack for T
283:     𝒢[T, ∗] ← ⊥                      // remove all entries in 𝒢 for T
```

4.9 Conclusion

Avalanche is well-known for its remarkable throughput and latency that are achieved through a metastable sampling technique. The pseudocode we introduce captures in a compact and relatively simple manner the intricacies of the system. We show that Avalanche, as originally introduced, possesses a vulnerability allowing an adversary to delay transactions arbitrarily. We also address such vulnerability with a modification of the protocol, Glacier, that allows Avalanche to satisfy both safety and liveness.

The developers of Avalanche have acknowledged the vulnerability, and the actual implementation does not suffer from it due to an alternative fix. Understanding this variant of Avalanche remains open and is subject of future work.

Chapter 5

DAG superiority

> These big words are all very impressive, but where are the results?
>
> Obelix

5.1 Introduction

In the ever-evolving landscape of distributed systems, achieving consensus among a set of parties has become a fundamental challenge that has garnered significant attention in recent years. Consensus protocols are a universal primitive in distributed computing, ensuring that a network of interconnected parties can collectively agree on a shared state despite potential failures or malicious actors. However, as the demands on distributed systems continue to grow, the need for consensus protocols that can deliver both higher throughput and lower latency has become increasingly pressing. This need is particularly relevant in permissionless consensus protocols as used by cryptocurrencies and blockchain protocols, which face stringent demands on their throughput and latency.

Traditional consensus protocols have exhibited considerable advancements in both throughput and latency since the first practical consensus protocols [77, 33]. One of the most promising lines of work are DAG consensus protocols as introduced by the "All you need is DAG" paper [63] and subsequently extended by Narwhal and Tusk [45], Bullshark [105], and Cordial Miners [64]. A common characteristic of these protocols is their capacity to enable every participant to generate blocks that reference previous blocks, forming a *directed acyclic graph* (*DAG*). In permissionless protocols like Bitcoin [88], every party (miner) can create a block upon successfully solving the cryptographic puzzle. Therefore, the concept of constructing a DAG that is later ordered, as proposed by Keidar et al. [63], holds the potential to enhance the throughput and latency of permissionless consensus protocols. In essence, DAG protocols may surpass traditional permissionless consensus protocols, which form a chain.

The evident approach to improving the throughput of chain protocols is to increase the block ratio, i.e., the number of block produced per unit of time, effectively accelerating the execution of the protocol as there is less time between created blocks. This goal can be pursued by lowering the difficulty in *Proof-of-Work* (*PoW*) protocols. However, increasing the block ratio may harm the protocol since it elevates the likelihood of forks where two different parties create blocks extending the chain. An abandoned block is one that is never output by the protocol, whenever a chain protocol forks, an abandoned block is produced. Therefore, despite the increased number of generated blocks, the number of abandoned blocks concurrently rises, adversely affecting the protocol's throughput. Moreover, it is imperative to recognize that the block ratio cannot be augmented arbitrarily without compromising the protocol's security.

In this chapter, we introduce a construction that takes as input a DAG protocol or a chain protocol Π, which may produce abandoned blocks, and produces a new DAG protocol Π' with the property that every created block is eventually output. Specifically, Π' creates the same number of blocks as the base protocol Π and outputs *every* created block of Π. We show that the safety and liveness of Π' reduces to the safety and liveness of Π. In simpler terms, Π' is as safe and live as Π. Furthermore, we establish that Π' has lower or equal *latency* as Π, while achieving strictly higher *throughput*. Our main contribution lies in a formal proof that chain protocols cannot achieve optimal throughput,

i.e., for any chain protocol Π, there is a DAG protocol Π' that is safe and life under the same assumptions as Π, with the same or better latency and better throughput.

Acknowledgement. The material contained in this chapter corresponds to the work 'We will DAG you' [7] submitted at FC24.

5.2 Related work

DAG protocols represent a recent breakthrough within the domain of permissioned consensus protocols [63, 45, 105, 64]. While DAG protocols have been previously introduced in the permissionless context, their adoption and success have been somewhat restrained due to their inherent complexity when compared to traditional chain protocols. Several well-known DAG protocols have exhibited vulnerabilities, highlighting challenges in their success. For instance, IOTA [95], one of the pioneering DAG protocols, has been susceptible to vulnerabilities such as Parasite-chain attacks [95, 94]. Another promising protocol, GhostDAG [103], has also revealed vulnerabilities in its design [79]. Even Avalanche [99], the most successful DAG protocol in terms of market capitalization, originally had vulnerabilities in its design [10].

An intriguing DAG protocol to note is Conflux [80], which leverages the GHOST consensus rule [104] and augments blocks with additional references to transform a chain protocol into a DAG. Li et al. [80] have demonstrated that Conflux's security is directly inherited from the security of GHOST. However, it is worth mentioning that the GHOST protocol has exhibited lower resilience than other consensus protocols in the presence of network malfunctions [89, 16].

Our contribution to this landscape is a formal proof of the superior performance of DAG protocols, facilitated by a construction that can be conceptualized as an extension of the Conflux construction [80]. Specifically, when we instantiate the throughput closure using GHOST [104], we arrive at Conflux [80].

5.3 Abstractions

We consider a set of n *parties* $\mathcal{P} = \{P_1, P_2, \ldots\}$ that interact with each other by exchanging messages through the network. A protocol Π for $\mathcal{P}$ consists of a collection of programs with instructions for all parties. In particular we are interested in the study of *chain protocol* and *DAG protocol* protocols, i.e., protocol that rely on a chain or a DAG to deliver blocks. These two concepts are formally defined below.

Chain and DAG protocols are pivotal tools employed to establish robust and secure ledgers, and as such, they must adhere to specific fundamental requirements.

Traditionally, the gold standard concept is *atomic broadcast* [27], which ensures that all parties deliver the same set of transactions in the same order. In this chapter, we consider *block-based atomic broadcast* (Definition 4) that includes the concept of a *block* in the interface and properties [6]. Concept that we recall below.

Definition 4 (Block-based atomic broadcast). A protocol implements *block-based atomic broadcast* with validity predicates $VT()$ and $VB()$ and block-creation function $FB()$ if it satisfies the following properties, except with negligible probability:

Validity: If a correct party invokes a *bab-broadcast*(tx), then every correct party eventually outputs *bab-deliver*(b), for some block b that contains tx.

No duplication: No correct party outputs *bab-deliver*(b) for a block b more than once.

Integrity: If a correct party outputs *bab-deliver*(b), then it has previously output the event *bab-mined*($b, \cdot$) exactly once.

Agreement: If some correct party outputs *bab-deliver*(b), then eventually every correct party outputs *bab-deliver*(b).

Total order: Let b and b' be blocks, and P_i and P_j correct parties that output *bab-deliver*(b) and *bab-deliver*(b'). If P_i delivers b before b', then P_j also delivers b before b'.

External validity: If a correct party outputs *bab-deliver*(b), such that $b = [tx_1, \ldots, tx_m]$, then $VB(b) = \text{TRUE}$ and $VT(tx_i) = \text{TRUE}$,

for $i \in 1, \ldots, m$. Moreover, if $FB(tx_1, \ldots, tx_m)$ returns b, then $VB(b) = \text{TRUE}$.

The block-based atomic broadcast abstraction can be implemented by protocols based on different approaches. These difference are not captured in Definition 4, but can relevant for the performance of the protocol. The two families of protocols of interest for this chapter are *chain protocol* and *DAG protocol* protocols. The distinguishing factor between them lies in the set of references to previously mined blocks. Specifically, for a given block b, we denote the set of *bab-mined* blocks referenced by b as *parents*(b), commonly known as the *parents* of b. Furthermore, the set of *bab-mined* blocks reachable through references from b is represented as *ancestors*(b) and is often referred to as the *ancestors* of b. A block b is a *descendant* of its ancestors. A block with no descendants is also called *leaf*.

Definition 22 (Chain protocol, DAG protocol). A block-based atomic broadcast protocol Π is a *DAG protocol* protocol if Π-*mined* blocks contain references to other Π-*mined* blocks, meaning that the set of references is not empty. Π is a *chain protocol* protocol if every Π-*mined* block refers to exactly one Π-*mined* block and for every honest party P_i there is a Π-*delivered* block b such that every Π-*delivered* by P_i is b or in *ancestors*(b). In essence, Π-*delivered* blocks form a chain.

Figure 5.1 illustrates an example of both chain and DAG protocols.

To set the stage, we make the assumption that both chain and DAG protocols begin with an initial, hard-coded block referred to as the *genesis* block. This genesis block is special in that it possesses an empty set of references. It is important to note that, according to Definition 22, chain protocols inherently are DAG protocols. The blocks mined in chain protocols produce a *tree*, a particular kind of DAG. Therefore, for the remainder of this chapter, we will use the term "DAG protocol" to encompass both DAG protocol and chain protocols, acknowledging this inclusion.

One significant implication of abstracting DAG protocols as block-based atomic broadcast (Definition 4) is that the protocol must define a function that operates on the directed acyclic graph (DAG) that produces a list of delivered blocks. It is worth mentioning that certain DAG protocols, such as the original Avalanche protocol [99, 10], do not output an

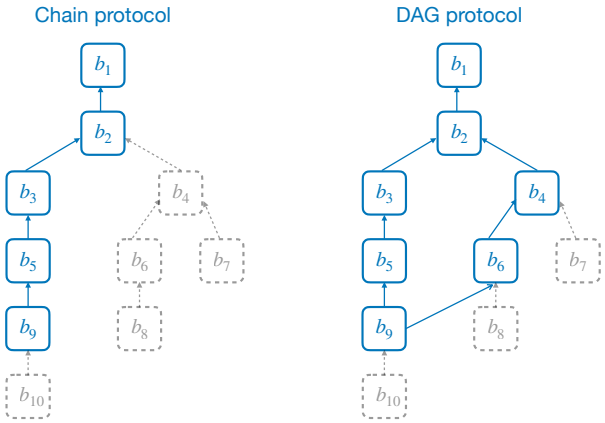


Figure 5.1. Comparison between a chain protocol and a DAG protocol. Blocks in blue (continuous lines) are the *bab-delivered* blocks, whereas grey (dashed) blocks are *bab-mined* but not *bab-delivered*. The protocol on the left is a chain protocol, each block refers to exactly one block and there is a block (b_9) such that every currently *bab-delivered* block is b_9 or an ancestor of it. The protocol on the right is a DAG protocol, block b_9 references multiple blocks.

ordered list of transactions but the list output by different parties may differ up to permutation. While DAG protocols can also be modeled as generic broadcast [93], situations arise where complete transaction ordering, as seen in calls to smart contracts, becomes necessary. For the purposes of this chapter, we focus on protocols that can be effectively modeled as block-based atomic broadcast. The results we derive in this context generalize straightforwardly to protocols modelled as generic broadcast.

5.4 Model

DAG protocols base their security on different techniques such as *proof of work* (PoW), *proof of stake* (PoS) [46], *proof of space-time* ($PoST$) [40], or *proof of elapsed time* ($PoET$) [24]. For the sake of simplicity, we restrict our model to PoW. Nevertheless, our model can readily be extended to incorporate other techniques.

Parties. Consistent with prior chapters, our protocol operates without explicit knowledge of the number or identities of the parties. The parties themselves remain unaware of these details as well. We assume a static network consisting of n parties, where up to f parties to be corrupted by the adversary, thereby exhibiting arbitrary behavior.

Blocks. A transaction tx, comprises a set of *inputs*, a set of *outputs*, and a collection of digital signatures, as in Bitcoin [88]. Transactions have size $|tx|$, and they are grouped into blocks, as introduced in Definition 4. Each block encompasses a specific number of transactions, denoted as m, a number of references to previously *bab-mined* blocks, quantified as n_{refs}, and further parameters essential for the proper execution of protocol Π. It is noteworthy that the size of a reference, represented as $|ref|$, is significantly smaller than that of a transaction, for simplicity, we consider it to be negligible. We reiterate that protocol Π defines external validity predicates, $VT()$ and $VB()$, responsible for determining the *validity* of a transaction or block.

Network & adversary. We consider the *synchronous-rounds* model (Section 2.2.1) and a *Byzantine* adversary (Section 2.1).

5.4.1 Abandoned blocks

Definition 23 (Execution). An *execution* is a history with an entry for each round containing the actions, a list of received messages, and a list of sent messages by each party in that round.

While an event may be theoretically possible within an execution, its occurrence might have a probability of zero. For instance, consider an algorithm that continuously flips an unbiased coin indefinitely. There could be an execution where all outcomes are heads, but the probability of this specific sequence of events happening is zero, as it is the limit of an infinite execution.

To circumvent these issues, we introduce the concept of a *partial execution*.

Definition 24. Given a protocol Π, the set of λ-*partial executions* Φ_λ is defined to be the set of λ-prefixes of all executions of protocol Π. A *partial execution* is an execution that belongs to Φ_λ for some $\lambda \in \mathbb{N}$.

Definition 25. Given an execution $\mathcal{E}$ of a block-based atomic broadcast protocol Π, an *abandoned* block in $\mathcal{E}$ is is an honestly *bab-mined* block b such that b is not *bab-delivered* in $\mathcal{E}$.

It is important to note that the validity property defined in block-based atomic broadcast (Definition 4) does not guarantee that every *bab-mined* block will eventually be *bab-delivered*. Instead, this property ensures that for each *bab-broadcast* transaction, there exists at least one *bab-delivered* block that contains it. The concept of abandoned blocks is a significant concern in the context of such protocols. Abandoned blocks have been honestly *bab-mined* but are never *bab-delivered*. The existence of abandoned blocks can severely impact the performance of a chain protocol or DAG protocol.

Definition 26. A protocol Π *permits abandoned blocks* if there exist a block b and a partial execution $\mathcal{E}$ such that: b is abandoned in any extension of $\mathcal{E}$.

Remark 3. Note that given a protocol that permits abandoned blocks, the probability, taken over the randomness of the protocol, of having at least one abandoned block in an execution is greater than zero, since partial executions happen with non-zero probability.

Determining whether a given protocol Π permits abandoned blocks or not can be a challenging task and, in some cases, may not be computable due to the need to simulate potentially infinitely long executions. However, for certain protocols like Bitcoin [88], the existence of abandoned blocks is a direct consequence forks occurring among honest miners. This phenomenon is formalized in the following definition.

Definition 27. Given an execution $\mathcal{E}$ of a given protocol Π, a round r *forked* if protocol Π outputs two events *bab-mined*(b, P_i) and *bab-mined*(b', P_j) in round r at two distinct honest parties P_i and P_j. A protocol with a forked round in at least one partial execution is a *forkable protocol*.

Lemma 23. *A forkable chain protocol* Π *permits abandoned blocks.*

Proof. Given a forkable protocol Π, there exist a round r in which two different honest parties output events *bab-mined*(b, P_i) and *bab-mined*(b', P_j). In particular $b \neq b'$ because their miners are different. Π is also a chain protocol. thus both b and b' have a unique reference to previously *bab-mined* blocks, so they cannot reference each other. Another implication of Π being a chain protocol is that at any point in the execution in the protocol there exists a *bab-mined* block b^* such that every *bab-delivered* is in *ancestors*(b^*). Since every block only contains a single reference and b and b' do not refer each other, we conclude that no honest parties can *bab-deliver* both b and b' simultaneously. □

Transactions that were originally included in abandoned blocks must be re-included in subsequent blocks to maintain the validity property (Definition 4). This re-inclusion consumes space in new blocks and has implications for both latency and throughput, as we formalize below.

5.4.2 Throughput and latency

Definition 28. Given a block-based atomic broadcast protocol Π, an adversary $\mathcal{A}$, and an execution $\mathcal{E}$, we define the *throughput* of Π in the presence of $\mathcal{A}$ in execution $\mathcal{E}$ as the average number of *bab-delivered* blocks per round and we denote by $\mathit{throughput}(\Pi, \mathcal{A}, \mathcal{E})$.

Definition 29. Given a block-based atomic broadcast protocol Π, the *throughput of* Π is defined to be $\mathit{throughput}(\Pi) := \inf_{\mathcal{A}} \mathrm{E}[\mathit{throughput}(\Pi, \mathcal{A}, \mathcal{E})]$, i.e., the infimum over all the possible adversaries $\mathcal{A}$ of the average over the randomness Π of $\mathit{throughput}(\Pi, \mathcal{A}, \mathcal{E})$ over all the possible executions.

Definition 30. The *goodput* of protocol Π is defined to be throughput of Π in the presence of an adversary that follows the instructions of the protocol.

Definition 31. Given a block-based atomic broadcast protocol Π, an adversary $\mathcal{A}$, an execution $\mathcal{E}$, and a transaction tx, we define *latency* of tx in the presence of adversary $\mathcal{A}$ in execution $\mathcal{E}$ as the number of rounds since tx is *bab-broadcast* until the first block containing tx is *bab-delivered*, and we denote it by $\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E}, tx)$. We define the *latency of* Π to be the average number of rounds, over the transactions tx in execution $\mathcal{E}$, since tx is *bab-broadcast* until the first block containing tx is *bab-delivered* and denote it by $\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E})$.

Definition 32. Given a block-based atomic broadcast protocol Π, The *latency* of protocol Π is defined as $\mathit{latency}(\Pi) = \sup_{\mathcal{A}} \mathrm{E}[\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E})]$, i.e., the supremum over all the possible adversaries $\mathcal{A}$ of the average over the randomness of the protocol of the $\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E})$ over the possible executions $\mathcal{E}$.

5.5 The throughput closure

We introduce a novel construction designed to enhance a given DAG protocol Π. This construction results in a DAG protocol, which we call *the throughput closure of* Π and denote by Π'. Protocol Π' possesses the unique property of ensuring that every honestly *bab-mined* block is

eventually *bab-delivered*. The mechanism by which protocol Π' accomplishes this feat involves the incorporation of additional references to blocks. For any given block b, protocol Π' defines the set *abandoned*(b) as the collection of valid blocks that will not be Π-*delivered* if b is to be Π-*delivered*. The block mining and delivery routines of the throughput closure Π' are built on top of their counterparts in Π.

Overview. As shown in Algorithm 16, when an honest party P_i Π-*mines* a block b, party P_i also Π'-*mines* the same block. However, in Π', the block b includes an additional set of references to the blocks in the set *abandoned*(b).

The modified delivery routine operates as follows: when a block b would be Π-*delivered*, all valid blocks in the set *abandoned*(b) are Π'-*delivered* in a fixed topological order immediately before b. This topological sort allows to order non Π-*delivered* blocks with respect to Π-*delivered* blocks deterministically according to the references included in the Π-*delivered* blocks. This is a crucial aspect as establishing a total order in a DAG can be generally challenging due to different parties having different partial views of the DAG. The topological sort τ ensure that all parties that have received block b agree on the same order. A canonical example for *topological sort* τ is to order the blocks in *abandoned*(b) according to their *depth* in the DAG, distance to genesis, breaking the ties according to the hash of the block. Note that if an adversary creates a block with low depth, it will be only Π-*delivered* when deeper block references it, thus the adversarial block is Π'-*delivered* concurrently with deeper blocks.

Constructing the set *abandoned*(b), even when it can be computed, may be challenging task, as we explained above. However, given a chain protocol Π the set *abandoned*(b) becomes trivial to compute as it is formed by every block that is not an ancestor of b. Furthermore, the set of references to *abandoned*(b) are the leaves of the DAG, with the exception of b. As an illustrative example, Figure 5.2 shows the application of this construction within the context of Bitcoin. If we consider Π to be GHOST protocol [104], we recreate the Conflux protocol [80]. Including references to the leaves in the DAG is the precise method for referring to the set *abandoned*(b) with a chain protocol Π. The same approach can be used with DAG protocols. This approach may be computationally

cheaper than than computing the leaves in set *abandoned*(b), however, some blocks may be referenced when there is no need, adding redundancy of references. Further insights into this alternative approach are provided below.

Detailed description. We describe the execution of the protocol from the perspective of an honest party P_i. When honest party P_i Π'-*broadcasts* a transaction tx, it invokes Π-*broadcast*(tx) (L 286–287). Notably, the broadcast of transactions occurs exactly as it does in protocol Π. When P_i triggers event Π-*mined*(b, P_i) (L 288–294), it initially computes the set *abandoned*(b) locally. To Π'-*mine* a new block b', P_i augments b by adding extra references to the leaves of the set *abandoned*(b) (L 290–292). Subsequently, P_i adds b' to the set of mined blocks $\mathcal{D}'$ (L 293) and triggers the event Π'-*mined*(b', P_i) (L 294).

When event Π'-*mined*(b', P_j) is triggered, P_i verifies the Π'-validity of b' and incorporates it into its local view (L 295–297). So far, the execution of Π' closely parallels that of Π. However, the key distinction lies in the delivery of blocks (L 298–302). When event Π-*deliver*(b) occurs, P_i searches for the block b' associated with b. P_i then assembles the set *ready*, which comprises the blocks to be Π'-*delivered* (L 299). This set is computed as the set-difference between the ancestors of block b' and the ancestors of the last delivered block b'_l. P_i subsequently updates the last delivered block to be b' (L 300). Finally, P_i applies a topological sorting algorithm τ to the set *ready* and Π'-delivers them accordingly (L 301–302).

A block b' is deemed valid (L 305–306) within protocol Π' if it satisfies two conditions: firstly, its associated block b must be Π-*valid*, and secondly, it must contain at least one Π'-valid transaction.

Algorithm 17 presents a greedy version of *abandoned*(b). In this approach, a party P_i adds references to b' for every block that is not already an ancestor of b within protocol Π.

The throughput closure mirrors protocol Π when the set *abandoned*(b) is empty for every block, indicating that the protocol does not permit the existence of abandoned blocks. However, if Π permits abandoned blocks, then there exists some executions of Π with a block b such that

Algorithm 16 Protocol Π' for party P_i.

Implements: block-based atomic broadcast Π'
Uses: block-based atomic broadcast Π
topological sort τ

State:
284: $\mathcal{D}' \leftarrow \emptyset$
285: $b'_\ell \leftarrow [\,]$

286:**upon event** Π'-*broadcast*(tx) **do**
287: **invoke** Π-*broadcast*(tx)

288:**upon event** Π-*mined*(b, P_j) **do**
289: **if** $P_i = P_j$ **then**
290: $weak \leftarrow leaves(abandoned(b, \mathcal{D}'))$
291: $b' \leftarrow b$
292: $bl'.wrefs \leftarrow weak$
293: $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{b'\}$
294: **invoke** Π'-*mined*(b', P_i)

295:**upon event** Π'-*mined*(b', P_j) **do**
296: **if** $VB'(b')$ **then**
297: $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{b'\}$

298:**upon event** Π-*deliver*(b) **do**
299: $ready \leftarrow ancestors'(b') \setminus ancestors'(b'_\ell)$
300: $b'_\ell \leftarrow b'$
301: **for** $b^* \in \tau(ready)$ **do**
302: **invoke** Π'-*deliver*(b^*)

303:**function** $abandoned(b, \mathcal{D}')$ **:**
304: **return** $\{b' \in \mathcal{D}' : b' \not\in ancestors'(b) \wedge incompatible(b, b')\}$

305:**function** $VB'(b')$ **:**
306: **return** $VB(b) \wedge \exists\ tx \in b' : undelivered(tx)$

Algorithm 17 Greedy approach for party P_i.

```
307:upon event Π-mined(b, P_j) do                          // Greedy approach
308:     if P_i = P_j then
309:          b' ← b
310:          weak ← leaves({b' ∈ D' : b' /∈ ancestors(b')})
311:          bl'.refs ← bl'.refs||weak
312:          D' ← D' ∪ {b'}
313:     invoke Π'-mined(b', P_i)
```

$abandoned(b) \neq \emptyset$, and the throughput closure diverges from the original protocol. The implementation of the throughput closure does entail an increase in local computation for parties. Specifically, parties need to scan the DAG and append a set of references to all leaves in $abandoned(b)$ to the currently mined block b. The computational complexity of determining $abandoned(b)$ can vary depending on the protocol, as discussed earlier. However, in the case of chain protocols, this set is relatively straightforward to compute. A party simply adds references to every leaf of a chain that has not been referenced by an ancestor.

5.6 Analysis

5.6.1 Security analysis

Theorem 24. *Given protocol DAG protocol* Π *implementing block-based atomic broadcast, its throughput closure* Π' *also implements block-based atomic broadcast.*

Proof. We demonstrate that the throughput closure Π' implements block-based atomic broadcast by leveraging the fact that Π does. Throughout this proof, we assume the perspective of an honest party P_i.

Validity: Assume that an honest party P_j Π'-*broadcasts* a given transaction tx. By construction, party P_j does so by invoking Π-*broadcast* transaction tx (L 286–287). The validity property

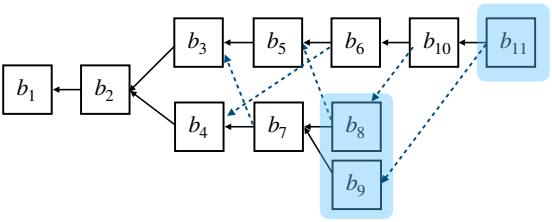


Figure 5.2. An example of our construction applied to Nakamoto consensus. The full lines denote the references of the Nakamoto consensus and the blue dashed lines denote the extra references included by the throughput closure. According to Nakamoto consensus, the main chain is the chain $b_1 \cdots b_{11}$ and blocks b_4, b_7, b_8, and b_9 are abandoned. Looking at b_{11}, the set $\mathit{abandoned}(b_{11})$ is formed by block b_9. Blocks b_4, b_7, and b_8 are not part of the $\mathit{abandoned}(b_{11})$ because b_{10} already references them. When delivering b_{11}, block b_9 would be delivered between b_{10} and b_{11}.

of protocol Π guarantees that party P_i eventually Π-*delivers* a block b containing transaction tx. Party P_i, by definition of the protocol, Π'-*delivers* the block b' consisting of block b with the addition of the extra set of references (L 298–302). If every transaction contained in b' is invalid, the block is not Π'-*deliver*. In the case of block b', the validity check can only fail if transaction tx fails the validity predicate. Since tx is Π'-*broadcast*, the external validity predicate is satisfied unless some block containing tx has been Π'*delivered*.

We conclude that for any honestly Π'-*broadcast* transaction tx, P_i eventually Π'-*delivers* a block b' containing tx, thus validity property of protocol Π' is satisfied.

Integrity: Party P_i only Π'-*delivers* blocks that it Π'-*delivers* or ancestors of those contained in the set $\mathcal{D}$ (L 298–302). A block b' enters the set $\mathcal{D}$ only after an invokation of Π-*mined*(b, P_j). We conclude that every Π'-*delivers* has previously been Π'-*mined*.

Agreement: Consider a block b' that is Π'-*delivered* by party P_i. We consider two different cases: when b whether b is Π-*delivered* or not. On the one hand, if b is Π-*delivered* by party P_i, every honest party eventually Π-*delivers* b, thus Π'-*delivers* b' as a consequence (L 298–302). On the other hand, if block bl' is Π'-*delivered* as a consequence of another block b^* is Π'-*delivered*. The same reasoning as above applies to block b^*, which implies the eventual Π'-*delivery* of block b'.

Total order: Consider two Π'*mined* blocks b'_1 and b'_2 and two honest parties P_i and P_j that $\Pi' - deliver$ both blocks. We distinguish four cases based on whether blocks b_1 and b_2 are Π-*delivered* or not.

Assume that both b_1 and b_2 are Π-*delivered*. Note that in the view of any honest party the order in which blocks b_1 and b_2 are Π-*delivered* is the same as blocks b'_1 and b'_2 are Π'-*delivered* (L 298–302). Due to the total order property of protocol Π, party P_i Π-*delivers* block b_1 and b_2 in the same order as party P_j, thus both parties Π'-*deliver* blocks b_1 and b_2.

If either b'_1 or b'_2 are Π'-*delivered* as a consequence of another block b_3 being Π-*delivered*. Since the set of blocks that are Π'-*delivered* as consequence of block b'_3 are Π'-*delivered* immediately before b'_3, any block b' Π'-*delivered* before (after) b' is also Π'-*delivered* before

(after) the set of blocks Π'-*delivered* as a consequence of b'. The same reasoning as above applies to this case. We conclude that P_i also Π'*delivers* both b'_1 or b'_2 in the same order as P_j.

The only case left is when both b'_1 and b'_2 are Π'-*delivered* as a consequence of two blocks b'_3 and b'_4 being Π'-*delivered*. If b'_3 and b'_4 are different the case is the same as before. If b'_3 and b'_4, both P_i and P_j use the topological order to determine in which order to Π'-*delivered*. Since the topological sorting is deterministic and depends only on block b'_3, both P_i and P_j Π'-*deliver* b'_1 and b'_2 in the same order.

External validity: The external validity property is imposed by lines L 305–306.

□

5.6.2 Throughput and latency

Theorem 24 states that the throughput closure Π' maintains the safety and liveness properties the original protocol Π. In this section, we delve into a comparative analysis of the performance aspects, through throughput and latency, between Π' and Π. It is important to note that both throughput and latency definitions take into account adversarial behavior, and the connection between the adversarial behavior of Π' and Π is discussed in the following remark.

Remark 4. Note that given an adversary $\mathcal{A}'$ for protocol Π', an adversary $\mathcal{A}$ for protocol Π can be constructed by merely removing the extra references from any block that $\mathcal{A}'$ Π'-*mines*. Additionally, given an adversary $\mathcal{A}$ for protocol Π, it can also be regarded as an adversary for protocol Π', as every action taken by $\mathcal{A}$ in protocol Π is allowed in protocol Π'.

Definition 33. Given an execution $\mathcal{E}'$ and an adversary $\mathcal{A}'$ for protocol Π', we define the equivalent execution of protocol Π as the execution $\mathcal{E}'$ without the extra references in each block and adversary $\mathcal{A}$ as discussed in Remark 4.

Lemma 25. *Given a DAG protocol* Π*, its throughput closure* Π' *achieves the same or lower latency as* Π*.*

Proof. Consider an execution $\mathcal{E}'$, an adversary $\mathcal{A}'$ for protocol Π', and a transaction tx that has not already been Π'-*delivered*. Denote by $\mathcal{E}$ the equivalent execution (Definition 33) of protocol Π. Note that by definition of Π', tx has not been Π-*delivered* either (L 298). Protocol Π' has two different mechanisms to Π'-*deliver*(tx).

On the one hand, if an event Π-*deliver*(b) for a block b containing transaction tx is triggered, then b is Π'-*delivered* (L 298). In this case, $\mathit{latency}(\Pi', \mathcal{A}', \mathcal{E}', tx)$ is the same as $\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E}, tx)$.

On the other hand, if an event Π-*deliver*(b') for a block b' that does not contains tx but is descendent of a block b containing tx., then block b' is Π'-*delivered* immediately before b (L 299). In this case, $\mathit{latency}(\Pi', \mathcal{A}', \mathcal{E}', tx)$ is strictly smaller than $\mathit{latency}(\Pi, \mathcal{A}, \mathcal{E}, tx)$.

Both cases discussed above imply that for every adversary, execution, and transaction, the latency of both protocols satisfy $\mathit{latency}(\Pi', \mathcal{A}', \mathcal{E}', tx) \leq \mathit{latency}(\Pi, \mathcal{A}, \mathcal{E}, tx)$. Hence, $\mathit{latency}(\Pi') \leq \mathit{latency}(\Pi)$ □

The next result clarifies the motivation for the term throughput closure.

Lemma 26. *Given a DAG protocol* Π*, then* $\mathit{throughput}(\Pi') \geq \mathit{throughput}(\Pi)$*, and if* Π *permits abandoned blocks, then* $\mathit{throughput}(\Pi') > \mathit{throughput}(\Pi)$*.*

Proof. Consider an execution $\mathcal{E}'$, an adversary $\mathcal{A}'$ for protocol Π', and a transaction tx that has not already been Π'-*delivered*. Denote by $\mathcal{E}$ the equivalent execution (Definition 33) of protocol Π.

On the one hand, if there is no abandoned block in the execution $\mathcal{E}'$, then, the set $\mathit{abandoned}(b')$ is empty for every block b'. Thus, no extra reference is added at any point in the execution of Π' the executions $\mathcal{E}$ and $\mathcal{E}'$ identical. We conclude that $\mathit{throughput}(\Pi, \mathcal{A}', \mathcal{E}) = \mathit{throughput}(\Pi', \mathcal{A}, \mathcal{E})$.

On the other hand, if there exists at least one abandoned block b' in execution $\mathcal{E}'$, then, the set $\mathit{abandoned}(b^*)$ is not empty for some block b^* that is eventually Π'-*delivered*. When b^* is Π'*delivered* so is b' (L 299).

We conclude that $throughput(\Pi', \mathcal{A}', \mathcal{E}') \geq throughput(\Pi', \mathcal{A}, \mathcal{E})$ for every possible adversary $\mathcal{A}'$ and execution $\mathcal{E}'$, thus $throughput(\Pi') \geq throughput(\Pi')$. Furthermore, if Π permits abandoned blocks, there exists an λ-partial execution with a block b that is abandoned in all its extensions. This means that the probability, over the randomness of the protocol, of having an abandoned block is strictly greater than zero (Remark 3). Thus, $\mathrm{E}[throughput(\Pi', \mathcal{A}', \mathcal{E}')] > \mathrm{E}[throughput(\Pi, \mathcal{A}, \mathcal{E})]$ for at least some adversary $\mathcal{A}'$. We conclude by noticing that if an adversary $\mathcal{A}^*$ prevents the exclusion of abandoned blocks, then $throughput(\Pi', \mathcal{A}^*, \mathcal{E}') > throughput(\Pi', \mathcal{A}', \mathcal{E}')$. Hence, we conclude that

$$\begin{aligned} throughput(\Pi') &= \inf_{\mathcal{A}'} \mathrm{E}[throughput(\Pi', \mathcal{A}', \mathcal{E}')] \\ &> \inf_{\mathcal{A}} \mathrm{E}[throughput(\Pi, \mathcal{A}, \mathcal{E})] = throughput(\Pi). \end{aligned}$$

□

Corollary 27. *Given a DAG protocol* Π*, then* $goodput(\Pi') \geq goodput(\Pi)$*. Furthermore, if* Π *allows for the existence of abandoned blocks, then* $goodput(\Pi') > goodput(\Pi)$*.*

Proof. Consider the proof of Lemma 26 limited to adversaries that follow the instructions of the protocol. □

Note that every chain protocol trivially permits abandoned block. We can finally conclude that DAG protocols are strictly better then chain protocols.

Theorem 28. *Given a chain protocol* Π*, there exists a DAG protocol* Π' *such that:* $latency(\Pi') \leq latency(\Pi)$ *and* $throughput(\Pi') > throughput(\Pi)$*.*

Proof. Lemma 23 states that a chain protocol Π permits abandoned blocks. Theorem 24 demonstrates that its throughput closure Π' implements block-based atomic broadcast. Lemma 26 establishes that $throughput(\Pi') > throughput(\Pi)$. Finally, Lemma 25 shows that $latency(\Pi') \leq latency(\Pi)$. □

5.7 Conclusion

This work introduced a construction that takes a blockchain or DAG protocol as input and produces a new DAG protocol with the same security guarantees. The construction output has better throughput and the same or lower latency when input with a blockchain protocol. Furthermore, this construction determines the set of possibly optimal protocols, showing that blockchain protocols are not optimal.

Chapter 6

An atomic broadcast protocol

> It's important to draw wisdom
> from many different places.
>
> Uncle Iroh

6.1 Introduction

Scalability poses a significant challenge for consensus protocols. Paradoxically, increasing the number of participants in a consensus system often degrades its performance rather than enhancing it.

Traditional consensus protocols, such as those Paxos [77], PBFT [33], or Hotstuff [111], primarily follow a *leader-based* approach. In this model, a single party proposes a value, and the remaining participants validate this proposal. Consequently, the protocol's workload becomes unevenly distributed among the participants, making the leader for a given instance a potential bottleneck. Another drawback emerges when the

chosen leader exhibits Byzantine behavior diverging from the prescribed protocol execution. In such cases, leader-based protocols require a *view-change* mechanism to select a new leader. View-change routines are highly sensitive to timing assumptions, and their operation can become prohibitively costly when the delay in message transmission is uncertain, as noted in [27]. Keidar et al. [63] initiated a line a work [45, 105, 64] that offers an elegant solution to the previously mentioned challenges by employing *leaderless* protocols. They recognized the value of the *common core* abstraction in designing an algorithm where every participant is granted the role of behaving as a leader in an instance.

The *common core* abstraction (Definition 7), as introduced by Canetti [32], can be understood as a variant of consensus, albeit weaker. In this variant, the outputs of different parties may differ, but there is a guaranteed existence of a *common core*, a set of specific size, included in the output of every honest party. Unlike traditional consensus, the key distinction here is that the exact contents of the common core remain unknown; its existence, however, is assured. The Gather protocol [32] serves as an implementation of the common core abstraction, utilizing *reliable broadcast* as its main building block.

Notably, the *reliable broadcast* primitive ensures that if two honest parties incorporate the value proposed by a party in their respective outputs, they both include the same value. In other words, it prevents honest parties from including different values corresponding to Byzantine parties. Another vital component of the leaderless approach is the *common coin*, employed to determine which values can be decided upon. If the party designated by the coin has proposed a value that is included in a sufficient number of outputs, that value can then be decided upon.

The first contribution of this work lies in the development of a weaker variant of the common core, termed *weak common core*. Notably, this novel concept does not require reliable broadcast, however, honest parties may include different values from a Byzantine party. The rest of the properties of the common core remain unaltered. The *weak common core* serves as a viable substitute for the standard common core in constructing leaderless consensus protocols, thanks to the presence of the common coin, which fortifies its robustness, compensating for the missing property. Importantly, since the weak common core does not require the use of reliable broadcast, it exhibits significantly reduced message complexity.

Comparison				
Protocol	**Network**	**Best**	**Average**	**Model**
PBFT	$1/n$	4	12	Part. Sync.
Hotstuff	$1/n$	4	12	Part. Sync.
DAG-Rider	~ 1	12	24	Async.
Narwhal	~ 1	10.5	20.5	Async.
Bullshark	~ 1	6	19.5	Part. Sync.
All set	~ 1	4	14	Async.

Table 6.1. Comparison of different leader-based and DAG protocols. The first column indicates the name of the protocol. The second column denotes the percentage of the network capacity of the parties is used during the execution, we assume a network of n parties. This way a network of $1/n$ means that only one n'th of the network is utilize. The third column denotes the best case latency, assuming that every party behaves honestly and randomness favours the protocol. The fourth column, considers the average latency in the presence of an adversary, average over all the randomness of the protocol. Both latencies are computed as rounds of communication. Finally the fifth columns denotes the network model.

The main contribution of this work is the design of a leaderless consensus protocol, modelled as *atomic broadcast*, built upon the foundation laid by the *weak common core*. This novel protocol exhibits resilience against up to a maximum of a third of the parties behaving maliciously and concurrently empowers high throughput by enabling every network participant to propose values, fully utilizing the capacity of the network. Furthermore, our protocol is characterized by its low latency, averaging just 14 rounds of communication, which decreases to 4 in the optimistic case (Table 6.1). We provide mathematical proofs to guarantee both *liveness* and *safety*, with formal verification accomplished with TLA [76].

Structure. In Section 6.3, we state the assumptions considered and we recall important concepts such as *atomic broadcast* and *common coin*. After that, we move to Section 6.4 to introduce the weak common core, an algorithm implementing the weak common core, and formally prove

its properties. We introduce the main contribution of this chapter in Section 6.5, starting with a high-level description, following with a detailed description, and concluding with a mathematical proof that our protocol implements *atomic broadcast*. After that, we conclude with a formal verification of such properties in Section 6.7.

6.2 Related work

The pursuit of scalability in consensus protocols has been the subject of extensive research in recent years. Buchman, Kwon, and Milosevic [26], endeavored to mitigate communication complexity by adopting a strategy of gossipping messages to specific parties instead of employing broadcasting, concluding an extensive set of improvements on traditional consensus protocols [101, 84]. Tendermint [26] and Avalanche [99] stand out as the most prominent protocols, to the best of our knowledge, that embrace this approach.

An alternative line of research, closely related to this work, was pioneered by Keidar et al. [63]. Their primary objective is to tackle the scalability challenge of consensus protocols through the implementation of a leaderless protocol known as *DAG-Rider*. This approach heavily relies on the *common core* [32] and *reliable broadcast* [25, 27] primitives. Keidar et al. [63] made substantial progress by significantly enhancing both scalability and throughput, albeit at the expense of increased latency. Subsequent work by Danezis et al. [45] involved a modification of DAG-Rider, where they separated transaction dissemination from the consensus layer, thereby reducing the overall protocol latency. However, their reliance on reliable broadcast prevented them from achieving a substantial reduction in latency. Spiegelman et al. [105] further trimmed latency by considering a variant of DAG-Rider in a *partially synchronous* network setting.

The most noteworthy reduction in latency was achieved by Keidar, Naor, and Shapiro [64] with their *Cordial Miners* protocol, which builds upon DAG-Rider but eliminates the need for reliable broadcast by introducing an additional round of communication. Both DAG-Rider and Cordial Miners leverage the properties of the common core as a black-box approach, mandating a minimum number of rounds in their protocols.

In contrast, our approach diverges in how we employ the common core. We consider a relaxation of the agreement property of the common core, which is subsequently factored into the block delivery routine.

Another orthogonal improvement of leaderless protocols is considered by Danezis et al. [45] and Spiegelman et al. [105]. This approach is centered around enhancing both the *throughput* and *adaptability* of the protocol. These protocols introduce the concept of *workers*, which accelerate the dissemination of transactions by generating *certificates of availability*, allowing the protocol to efficiently process hashes of transaction batches. This approach significantly boosts the number of transactions processed since the protocol exclusively deals with the hashes. Moreover, the ability to add or remove workers provides the protocol with the flexibility to adapt to the current system load.

Our protocol achieves similar results but eliminates the necessity for certificates of availability, leading to a reduced number of required signatures.

6.3 Model

We consider the Byzantine adversary (Section 2.1) in the asynchronous model (Section 2.2.2). We aim to build a protocol implementing atomic broadcast, definition that we recall below:

Definition 2 (Atomic broadcast) : *A protocol solves* atomic broadcast *with validity predicate V if it satisfies the following conditions, except with negligible probability:*

Validity: *If an honest party* ab-broadcasts *a block b, then it eventually* ab-delivers *b.*

Agreement: *If a honest party* ab-delivers *a block b, then all honest parties eventually* ab-deliver *b.*

Integrity: *For any block b, every honest party* ab-delivers *b at most once, and only if it was submitted by some user.*

Total order: *If honest parties P_i and P_j both* ab-deliver *blocks b and b', then p* ab-delivers *b before b' if and only if P_j* ab-delivers *b before b'.*

External validity: *If an honest party ab-delivers a block b, then* $V(b) = \text{TRUE}$.

Note that in this work we consider blocks denoted by b instead of transactions, this is due to implementation details that do not belong in this thesis. The properties remain unaltered.

Moreover, we also recall the concepts of common coin and common core from Chapter 2 since we use them as essential building blocks.

Definition 6 (Common coin). A protocol solves *common coin* with domain $\mathcal{D}$ and bias ε if it satisfies the following conditions, except with negligible probability:

Termination: Every correct party eventually *c-output*() a coin value.

Unpredictability: The probability that an adversary predicts the *c-output*() value before at least one honest party invokes *c-release*() is at most $\frac{1}{|\mathcal{D}|} + \varepsilon$.

Matching: With probability at least δ, every correct party *c-outputs* the same value. If $\delta = 1$, the coin is called *perfect*

No bias: If all correct parties *c-output*() the value, the distribution of the coin is uniform over $\mathcal{D}$.

As explained in Chapter 2, we assume perfect coins with ε arbitrarily small. Such coins can be constructed efficiently in a distributed setting [106].

Definition 7 (Common core). A protocol solves *common core* if it satisfies the following conditions with all but negligible probability:

Validity: Every honest party P_i eventually *delivers* a set U_i.

Common core: There exists a core set U^* of size at least $2f + 1$ that is included in the *delivered* set of every honest party.

Integrity: If honest party P_i includes (P_j, v_j) in its *delivered* set U_i, and j is honest, then v_j its input.

Agreement: If two honest parties include the pairs (P_j, v) and (P_j, v') in their *delivered* sets, then $v = v'$.

The agreement property of the common core requires protocols to use reliable broadcast [31], with the consequent latency tradeoff. The rest of the properties can be guaranteed without the use of reliable broadcast, as detailed in the following section.

6.4 Weak common core

We introduce a novel and more relaxed variant of the common core abstraction (Definition 7), which we refer to as the *weak common core*. The primary distinction between the weak common core and the standard common core abstractions lies in the agreement property. In the standard common core abstraction, when two honest parties include the value v_j from a specific party P_j, they unanimously concur on the same value. However, in the weak common core abstraction, if party P_j is Byzantine, these values may differ among honest parties.

Our weak common core abstraction is accessed through a single *core-broadcast*(P_i, v_i) event per party and one or multiple *core-deliver*(P_j, U_j) events. We consider a different interface where every party outputs a set corresponding to other parties in order to build a clearer description of our protocol. It is enhanced with a validity predicate that determines, whether an output set $\mathcal{U}$ is valid. Different instantiations of the validity predicate yield to different abstractions of the weak common core.

Definition 34. A protocol solves *weak common core* with validity predicate V if it satisfies the following conditions with all but negligible probability:

Validity: Every honest party P_i eventually outputs *core-deliver*(P_i, U_i).

Integrity: If an honest party P_i outputs the event *core-deliver*(P_j, U_j), then U_j is valid and for any value (P_k, v_k) contained in U_j with P_k honest, the value v_k has been *core-broadcast* by P_k.

Weak agreement: If any honest party outputs *core-deliver*(P_j, U_j) with honest P_j, then every honest party eventually outputs *core-deliver*(P_j, U_j). Furthermore, if two honest parties output events *core-deliver*(P_j, U_j) and *core-deliver*(P_j, U_j') with

P_j honest, then $U_j = U'_j$. Furthermore, any honest party *core-deliver*(P_j, U_j) at most once per party P_j.

Common core: There exists a core set U^* formed by *core-broadcast* values from at least $2f + 1$ different parties P_j that is included in every valid *core-delivered* set.

Self inclusion: If an honest party P_i outputs *core-deliver*(P_i, U_i), then U_i includes the value (P_i, v_i) *core-broadcast* by P_i.

The Gather Protocol [32], is a well-established protocol designed to implement the common core abstraction. Algorithm 18 introduced a modification of the Gather Protocol, obtained by substituting the reliable broadcast primitive with the bare broadcast of messages to all parties. Algorithm 18 implements the weak common core with a validity predicate denoted as V_c, as defined below.

A valid *core-delivered* set U_i is a set containing valid sets T_j created by at least $2f + 1$ different parties. A valid T_j contains at least $2f + 1$ sets S_k from different parties. Finally, a valid S_k must contain values from at least $2f + 1$ different parties. For simplicity, we assume that any value or set in the protocol contains information to verify its authenticity.

We describe the actions of honest party P_i according todo Algorithm 18. The protocol initiates as party P_i triggers the event *core-broadcast*(P_i, v_i) using its input value v_i and sends the message $[\text{FIRST}, P_i, v_i]$ (L 314–321) to every party. Subsequently, P_i awaits the receiving of input values from various parties (L 322–324). If P_i receives multiple values from the same party, only the first value is considered; this rule applies throughout this description.

Once P_i has received input values from at least $2f + 1$ parties, including P_i itself, P_i aggregates all these values into the set S_i and sends the message $[\text{SECOND}, (P_i, S_i)]$ to all parties (L 325–327). P_i then awaits the delivery of sets S_j from different parties (L 328–330) and compiles these sets into a new set named T_i.

Upon receiving sets S_j from at least $2f + 1$ parties, including itself, P_i sends the message $[\text{THIRD}, (P_i, T_i)]$ to all parties (L 331–333). Subsequently, P_i waits for sets T_j from different parties (L 328–330) and assembles these sets in an output set named U_i.

Algorithm 18 Weak common core (party P_i)

Initialization

314: **upon** *core-broadcast*(P_i, v_i) **do**
315: $S_i \leftarrow \emptyset$ // auxiliary set
316: $T_i \leftarrow \emptyset$ // auxiliary set
317: $U_i \leftarrow \emptyset$ // output set
318: $\mathcal{O}_i \leftarrow \emptyset$ // received output sets
320: $r \leftarrow 1$ // round counter
321: send $[\text{FIRST}, (P_i, v_i)]$ to every party // P_i included

Algorithm

322: **upon** receive valid $[\text{FIRST}, (P_j, v_j)]$ from party P_j **do**
323: **if** $\nexists (x, y) \in S_i : x = P_j$ **then** // only first received value
324: $S_i \leftarrow S_i \cup \{(P_j, v_j)\}$

325: **upon** $r = 1 \wedge |S_i| \geq 2f + 1 \wedge \mathit{self}(S_i)$ **do**
326: $r \leftarrow 2$
327: send $[\text{SECOND}, (P_i, S_i)]$ to every party

328: **upon** receive valid $[\text{SECOND}, P_j, S_j]$ from party P_j **do**
329: **if** $\nexists (x, y) \in T_i : x = P_j$ **then** // only first received value
330: $T_i \leftarrow T_i \cup \{S_j\}$

331: **upon** $r = 2 \wedge |T_i| \geq 2f + 1 \wedge \mathit{self}(T_i)$ **do**
332: $r \leftarrow 3$
333: send $[\text{THIRD}, (P_i, T_i)]$ to every party

334: **upon** receive valid $[\text{THIRD}, (P_j, T_j)]$ from party P_j **do**
335: **if** $\nexists (x, y) \in U_i : x = P_j$ **then** // only first received value
336: $U_i \leftarrow U_i \cup \{T_j\}$

337: **upon** $r = 3 \wedge |U_i| \geq 2f + 1 \wedge \mathit{self}(U_i)$ **do**
338: $r \leftarrow 0$
339: send $[\text{OUTPUT}, (P_i, U_i)]$ to every party

Algorithm 19 Weak common core (party P_i)

340:**upon** receive valid $[\text{OUTPUT}, (P_j, U_j)]$ **do**
341: **if** $\nexists (x, y) \in \mathcal{O}_i : x = P_j$ **then** // only the first received value
342: $\mathcal{O}_i \leftarrow \mathcal{O}_i \cup \{(P_j, U_j)\}$
343: *core-deliver*$(P_j, \textit{values}(U_j))$ // return the values

344:**function** *self*(S)**:**
345: **return** $\exists x \in S : x[1] = i$

Upon receiving sets T_j from at least $2f + 1$ parties, including itself, P_i sends the message $[\text{OUTPUT}, (P_i, U_i)]$ to all parties (L 337–339). When party P_i receives a message $[\text{OUTPUT}, (P_j, U_j)]$ from another party P_j, P_i triggers an output event *core-deliver*$(P_j, \textit{values}(U_j))$, with *values*$(U_j)$ encompassing the *core-broadcast* values within U_j (L 340–343). It is important to note that if P_i receives multiple $[\text{OUTPUT}, (P_j, U_j)]$ messages from the same P_j, only the first received message triggers the output event *core-deliver*$(P_j, \textit{values}(U_j))$.

Theorem 29. *Algorithm 18 implements weak common core with validity predicate* V_c.

Proof. The proof is structured by property.

Validity: Every honest party send a message $[\text{FIRST}, (i, v_i)]$ to every party (L 314–321). Thus, every honest party receives at least $2f+1$ valid $[\text{FIRST}, (j, v_j)]$ (L 322), note that an adversary cannot force a message to be sent by an honest party to be invalid. Consider an honest party P_i, this condition guarantees that P_i eventually creates a valid set S_i and sends it to every party (L 325–327). Iterating over the same argument, every honest party eventually receives at least $2f + 1$ valid messages $[\text{SECOND}, (j, S_j)]$ (L 328–330),produces valid sets $\mathcal{T}$ that are sent to everyone (L 331–333). Analogously, every honest party eventually receives at least $2f + 1$ valid messages $[\text{THIRD}, (j, T_j)]$ (L 334–336) which allow the honest parties to output a valid set U_i (L 337–339).

Integrity: Consider an honest value (P_j, v_j) contained in a valid output set. By assumption (P_j, v_j) contains information to verify that the

value (P_j, v_j) was indeed created by party P_j. Since P_j is honest, v_j is its sole *core-broadcast*(P_j, v_j) value.

Weak agreement: Consider honest parties P_i and P_j such that P_i has *core-delivered*(P_j, U_j). Since P_i has *core-delivered*(P_j, U_j), P_i has received a message $[\text{OUTPUT}, (P_j, U_j)]$ (L 340–343). Which implies that P_j has sent the message $[\text{OUTPUT}, (P_j, U_j)]$ to every party and has executed lines L 337–339.

Furthermore, the integrity property guarantees that U_j is valid. Thus, every honest party eventually receives the message $[\text{OUTPUT}, (P_j, U_j)]$ and outputs *core-delivered*(P_j, U_j) (L 340–343).

Common core: We claim that there exists an honest S_i such that S_i is not a subset of T_j for at most f honest parties. Let $\mathcal{S}$ and $\mathcal{T}$ denote the set of honest S_i and T_i respectively and $\mathcal{Q} = \{(S_i, T_j) : i, j \in \mathcal{N} \setminus \mathcal{F} \wedge S_i \not\subseteq T_j\}$ denote the set of pairs of parties such that there is an honest S_i produce by party i that is not contained in a set T_j created by honest party j. By construction of the T_j sets, the number of honest sets S_i that are not a subset of T_j is at most f, i.e., $|\{S_i : i \in \mathcal{N} \setminus \mathcal{F} \wedge S_i \not\subseteq T_j\}| \leq f$ and $|\mathcal{S}| \geq |\mathcal{T}|$. Thus, the cardinality of $\mathcal{Q}$ is bounded by $|\mathcal{Q}| \leq f|\mathcal{T}| \leq f|\mathcal{S}|$. Thus, there must be at least one honest set S_i that is not a subset of at most f honest T_i.

We also claim that there for any a set of at least $f + 1$ honest sets T_j denoted by $\mathcal{T}'$ and for every valid U_k set there exists a T_j contained in U_k,i.e., $\forall$valid $U_k, \exists T_j \in \mathcal{T}' : T_j \subseteq U_k$. Note that each U_k is a superset of at least $f+1$ honest T_j and the cardinally of $\mathcal{T}$ is upper-bounded by $2f + 1$, thus the claim is satisfied.

Assume that some party outputs a valid U_k set, by construction there exists at least $f + 1$ honest T_j and S_j sets. Define $\mathcal{T}'$ to be a set of honest T_i of size exactly $f + 1$. The first claim guarantees that there exists an S_i set that is not contained in at most f honest T_j sets, i.e., S_i is a subset of $\mathcal{T}'$. Whereas, the second claim guarantees that for any set $\mathcal{T}'$ of at least $f + 1$ honest T_j, every U_k is a superset of at least one element of $\mathcal{T}'$. We conclude that every valid U_k is a superset of S_i, thus it contains at values from at least $2f + 1$ values from different parties (by construction of S_i).

Self inclusion: Note that given an honest party P_i, the following holds

$S_i \subseteq T_i \subseteq U_i$ by construction since P_i must create an S_i set before creating a T_i set an the latter before U_i. Hence, the self inclusion property is satisfied.

□

6.5 Atomic broadcast protocol

We provided a top-down description of our algorithm *All set*. First we start with a high-level description, then we define key concepts for Algorithm 20, and we conclude with a detailed definition accompanied with a pseudocode.

6.5.1 Overview

Our protocol consists of several instances of the weak common core (Section 6.4) that are linked with the help of the common coin (Definition 6) protocol to implement atomic broadcast (Section 6.3).

Each honest party P_i selects a block to *ab-broadcast* together with references to $2f + 1$ outputs from different parties of the previous instance of the weak common core protocol and *core-broadcasts* them. When an P_i *core-delivers*(P_j, U_j), it stores the set and waits until it has *core-delivered*(P_j, U_j) from $2f + 1$ different parties. When enough sets have been *core-delivered*, P_i releases the common coin to designate a party as a anchor for this instance. For the ease of notation, we assume that the common coin runs in the background and we do not specify its events in the description. The *core-broadcast* block created by the anchor and its ancestors are *ab-delivered*, given that the block satisfies a set of conditions. A deterministic topological sort then applied to guarantee that every party *ab-delivers* the block in the same order. Each party then creates a block to *ab-broadcast* together with references to at least $2f + 1$ output from the new instance of the weak common-core and repeats the process. Parties can also include extra references older *core-delivered* sets aiming to *ab-deliver* blocks that otherwise would be left behind.

6.5.2 Definitions

Algorithm 20 uses the concepts of *blocks*, *parents*, and *ancestors*, concepts formalized below.

Definition 35. A *block* b is tuple $[P_i, transactions, ref, ref']$ where P_i denotes the creator of block b, *transactions* denotes a set of transaction, and ref, ref' are two sets of references to sets of blocks.

The sets of references ref and ref' are used to include sets from the past and previous instances of the common core respectively.

Definition 36 (Block). Given a block $b = [P_i, transactions, ref, ref']$, we define the *parents* of b, denoted as $parents(b)$, to be the set of blocks included in the sets referenced by ref. The set of blocks included in the sets referenced by ref' constitutes the set of *relatives* of b, denoted by $relatives(b)$.

In simple words, the parents of a given block of wave w are the blocks from wave $w-1$ that are included in the sets referenced by b. The relatives of b are the blocks from earlier waves than $w-1$ included in sets referenced by b. The combination of both is a recursive manner constitutes the set of ancestors of b, formalized below.

Definition 37 (Ancestors). Given $b = [P_i, transactions, ref, ref']$ a block, the set of *ancestors* of b, denoted by $ancestors(b)$, is the set of blocks formed by b itself and the ancestors of the parents and relatives of b, i.e.,

$$ancestors(b) = \{b\} \bigcup_{b' \in parents(b)} ancestors(b') \bigcup_{b' \in relatives(b)} ancestors(b').$$

Definition 38 (Strong and weak ancestors). Given a block $b = [P_i, transactions, ref, ref']$, the set of *strong ancestors* of b, denoted by $strong(b)$, is the set of blocks formed by b itself and the strong ancestors of the parents of b, i.e.,

$$strong(b) = \{b\} \bigcup_{b' \in parents(b)} strong(b').$$

The set of *weak ancestors* of b, denoted by $weak(b)$, is defined to be the set of ancestors of b that are not strong ancestors, i.e., $weak(b) = ancestors(b) \setminus strong(b)$.

6.5.3 Detailed explanation

Our protocol is structured in waves built around instances of the weak common core (Section 6.4). The weak common core is augmented with an identifier w to indicate the instance of the corresponding message. Honest parties are allowed to increment their wave number only after outputting at least $2f + 1$ *core-deliver*(w, P_j, U_j) from different parties P_j. We assume a common coin protocol (Definition 6) running in the background, such that parties can obtain the value of a coin whenever needed. We describe the protocol from the perspective of an honest party P_i, all the honest parties play a symmetrical role. We assume that every message sent contains all its ancestors, in an implemention, a hash to the ancestors is sufficient with the aid of a poll mechanism for missing blocks.

Initialization. Party P_i starts the protocol by initializing the wave and last delivered block variables and *core-broadcasting* block $b = [P_i, \mathit{transactions}(), \emptyset, \emptyset]$ (L 352–356). Block b contains an identifier of party P_i, a set of transactions but no references to *core-delivered* sets from the previous wave, nor previous waves in contrast to later blocks.

Wave. Once the protocol has been initialized, the structure of any wave is equivalent. When triggering an event *core-deliver*(w_j, P_j, U_j), party P_i stores the set U_j in the variable $\mathcal{U}[w_j, j]$ (L 359–363), the value w_j denotes the wave, or instance of the weak common core, in which U_j is *core-delivered*. P_i also updates the sets $\mathcal{B}$ and $\mathcal{U}$ with every block, and respectively set, contained in U_j (L 361), this means that $\mathcal{B}$ ($\mathcal{U}$) may contain several blocks (sets) corresponding to the same party and wave. Note that, wave w_j does not need to equal the local value w of party P_i, honest parties accept events *core-deliver*(w_j, P_j, U_j) of present, past, and future waves.

Party P_i waits until it *core-delivers*(w, P_j, U_j) from at least $2f+1$ parties including its own set (L 364) to conclude the wave. Once the wave can be finalized (L 364–370), P_i selects the anchor of the wave using the common coin (L 365) and delivers the corresponding set of blocks (L 366). The delivery of blocks is non-trivial and it is explained in the next paragraph. After the delivery routine, P_i increments the wave

number (L 367) and creates a new block for wave $w+1$ with references to every *core-delivered* set $\mathcal{U}[w,*]$ from wave w and every *core-delivered* set from earlier waves $\mathcal{U}[<w,*]$ that is not an ancestor of any $\mathcal{U}[w,*]$ (L 368–369). In the case that for some j there is more than one *core-delivered* set in $\mathcal{U}[r,*]$, party P_i selects only the first element of the list, the element that P_i received first. Party P_i concludes the wave by *core-broadcasting* the new block b (L 370).

Delivery. The delivery routine (L 373–390) takes as input a wave number w and produces as output a list of blocks to be *ab-delivered*. If P_j is an anchor for wave w, the outcome of the coin for wave w, then P_i first searches for any block b from party P_j in $\mathcal{B}[w]$, i.e., created in wave w. Furthermore, P_i verifies that b is *voted* for by at least $2f+1$ parties. A party P_k *votes* for a block b of wave w (L 377) if P_k includes block b in its *core-deliver*(w, P_k, U_k) and it does not include any other block from the same creator and wave. This voting mechanism guarantees two important properties: if any party P_k includes an honest blocks b in its *core-deliver*(w, P_k, U_k), it votes for b since honest parties only *core-broadcast* a block per wave. If there is a block b created by the anchor that is voted by at least $2f+1$ parties, party P_i constructs the set *strong*(b) consisting of the strong ancestors of b and and applies the ordering function τ to it (L 375).

The ordering function τ is defined as follows (L 379–391). Given a set S, party P_i searches for the anchor block in the set *strong*(b) of the highest wave wave w (b excluded), i.e., blocks of the shape $b_2 = [w_2, \mathcal{C}[w_2], *, *, *]$, of wave greater than the last delivered wave w_ℓ. If such block exists, invokes $\tau(b_2)$ and repeats the procedure described above. When a block $b_{m+1} = [w_{m+1}, \mathcal{C}[w_{m+1}], *, *, *]$ cannot be found, P_i *ab-delivers* the ancestors (weak ancestors included) of b_m that have not been already *ab-delivered* in some deterministic order. A block is *ab-delivered* only if no block from the same wave and creator has not previously been *ab-delivered* (L 387). P_i, then, delivers the ancestors of b_{n-1} that have not been already *ab-delivered* and repeats until b. Finally party P_i updates the last delivered wave w_ℓ (L 376. For the sake of notation, we consider $\tau_0(b)$ to be the list of *ab-delivered* by the invocation of τ with b as input with $w_\ell = 0$.

Algorithm 20 All set (party P_i)

State

346: $w \in \mathbb{N}$ // current wave
347: $\mathcal{B}[w]$ // hashmap of the set of blocks
348: $\mathcal{U}[w, i]$ //hashmap of output set of each common core instance
349: $\mathcal{C}[w]$ //hashmap of values of coins
350: $\mathcal{V}$ //set of already included blocks
351: $\mathcal{D}$ //set of *ab-delivered* blocks

Initialization

352: **upon** *initialization*() **do**
353: $w \leftarrow 1$
354: $b_\ell \leftarrow \emptyset$
355: $b \leftarrow [P_i, transactions(), \emptyset, \emptyset]$
356: *ab-broadcast*(b)

Algorithm

357: **upon** *ab-broadcast*(b) **do**
358: *core-broadcast*(w_i, P_i, b_i)

359: **upon** *core-deliver*(w_j, P_j, U_j) **do**
360: $\mathcal{U}[w_j, j] \leftarrow U_j$
361: *update*(U_j) //update $\mathcal{B}$ and $\mathcal{U}$ according to U_j
362: **for** $b_j \in U_j$ **do**
363: $\mathcal{B}[w] \leftarrow \mathcal{B}[w] \cup \{b_j\}$

364: **upon** $|\mathcal{U}[w, *]| \geq 2f + 1 \wedge \mathcal{U}[w, i] \neq \perp$ **do**
365: $\mathcal{C}[w] \leftarrow Coin(w)$
366: *deliver*(w)
367: $w \leftarrow w + 1$
368: $\mathcal{V} \leftarrow ancestors(\mathcal{U}[w-1, *])$
369: $b \leftarrow [i, transactions(), \mathcal{U}[w-1, *], \mathcal{U}[< w-1, *] \setminus \mathcal{V}]$
370: *ab-broadcast*(b)

Algorithm 21 Functions (party P_i)

371:**function** *transactions*()**:**
372: **return** set of transactions

373:**function** *deliver*(w)**:**
374: **if** $\exists B \in \mathcal{B}[w]$ such that $creator(b) = \mathcal{C}[w] \wedge$
$|\{ U \in \mathcal{U}[w, *] : vote(b, U)\}| \geq 2f + 1$ **then**
375: $D \leftarrow \tau(b)$
376: $w_l \leftarrow wave(b)$

377:**function** *vote*(b, U)**:**
378: **return** $b \in U \wedge \nexists b' \in U :$
$b' \neq B \wedge wave(b') = wave(b) \wedge creator(b') = creator(b)$

379:**function** $\tau(b)$ **:**
380: $S \leftarrow strong(b)$
381: $w \leftarrow wave(b)$
382: $\mathcal{A} \leftarrow \{b' = [\mathcal{C}[w'], w', *, *] \in S : w_\ell < w' < w\}$
383: **if** $\mathcal{A} \neq \emptyset$ **then**
384: $b^* \leftarrow \max(\mathcal{A})$ //block from the highest wave
385: $\tau(b^*)$
386: **for** $b' \in ancestors(b) \setminus \mathcal{D}$ in some deterministic order **do**
387: **if** $\nexists b' \in \mathcal{D} :$
$wave(b') = wave(b) \wedge creator(b') = creator(b)$ **then**
388: *ab-deliver*(b')
389: $\mathcal{D} \leftarrow \mathcal{D}||b'$
391: **return** $\mathcal{D}$

6.6 Analysis

We prove a series of lemmas that lead to the conclusion that our protocol implements atomic broadcast.

Lemma 30. *Given wave w and an honest party P_i, the probability that the commitment rule (L 374) is satisfied in the view of P_i by a core-broadcast(w, P_j, b) block b is at least $1/3$. Furthermore, if a core-broadcast(w, P_j, b) block b and a core-broadcast(w, P_k, b') block b' satisfy the commitment rule in wave w, then $b = b'$.*

Proof. According to the common core property of the weak common core (Definition 34), every block b contained in the common core U^*, is also contained in any valid *core-delivered* set U_j. Denote the set honest blocks contained in the common core by $\mathcal{H}$, it holds that $|\mathcal{H}| \geq f+1$ since there exists blocks *core-broadcast*(w, P_j, b_j) by at least $2f+1$ parties P_j in the common core of wave w. Since honest parties only *core-broadcast* a single block per wave, every block contained in $\mathcal{H}$ satisfies L 374. Notice that the value of any honest party inside the common core satisfies L 374, since honest parties do not create multiple values. In the case of an honest block b, the voting function *vote*(b, U) degenerate into the inclusion function $b \in U$. The unpredictability property of the common coin guarantees an uniform distribution over the set of parties (Definition 6). Hence, the probability that L 374 is satisfied is at least $\frac{f+1}{n} > \frac{1}{3}$.

Assume that two distinct blocks b, b' satisfy the commitment rule (L 374). According to the matching property of the common coin, both blocks have been created by the same party. Besides, block b is included in *core-delivered*(w, P_j, U_j) sets that do not include the other block (L 377) from at least $2f+1$ parties, at least $f+1$ honest parties. The same argument applies to b'. Since there exists only $2f+1$ honest parties, both b and b' cannot simultaneously satisfy L 374 unless there is an honest party P_j that *core-delivered* two different sets U_j, U'_j sets. We conclude that at $b = b'$. □

The proof of Lemma 30 is based on the properties of the common core. However, it is still possible that a block b satisfies L 374 without been part of the common core and b does not satisfy L 374 in the view of a different honest party.

Lemma 31. *Given a core-broadcast*(w, P_j, b) *block* b *satisfying the commitment rule (L 374) in the view of at least one honest party, then* $b \in strong(b')$ *for every valid core-broadcast*(w', P_k, b') *in any wave* $w' > w$.

Proof. Consider a wave w and a *core-broadcast*(w, P_i, b) block b that satisfies the commitment rule (L 374) in the view of at least one honest party. Consider also another wave $w' > w$ and a valid *core-broadcast*(w', P_j, b') block b'. Since b satisfies the commitment rule (L 374), b is included in *core-delivered*(w, P_j, U) from at least $2f + 1$ parties P_j, from at least $f + 1$ honest parties P_j. We proceed using induction over $\Delta = w' - w$.

Assume that $\Delta = 1$, a valid *core-broadcast*(w', P_j, b') block b' contains *core-delivered*(w, P_k, U_k) sets U_k from at least $2f+1$ different parties, at least $f+1$ honest. Whereas, block b is contained in at least $f+1$ honest *core-delivered*(w, P_k, U_k) . By quorum intersection, there is at least one honest *core-delivered*(w, P_k, U_k) set U_k containing b that is included in b'. Thus, $b \in strong(b')$.

Assume now that block $b \in strong(b')$ for $\Delta = \Delta_0$ and consider a *core-broadcast*(w', P_j, b') block b' from a later wave $w' = w + \Delta_0 + 1$. From the induction hypothesis, every valid *core-broadcast*$(w+\Delta_0, P_j, b')$ block b' satisfies that $b \in strong(b')$. Thus, every block contained in every valid *core-deliver*$(w + \Delta_0, P_j, U_j)$ set U_i has b as an ancestor. We conclude by realizing that any valid *core-broadcast*$(w + \Delta_0 + 1, P_j, b')$ block b' satisfies that $b \in strong(b')$, since a valid *core-broadcast*$(w + \Delta_0 + 1, P_j, b')$ block b' contains valid *core-deliver*$(w + \Delta_0, P_j, U_j)$ sets and the ancestry property is transitive. □

Lemma 32. *Any two blocks* b, b' *satisfying the commitment rule (L 374) in the view of at least one honest party satisfy:* $b \in strong(b')$ *if* $wave(b) < wave(b')$*, or* $b' \in strong(b)$ *if* $wave(b') < wave(b)$.

Proof. If both b and b' have been *core-broadcast* in the same wave w, Lemma 30 guarantees that $b = b'$, thus $b \in strong(b')$. If b and b' have been *core-broadcast* in waves w and $w' > w$, Lemma 31 guarantees that $b \in strong(b')$. Similarly, if $w > w'$, we conclude $b' \in strong(b)$. □

The next result guarantees that eventually every honest part *core-delivers* a set corresponding to an arbitrarily large wave.

Lemma 33. *For every $w \in \mathbb{N}$, every honest party P_i that eventually core-delivers(w, P_i, U_i).*

Proof. We proceed by induction. Consider the first wave $w = 1$, each honest party *core-broadcasts*$(1, P_j, b_j)$ a block b_j as part of the initialization routine (L 352–356). The validity property of the weak common core guarantees that every honest party P_i eventually *core-delivers*$(1, P_j, U_j)$ for every honest party P_j. Thus, honest party P_i eventually *core-delivers*$(1, P_i, U_i)$.

Assume now that an honest party P_i eventually *core-delivers*$(w - 1, P_j, U_j)$ from at least $2f + 1$ different parties. Party P_i then is allowed to increment their wave to w and *core-broadcast*(w, P_i, b_i) a block b_i (L 364–370). Eventually every honest party *core-broadcasts* a block corresponding to wave r. We conclude by using the validity property of the weak common core to guarantee that every honest party P_j eventually *core-deliver*(w, P_j, U_j) a set U_j. Thus P_i eventually *core-deliver*(w, P_i, U_i).

□

Lemma 34. *Given to anchor blocks b, b' that satisfy L 374, then $\tau_0(b) \prec \tau_0(b')$ or $\tau_0(b') \prec \tau_0(b)$, i.e, one output is a prefix of the other.*

Proof. Without loss of generality assume that $w = wave(b) \leq wave(b') = w'$. In this case Lemma 32 guarantees that $b \in strong(b')$. Furthermore, $b \in strong(b^*)$ for any valid block of wave $w^* > w$. We proceed by induction over $\Delta = w' - w$.

If $\Delta = 0$ then $b = b'$ and the statement is trivial.

If $\Delta > 0$, we inductively assume that $\tau_0(b) \prec \tau_0(b'')$ for all anchor blocks b'' in the strong ancestors of b' such that $wave(b'') < w + \Delta$. The first step in the computation of $\tau_0(b')$ is selecting the anchor block b^* with the highest wave in the strong ancestors of b' (L 384). Because $b \in strong(b')$ and $w^* = wave(b^*)$ is maximal among all strong ancestors of b', we have $w^* \geq w$. Moreover b^* is an ancestor of b', and thus $w < w'$.

Consequently, $w \leq w^* < w + \Delta = w'$, the inductive hypothesis holds, and we conclude $\tau_0(b) \prec \tau_0(b^*) \prec \tau_0(b')$.

□

Theorem 35. *You are all set (Algorithm 20) implements atomic broadcast.*

Proof. We structure the proof property by property.

Validity: Assume that honest party P_i *ab-broadcasts* (L 357) a given block b_i in wave w, then P_i *core-broadcast*(w, P_i, b_i) (L 358). The validity property of the weak common core guarantees that party P_i eventually *core-delivers*(w, P_i, U_i) a set U_i (L 359). Every honest party eventually *core-delivers*(w, P_i, U_i) and adds it to its local view (L 363)

Consider now another honest party P_j, note that P_j eventually *core-broadcast*(w', P_j, b'_j) a block b_j with U_i as either parent or relative and $w' \geq w$ (L 364–370). Set U_i would be a parent if P_j *core-delivers*(w, P_i, U_i) before completing wave $r + 1$ (L 364), otherwise, U_i is a relative. The *core-broadcast*(w', P_j, b'_j) is guaranteed by Lemma 33. Regardless of the case, any *core-broadcast*(w^*, P_j, b^*_j) block with $w^* \geq w'$ that satisfies the commitment rule (L 374) implies the *ab-delivery*(b_i), since b_i is an ancestor of b_j, thus an ancestor of b'_j (L 373–390. Hence, there exists a wave w_0 such that the satisfaction of the commitment rule (L 374) by any honest block in wave $w \geq w_0$ implies *ab-deliver*(b_i) .

On the one hand, Lemma 30 guarantees that an honest satisfies the commitment rule (L 374) with probability at least 1/3 after each wave. Lemma 33 guarantees that every honest party P_i party eventually *core-delivers*(w, P_i, U_i) a set U_i for any arbitrary wave w. Thus, the commitment rule (L 374) is eventually satisfied by an honest *core-broadcast*(w', P_j, b_j) block b_j with $w' \geq w_0$ and *ab-delivered*(b_i).

Agreement: Assume that an honest party P_i *ab-delivers*(b_i) in wave w. This means that there exists a block b' in the view of P_i, possibly $b' = B$, that satisfies the commitment rule (L 374) and b is an ancestor of b' (L 373–390). Using Lemma 32, b' is an

ancestor of any future block that satisfies L 374 in the view of any honest party. Furthermore, L 374 will eventually be satisfied by some honest block (Lemma 30 and Lemma 33). Thus, every other honest party eventually *ab-delivers*(b_i).

Integrity: Follows from Lines L 373–390, since an honest party *ab-delivers* each block at most once.

Total order: The total order property is guaranteed by Lemma 34. Block are *ab-delivered* when some block satisfies the commitment rule (L 374). Consider two parties P_i and P_j and a block b such that, P_i *core-delivers*(b) because b_i satisfies the commitment rule and P_j because b_j satisfies the commitment rule. If $wave(b_i) = wave(b_j)$, then $b_i = b_j$ and b is *ab-delivered* in the same position by both parties. Without loss of generality, assume that $wave(b_i) < wave(b_j)$, than Lemma 34 guarantees that $\tau_0(b_i) \prec \tau_0(b_j)$, i.e., both P_i and P_j deliver b in the same position. We conclude that for pair of parties P_i and P_j and pair of block b and b', P_i *ab-delivers* b before b' if and only if P_j does.

□

One we have proved that All set (Algorithm 20) implements atomic broadcast we analyze its latency.

Theorem 36. *The average number of waves until an honest party commits a leader block is at most* 3 *in the most adversarial case and* 3/2 *if every party is honest.*

Proof. On the one hand, Lemma 30 guarantees that the probability that an honest party P_i commits a leader block is 1/3, thus the number of waves until a block is committed no greater than 3, regardless of the adversarial behavior. On the other hand, when every party is honest, the function $vote(b, U)$ (L 377) simply verifies if block b is contained in set U. Thus, every block contained in the common core satisfies the commitment rule (L 374). Hence, the probability that the commitment rule (L 374) is satisfied is at least 2/3, which implies that the commitment rule (L 374) is satisfied on average at most every 3/2 waves. □

From Theorem 36 follows that in the adversarial case, a block is *ab-delivered* on average after the finalization of its wave and three

completed waves, thus 14 rounds of communication. However, in the best case, only a save is required, thus 4 rounds of communication.

6.7 Formal verification

We have formally verified properties of Algorithm 20 with the assistance of TLA[1]. We have considered a network of four parties, three honest and one Byzantine party as described in the code below. We have verified the following properties for two waves.

Property 37 (Uniqueness). For any wave, at most one block from each party is included voted by at least $2f + 1$ different parties.

Property 38 (Follow). If a block b is included in the *core-delivered* set by at least $2f + 1$, b is a strong ancestor of every block from the next wave.

Note that Property 37 corresponds to the statement of Lemma 30. Whereas, Property 38 corresponds to the main step in the proof of Lemma 31, the statement of Lemma 31 cannot be formally verified as it requires an infinite execution. Furthermore, both properties can be formally verified in only two waves, by definition. Below, we show the TLA-code used for this formal verification.

MODULE *all_set*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *Randomization*, *TLC*

CONSTANTS N, f, *rmax*

VARIABLES B, U, *round*, *Leaders*, *Core*

ASSUME $NF \triangleq N \in Nat \land f \in Nat \land (N > 3 * f)$

$Proc \triangleq 1 \,..\, N$

[1] https://lamport.azurewebsites.net/tla/tla.html

$Values \triangleq 1 \,..\, f+1$

$vars \triangleq \langle B,\ U,\ round,\ Leaders,\ Core \rangle$

$Init \triangleq$

$\quad \wedge\ B = \{\langle 1,\ i,\ 1,\ \{\}\rangle : i \in Proc\} \cup \{\langle 1,\ i,\ 2,\ \{\}\rangle : i \in 1 \,..\, f\}$

$\quad \wedge\ U = \{\}$

$\quad \wedge\ Leaders = \{\}$

$\quad \wedge\ Core = \{\langle 1,\ \{1,\ 2,\ 3\}\rangle,\ \langle 2,\ \{1,\ 2,\ 4\}\rangle,\ \langle 3,\ \{1,\ 3,\ 4\}\rangle\}$

$\quad \wedge\ round = 1$

$Create\_Block \triangleq$

$\quad \wedge\ \neg(\mathrm{E}\, x \in B : x[1] = round)$

$\quad \wedge\ \mathrm{E}\, subsetb1 \in \textsc{subset}\ (U) :$

$\quad\quad \wedge\ Cardinality(\{j \in Proc :$

$\quad\quad\quad \mathrm{E}\, x \in subsetb1 : x[2] = j\}) = 2 * f + 1$

$\quad\quad \wedge\ \forall\, x \in subsetb1 : x[1] = round - 1$

$\quad\quad \wedge\ \mathrm{E}\, x \in subsetb1 : x[2] = 1$

$\quad\quad \wedge\ (\forall\, x,\ y \in subsetb1 : x[2] = y[2] \implies x = y)$

$\quad\quad \wedge\ \mathrm{E}\, subsetb2 \in \textsc{subset}\ (U) :$

$\quad\quad\quad \wedge\ Cardinality(\{j \in Proc :$

$\quad\quad\quad\quad \mathrm{E}\, x \in subsetb2 : x[2] = j\}) = 2 * f + 1$

$\quad\quad\quad \wedge\ \forall\, x \in subsetb2 : x[1] = round - 1$

$\quad\quad\quad \wedge\ \mathrm{E}\, x \in subsetb2 : x[2] = 1$

$$
\begin{array}{l}
\land\, (\forall\, x,\, y \in \mathit{subsetb2} : x[2] = y[2] \implies x = y) \\
\land\, \mathrm{E}\, \mathit{subset2} \in \textsc{subset}\ (U) : \\
\quad \land\, \mathit{Cardinality}(\{j \in \mathit{Proc} : \\
\qquad \mathrm{E}\, x \in \mathit{subset2} : x[2] = j\}) = 2 * f + 1 \\
\quad \land\, \forall\, x \in \mathit{subset2} : x[1] = \mathit{round} - 1 \\
\quad \land\, \mathrm{E}\, x \in \mathit{subset2} : x[2] = 2 \\
\quad \land\, (\forall\, x,\, y \in \mathit{subset2} : x[2] = y[2] \implies x = y) \\
\quad \land\, \mathrm{E}\, \mathit{subset3} \in \textsc{subset}\ (U) : \\
\qquad \land\, \mathit{Cardinality}(\{j \in \mathit{Proc} : \\
\qquad\quad \mathrm{E}\, x \in \mathit{subset3} : x[2] = j\}) = 2 * f + 1 \\
\qquad \land\, \forall\, x \in \mathit{subset3} : x[1] = \mathit{round} - 1 \\
\qquad \land\, \mathrm{E}\, x \in \mathit{subset3} : x[2] = 3 \\
\qquad \land\, (\forall\, x,\, y \in \mathit{subset3} : x[2] = y[2] \implies x = y) \\
\qquad \land\, \mathrm{E}\, \mathit{subset4} \in \textsc{subset}\ (U) : \\
\qquad\quad \land\, \mathit{Cardinality}(\{j \in \mathit{Proc} : \\
\qquad\qquad \mathrm{E}\, x \in \mathit{subset4} : x[2] = j\}) = 2 * f + 1 \\
\qquad\quad \land\, \forall\, x \in \mathit{subset4} : x[1] = \mathit{round} - 1 \\
\qquad\quad \land\, \mathrm{E}\, x \in \mathit{subset4} : x[2] = 4 \\
\qquad\quad \land\, (\forall\, x,\, y \in \mathit{subset4} : x[2] = y[2] \implies x = y) \\
\qquad\quad \land\, B' = B \cup \{\langle \mathit{round},\, 1,\, \{x[3] : x \in \mathit{subsetb1}\}\rangle, \\
\qquad\quad \langle \mathit{round},\, 1,\, \{x[3]\ \ : x \in \mathit{subsetb2}\}\rangle, \\
\qquad\quad \langle \mathit{round},\, 2,\, \{x[3] : x \in \mathit{subset2}\}\rangle, \\
\qquad\quad \langle \mathit{round},\, 3,\, \{x[3] : x \in \mathit{subset3}\}\rangle,
\end{array}
$$

$$\langle round,\ 4,\ \{x[3] : x \in subset4\}\rangle\}$$

$\wedge$ UNCHANGED $\langle U,\ Core,\ Leaders,\ round\rangle$

$Create\_Set \triangleq$

$\wedge \neg(\mathrm{E}\, x \in U : x[1] = round)$

$\wedge \mathrm{E}\, core\_set \in Core :$

$\wedge\ core\_set[1] = round$

$\wedge \mathrm{E}\, subset \in \mathrm{SUBSET}\ (B) :$

$\wedge \forall\, j \in core\_set[2] : \mathrm{E}\, x \in subset :$

$\wedge\ x[2] = j$

$\wedge\ x[1] = round$

$\wedge \forall\, k \in Proc : Cardinality(\{y \in subset : y[2] = k\}) = 1$

$\wedge \mathrm{E}\, xb1,\ xb2,\ x2,\ x3,\ x4 \in B :$

$\wedge\ xb1[1] = round$

$\wedge\ xb2[1] = round$

$\wedge\ x2[1] = round$

$\wedge\ x3[1] = round$

$\wedge\ x4[1] = round$

$\wedge\ U' = U \cup \{\langle round,\ 1,\ subset \cup \{xb1\}\rangle,$

$\langle round,\ 1,\ subset \cup \{xb2\}\rangle,\ \langle round,\ 2,\ subset \cup \{x2\}\rangle,$

$\langle round,\ 3,\ subset \cup \{x3\}\rangle,\ \langle round,\ 4,\ subset \cup \{x4\}\rangle\}$

$\land$ UNCHANGED $\langle B,\ Leaders,\ Core,\ round\rangle$

$Deliver \triangleq$

$\land \forall\, j \in Proc : \mathrm{E}\, u \in U : u[1] = round \land u[2] = j$

$\land \mathrm{E}\, b \in B :$

$\land\ Cardinality(\{j \in Proc : \mathrm{E}\, u \in U :$

$u[1] = b[1] \land u[2] = j \land b \in u[3] \land$

$(\forall\, x,\ y \in u[3] : (x[2] = y[2]) \implies x = y)\}) \geq 2 * f + 1$

$\land\ Leaders' = Leaders \cup \{b\}$

$\land$ UNCHANGED $\langle B,\ U,\ round,\ Core\rangle$

$Increase \triangleq$

$\land\ round < rmax$

$\land \forall\, j \in Proc : \mathrm{E}\, u \in U : u[1] = round \land u[2] = j$

$\land\ round' = round + 1$

$\land$ UNCHANGED $\langle B,\ U,\ Leaders,\ Core\rangle$

$Stop \triangleq$

$\land$ UNCHANGED $vars$

$Next \triangleq$

$\lor\ Create\_Block$

$\lor\ Create\_Set$

$\lor\ \mathit{Deliver}$

$\lor\ \mathit{Increase}$

$\lor\ \mathit{Stop}$

$\mathit{Unique} \triangleq$

$\forall\, i \in \mathit{Proc} : \forall\, r \in 1\,..\,\mathit{rmax} :$

$\mathit{Cardinality}(\{\mathit{leader} \in \mathit{Leaders} : \mathit{leader}[1] = r \land \mathit{leader}[2] = i\}) \leq 1$

$\mathit{Follow} \triangleq$

$\forall\, \mathit{leader} \in \mathit{Leaders} : \mathit{Cardinality}(\{b \in B : b[1] = \mathit{leader}[1] + 1 \land$

$(\forall\, u \in b[3] : \neg(\mathit{leader} \in u))\}) < 1$

6.8 Conclusion

In this chapter, we have introduced a novel protocol that relies on instances of a weaker variant of the common core, coupled with a common coin. Our contribution extends beyond demonstrating that this protocol successfully implements atomic broadcast; we have also undertaken formal verification of its properties using TLA. Moreover, the delivery of every broadcast block has been ensured. In line with the findings presented in Chapter 5, it is noteworthy that All set (Algorithm 20) has the potential to be an optimal protocol.

Chapter 7

Sandwich-attack prevention

> If you are a true hero, you shouldn't steal.
>
> The Luminary

7.1 Introduction

The field of blockchain protocols has proved to be extremely robust. Since its creation with Bitcoin [88], it had gone through several enhancements such as Ethereum [2] and has seen the appearance of *decentralized finance* (DeFi). With this, some design flaws started to show up. Blockchains would ideally allow users to trade tokens with each other in a secure manner. However, existing designs do not consider users trading tokens of one platform for FIAT currency or tokens of a different platform, arguably one of the major flaws of today's blockchain platforms, *maximal extractable value* (MEV) [44]. Current estimates show that the total volume of MEV since 2020 is around 675M USD [3].

From a social welfare perspective, while MEV is profitable to miners, it presents a serious invisible tax on the users on the blockchain. Indeed the financial losses built up over time could potentially shy away users from the blockchain, and consequently impact the security of the chain.

Sandwich attacks are one of the most common types of MEV [37] accounting for a loss of 174M USD over the span of 33 months [96] for users of Ethereum. Sandwich attacks leverage the miner's ability to *select* and *position* transactions within a block. Consider the simple example of a sequence of transactions that swap one asset X for another asset Y in a decentralized exchange where exchange rates are computed automatically based on some function of the number of underlying assets in the pool (e.g., a constant product market maker [11]). Now suppose there is a miner that also wants to swap some units of X for Y. The most favorable position for the miner would be to place their transaction at the start of the sequence, so as to benefit from a lower X-to- Y exchange rate. This approach achieves a simple arbitrage strategy for any sequence of X-to-Y swaps: the miner can insert an X-to-Y exchange at the start of the sequence and use the computed exchange rate to sell, say, k units of X to get units of Y. The miner then front-runs the sequence of X-to-Y swaps, i.e., it inserts its own transaction at the start. To finish off the attack, the miner back-runs the sequence with another transaction of its own that swaps some units of Y to X, i.e., inserts this transaction at the end, and will often obtain more than k units of X. In this way, the miner profits from its insider knowledge and its power to order transactions. Refer to Section 7.7 for a detailed description of exchange rate computation and sandwich attacks.

Since any miner of a given block has full control over the transactions added to the block in the majority of the protocols, as well as over the way transactions are ordered, it is often straightforward for the miner to launch the above attack. Consequently, this gives miners a lot of power as they control precisely the selection and positioning of transactions with every block they mine. This problem has received broad attention in the practice of DeFi and in the scientific literature.

A classic technique to mitigate this attack is thus to remove the control over the positioning of the transactions in the block from the adversary, whether by using a trusted third party to bundle and order the transac-

tions as in flashbots[1], Eden[2], or OpenMEV[3]. Another method works by imposing a fair ordering of the transactions using a consensus algorithm that respects the order in which miners and validators first received the transactions [67]. The classic solutions either affect the centralization of the protocol or its efficiency. Furthermore, they cannot easily be implemented on top of existing blockchain protocols.

In this work, we introduce the *Partitioned and Permuted Protocol*, abbreviated Π^3, an efficient decentralized algorithm that does not rely on external resources to counter front-running. It renders sandwich attacks unprofitable and can easily be implemented on top of an existing blockchain protocol Π.

Protocol Π^3 determines the final order of transactions in a block B_i, created by a miner M_i, through a *uniformly randomly chosen permutation* Σ_i. To explain the method, let us focus on three transactions in B_i, a victim transaction tx^* submitted by a client, and the front-running and back-running transactions, tx_1 and tx_2, respectively, created by the miner. Since any relative ordering of these three transactions is equally probable, tx_1 will be ordered before tx_2 with the same probability as tx_2 before tx_1, hence the miner will profit or make a loss with the same probability. Protocol Π^3 uses a fresh permutation for each block; it is chosen by a set of *leaders*, which are recent miners in the blockchain. We recognize and overcome the following challenges.

First, Σ_i must not be known before creating B_i, otherwise M_i would have the option to use Σ_i^{-1}, the inverse of Σ_i, to initially order the transactions in B_i, so that the final order is the one that benefits M_i. We overcome this by making Σ_i known only *after* M_i *has been mined*. On the other hand, if Σ_i is chosen after creating B_i, a coalition of leaders would be able to try multiple different permutations and choose the most profitable one — the number of permutations a party can try is only limited by their processing power. For these reasons, we have the leaders *commit* to their contributions to Σ_i before B_i becomes known, producing unbiased randomness. To incentivize leaders to open their commitments, our protocol Π^3 employs a delayed reward release mechanism that only releases the payment to leaders when they have generated and opened all commitments.

[1] https://www.flashbots.net
[2] https://www.edennetwork.io
[3] https://openmev.xyz/

In some cases, however, performing a sandwich attack might still be more profitable than the block reward, and hence a leader might still choose to not reveal their commitment to bias the resulting permutation. In general, a coalition of k leaders can choose among 2^k permutations out of the $n_t!$ possible ones, where n_t denotes the number of transactions in the block. It turns out that the probability that tx_1, tx^*, and tx_2 appear in that order in one of the 2^k permutations can be significant for realistic values. Protocol Π^3 mitigates this by *dividing each transaction* into m chunks, which lowers the probability of a profitable permutation in two ways. First, the number of possible permutations is much larger, $(n_t m)!$ instead of $n_t!$. Second, a permutation is now profitable if the majority of chunks of tx_1 appear before the chunks of tx^*, and vice versa for the chunks of tx_2. As we discuss, the probability of a profitable permutation approaches zero rapidly as the number of chunks m increases. We discuss how to implement the chunking mechanism while preserving transaction integrity and atomicity.

Organization. In this chapter, we introduce a construction that takes as input a blockchain protocol Π and produces a new blockchain protocol Π^3 in which sandwich attacks are no longer profitable. We begin by revisiting the concept of *atomic broadcast* [27] and setting the model for the analysis. Secondly, we introduce our construction justifying how miners are incentivized to follow the protocol before moving on to analyzing the construction in detail. Thirdly, we guarantee that the construction does not include any vulnerability to the protocol by showing that Π^3 implements a variant of atomic broadcast if Π does. This part of the analysis is performed in the traditional *Byzantine* model. Fourthly, we consider the *rational* model to show that sandwich attacks are no longer profitable in Π^3. We consider the dual model of *Byzantine* for the security analysis and *rational* for the analysis of the sandwich attacks because we considered it to be a perfect fit to show that the security of Π^3 is not weakened even against an adversary that obtains nothing for breaking the protocol, as well as, we can assume that any party attempts to extract value from any sandwich attack. In other words, we consider both the security analysis and the analysis of the sandwich attack in the worst scenario possible for the protocol. Lastly, we conclude the chapter with an empirical analysis of the protocol under real-life data, as well as an analysis of the additional overhead introduced by our protocol.

Acknowledgement. The material contained in this chapter corresponds to the work 'Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks' [6] to be published at Opodis23.

7.2 Related work

The idea to randomize the transaction order within a block is folklore in the blockchain space. It has been explored by Yanai [109] and also implemented in the wild [4]. To the best of our knowledge, we are the first to implement the randomization using on-chain randomness and to provide a security analysis for this model. Additionally, Randomspam [4] also acknowledges that some spamming attacks can occur with randomized transactions, where the attacker aims to insert several low-cost transactions to maximize the probability that some of these transactions are positioned exactly at a profitable transaction. Our work reduces the success probability of these attacks by first chunking each transaction into smaller parts and then permuting all chunked transactions, rendering exact positioning attacks less profitable without adding more transactions and incurring larger gas costs.

A recent line of work [67, 75, 65, 29, 66] formalizes the notion of *fair ordering* of transactions. These protocols ensure, at consensus level, that the final order is consistent with the local order in which transactions are observed by parties. Similarly, the Hashgraph [17] consensus algorithm aims to achieve fairness by having each party locally build a graph with the received transactions. As observed by Kelkar *et al.* [67], a transaction order consistent with the order observed locally for any pair of transactions is not always possible, as Condorcet cycles may be formed. As a result, fair-ordering protocols output a transaction order that is consistent with the view of only some *fraction* of the parties, while some transactions may be output in a batch, i.e., with no order defined among them. Moreover, although order-fairness removes the miner's control over the order of transactions, it does not eliminate front-running and MEV-attacks: a *rushing* adversary that becomes aware of some tx early enough can broadcast its own tx' and make sure that sufficiently many parties receive tx' before tx.

Another common defense against front-running attacks is the *commit*

and reveal technique. The idea is to have a user first commit to a transaction, e.g., by announcing its hash or its encryption, and, once the order is fixed, reveal the actual transaction. However, an adversary can choose not to reveal the transaction, should the final order be non-optimal. Doweck and Eyal [48] employ time-lock puzzle commitments [98], so that a transaction can be brute-force revealed, and protocols such as Unicorn [78] and Bicorn [38] employ verifiable delay functions [23] to mitigate front-running. Whereas they indeed manage to mitigate front running, the main disadvantages of these solutions are threefold: firstly, transactions may be executed much later than submitted, with no concrete upper bound on the revelation time. Secondly, a delay for the time-lock puzzle has to be chosen which matches the network delay and adversary's computational power. Finally, it is unclear who should spend the computational power to solve the time-lock puzzles, especially in proof of work blockchains where this shifts computational power away from mining.

A different line of work [49, 86, 28, 97, 112] hides the transactions until they are ordered with the help of a *committee*. For instance, transactions may be encrypted with the public key of the committee, so that its members can collaboratively decrypt it. However, this method uses threshold encryption [47] and requires a coordinated setup. Also multiparty computation (MPC) has been used [19, 5, 82] to prevent front-running. MPC protocols used in this setting must be tailor-made so that misbehaving is identified and punished [18, 71]. A disadvantage of the aforementioned techniques is that the validity of a transaction can only be checked after it is revealed. These techniques also rely on strong cryptographic assumptions and coordination within the committee. The protocol presented in this work disincentivizes sandwich attacks without requiring hidden transactions or employing computationally heavy cryptography.

Another widely deployed solution against front-running involves a dedicated *trusted third party*. Flashbots[4], Eden[5], and OpenMEV[6] allow Ethereum users to submit transactions to their services, then order received transactions, and forward them to Ethereum miners. Chainlink's Fair Sequencing Service [34], in a similar fashion, aims to collect en-

[4] www.flashbots.net
[5] www.edennetwork.io
[6] https://openmev.xyz/

crypted transactions from users, totally orders them, and then decrypts them. The third-party service may again be run in a distributed way. The drawback with these solutions is that attacks are not eliminated, but trust is delegated to a different set of parties.

An orthogonal but complementary line of research is taken by Heimbach and Wattenhofer [61]. Instead of eliminating sandwich attacks, they aim to improve the resilience of ordinary transactions against sandwich attacks by strategically setting their slippage tolerance to reduce the risk of both transaction failure as well as sandwich attacks.

Last but not least, Baum *et al.* [**?**] and Heimbach and Wattenhofer [61] survey the area of front-running attacks.

7.3 Model

Notation. For a set X, we denote the set of probability distributions on X by $\mu(X)$. For a probability distribution $\nu \in \mu(X)$, we denote sampling x from X according to ν by $x \leftarrow \nu$.

7.3.1 Abstractions

Parties broadcast transactions and deliver blocks using the events *bab-broadcast*(tx) and *bab-deliver*(b), respectively, where block b contains a sequence of transactions $[tx_1, \ldots, tx_m]$. The protocol outputs an additional event *bab-mined*(b, P), which signals that block b has been *mined* by party P_i, where P_i is defined as the *miner* of b. Notice that *bab-mined*(b, P) signals only the creation of a block and not its delivery. In addition to predicate $VT()$, we also equip our protocol with a predicate $VB()$ to determine the validity of a block. Moreover, we define a function $FB()$, which describes how to fill a block: it gets as input a sequence of transactions and any other data required by the protocol and outputs a block. These predicates and function are determined by the higher-level application or protocol.

In this chapter we model protocols *block-based atomic broadcast* (Definition 4) concept recalled below.

Definition 4 (Block-based atomic broadcast). A protocol implements *block-based atomic broadcast* with validity predicates $VT()$ and $VB()$ and block-creation function $FB()$ if it satisfies the following properties, except with negligible probability:

Validity: If a correct party invokes a *bab-broadcast*(tx), then every correct party eventually outputs *bab-deliver*(b), for some block b that contains tx.

No duplication: No correct party outputs *bab-deliver*(b) for a block b more than once.

Integrity: If a correct party outputs *bab-deliver*(b), then it has previously output the event *bab-mined*($b, \cdot$) exactly once.

Agreement: If some correct party outputs *bab-deliver*(b), then eventually every correct party outputs *bab-deliver*(b).

Total order: Let b and b' be blocks, and P_i and P_j correct parties that output *bab-deliver*(b) and *bab-deliver*(b'). If P_i delivers b before b', then P_j also delivers b before b'.

External validity: If a correct party outputs *bab-deliver*(b), such that $b = [tx_1, \ldots, tx_m]$, then $VB(b) = \text{TRUE}$ and $VT(tx_i) = \text{TRUE}$, for $i \in 1, \ldots, m$. Moreover, if $FB(tx_1, \ldots, tx_m)$ returns b, then $VB(b) = \text{TRUE}$.

Furthermore, we enhance our block-based atomic broadcast primitive with a *fairness* property recall below.

Definition 5 (Fairness). A *block-based atomic broadcast* protocol is *fair* if it satisfies the following property, except with negligible probability:

Fairness: There exists $C \in \mathbb{N}$ and $\mu \in \mathbb{R}_{>0}$, such that for all $N \geq C$ consecutive delivered blocks, the fraction of the blocks whose miner is correct is at least μ.

Observe that the properties assure that *bab-mined*(b, P) is triggered exactly once for each block b, hence each block has a unique miner. For ease of notation, we define on a block b the fields $b.txs$, which contains its transactions, and $b.miner$, which contains its miner. Since blocks are delivered in total order, we can assign them a *height*, a sequence number

in their order of delivery, accessible by *b.height*. Finally, for simplicity we assume that a delivered block allows access to all blocks with smaller height, through an array *b.chain*. That is, if $b.height = i$ then $b.chain[i']$ returns b', such that $b'.height = i'$, for all $i' \leq i$.

7.3.2 Blockchain and network

Blockchain protocols derive their security from different techniques such as *proof of work* (PoW) [88], *proof of stake* (PoS) [46], *proof of space-time* ($PoST$) [40], or *proof of elapsed time* ($PoET$) [24]. In the remainder of the work, we consider a generic protocol Π that has a probabilistic termination condition, capturing all the model above. Furthermore, we model Π as block-based atomic broadcast.

Adversarial & network. We consider a dual adversarial model. For the security analysis, we consider a *Byzantine* adversary, for the analysis of *Sandwich attacks* we consider the *Rational* model (Section 2.1). Both adversaries are considered in the *synchronous-rounds* model (Section 2.2.1.)

Transactions & Blocks. A transaction *tx* contains a set of *inputs*, a set of *outputs*, and a number of digital signatures. Transactions are batched into *blocks*. A block contains a number of transactions, n_t, for simplicity we assume n_t to be constant. A block b may contain parameters specific to protocol Π such as references to previous blocks, but we abstract the logic of accessing them in a field *b.chain*, as explained in the context of Definition 4. We allow conditional execution of transactions across blocks, i.e., a transaction can be executed conditioned on the existence of another transaction in a previous block.

7.4 Protocol

Our proposed protocol Π^3 ("Partitioned and Permuted Protocol") contains two modifications to a given underlying blockchain protocol Π in order to prevent sandwich MEV attacks.

Our first modification involves randomly permuting the transactions in any given block. Note that a naive way of doing so is to use an external oracle (e.g., DRAND [1] or NIST beacon [68]) to generate the randomness which will be applied to a given block. However, using an external source of randomness relies on strong trust assumptions on the owners of the source, leaves our protocol vulnerable to a single point of failure and it introduces incentive issues between miners of the chain and owners of the external source. To avoid this, our protocol uses miners of the immediately preceding blocks to generate the randomness. These miners, which we refer to as *leaders* for the given block, are in charge of generating random *partial seeds*. These partial seeds are then combined to form a *seed* which will be the input into a PRG to produce a random permutation that is applied to the transactions in the block. To ensure that the permutation is random, we need first to achieve that leaders participate in the generation of random partial seeds and secondly to ensure the partial seeds generated by the leaders are random. That is, the leaders should not commit to the same partial seed each time or collude with other leaders to generate biased partial seeds. To incentivize each leader to participate in the generation of the seed, Π^3 stipulates that they commit to their partial seed and present a valid opening during the commitment opening period, otherwise their reward will be burned. In typical blockchain protocols, the miner of a block receives the block reward immediately. In Π^3, the miner does not receive the reward until a certain number of additional blocks has been mined. We refer to this as a *waiting phase* and stress that the precise length of the waiting phase is a parameter in our protocol that can be tweaked.

Our second modification is to divide the transfers of each transaction into smaller chunks before permuting the chunked transactions of a block. This modification increases the cardinality of the permutation group in order to reduce the effectiveness of any attack aiming to selectively open partial seeds in order to bias the final permutation.

We stress that our proposed modifications incur minimal computational overhead, since the only possible overhead corresponds to transaction delivery and this aspect is computationally cheap. Thus, the only noticeable impact of our protocol is latency. In Section 7.6 we provide an in-depth analysis of the efficiency impact of our proposed modifications.

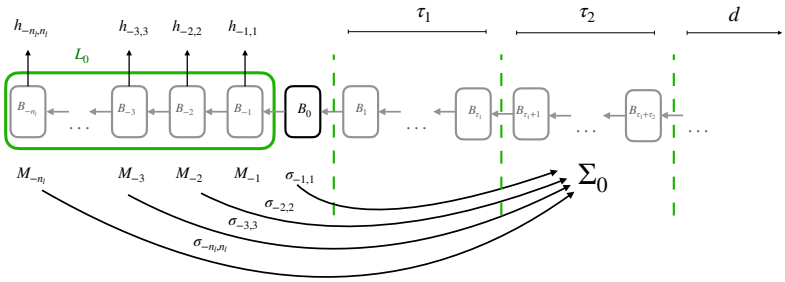


Figure 7.1. We illustrate the routine for the creation of the random permutation σ_0 created by the leader set L_0 and to be applied on block B_0 in Bitcoin. The leaders L_0 for block B_0 is formed by the miners of the n_ℓ blocks before B, marked with the green box. When party M_i mined block B_i, the party generated some random seed $\sigma_{-i,i}$ and included its commitment $h_{-i,i}$ as part of the newly mined block. After block B_0 is mined, the leaders wait for τ_1 blocks before opening the commitments. The commitments must be included in the following τ_2 blocks. Finally the parties wait until every block containing openings are confirmed before delivering block B_0

7.4.1 Permuting transactions

Protocol Π^3 consists of the following four components (see Figure 7.1): block mining, generation of the random permutation, reward (re)-distribution, and chunking the transactions.

Appending the partial seeds. Let n_ℓ be the size of the leader set for each block. The miner M_i of block B_i is part of the leader set of blocks B_{i+j}, for $j \in [n_\ell]$. M_i must therefore contribute a partial seed $\sigma_{i,j}$ for each of these n_ℓ blocks following B_i. Hence, M_i needs to create n_ℓ random seeds $\sigma_{i,1}, \ldots, \sigma_{i,n_\ell}$ and commitments to them, $C(\sigma_{i,1}), \ldots, C(\sigma_{i,n_\ell})$. The commitments $C(\sigma_{i,j})$, for $j \in [n_\ell]$, are appended to block B_i, while the seeds $\sigma_{i,j}$ are stored locally by M_i. A block that does not contain n_ℓ commitments is considered invalid.

Looking ahead, we want that any party knowing the committed value can demonstrate it to any other party. Thus, the more standard commitments schemes such as Pedersen commitment [92] are ill-suited. Instead, Π^3 uses a deterministic commitment scheme for committing to permutations, in particular, a collision-resistant cryptographic hash function. When the entropy of the committed values is high enough, then a hash function constitutes a secure commitment scheme. Since the parties commit to a random partial seed, hash functions suffice and yield a cheap commitment scheme.

Opening the commitments. Let $\tau_1, \tau_2 \in \mathbb{N}_{>0}$. Between τ_1 and $\tau_1 + \tau_2$ blocks after the creation of some block B_i, the commitments of the partial permutation to be applied on block B_i must be opened. The miners of these blocks also need to append the openings to their blocks, unless a previous block in the chain already contains them (see below for more details). The parameter τ_1 controls the probability of rewriting block B_i after the commitments have been opened. Whereas, parameter τ_2 guarantees that there is enough time for all the honest commitments to be opened and added to some block. Any opening appended a block B_j for $j > i + \tau_1 + \tau_2$ is ignored. We note that specific values of τ_1 and τ_2 might cause our protocol to suffer an increase in latency. We leave these parameters to be specified by the users of our protocol. For the interested reader, we discuss latency-security trade-offs in Section 7.6. The τ_1 blocks created until opening the commitments takes place is known as *silent phase*, whereas the following τ_2 blocks is known as *loud phase*.

A possible way to record the opening of commitments is for the miners that own the commitments to deploy a smart contract that provides a method $open(i, j, \sigma_{i,j})$, where $\sigma_{i,j}$ is a (claimed) opening of the j-th commitment $h_{i,j}$ published in the i-th block B_i. We remark, that the smart contract serves only as proof that an opening to a commitment has been provided, and does not add any functionality to the protocol, so other proof mechanisms can also be considered. The protocol monitors the blockchain for calls to this method. The arguments to each call, as well as the calling party and the block it appears on, are used to determine the final permutations of the blocks and the distribution of the rewards, which we will detail below. We stress that *not* opening a commitment does not impact the progress of protocol, as unopened

commitments are ignored.

Deriving the permutation from partial seeds. Let the seed σ_i for block B_i be defined as $\sigma_{i-1,1} \oplus \sigma_{i-2,2} \oplus \ldots \oplus \sigma_{i-n_\ell,n_\ell}$. Given the seed σ_i, let $r_i := G(\sigma_i)$, where $G : \{0,1\}^\lambda \to \{0,1\}^\ell$ is a pseudorandom generator. If at least one of the partial seeds $\sigma_{i,j}$, for $j \in [n_\ell]$, is chosen at random, then σ_i is random as well, and r_i is indistinguishable from a random number [22] without the knowledge of $\sigma_{i,j}$. There are standard algorithms to produce a random permutation from a polynomial number of bits [42].

Incentivizing the behavior. A crucial factor in the security of Π^3 against sandwich MEV attacks is that the permutation used to order transactions within a block should be truly random. Thus, the miners should generate all partial seeds uniformly at random. To incentivize them to do so, we exploit the fact that all leaders remain in the waiting phase for a period of time, which means that they have not yet received the block rewards and fees for mining their block on the blockchain. Note that the waiting phase is $n_\ell + \tau_1 + \tau_2 + d$ blocks long. This implies that their rewards can be claimed by other miners or burned if a party diverges from the proper execution, according to the rules described below. Consider a partial permutation $\sigma_{i,j}$ committed by miner M_i of block B_i. Recall that $\sigma_{i,j}$ will be applied on block B_{i+j} and that miners can be uniquely identified due to the *bab-mined*() event.

1. Before τ_1 blocks have been appended after block B_{i+j} any other leader of the leader set $\mathcal{L}_{i+j}$ who can append a pre-image of $h_{i,j}$ to the chain can receive the reward and fees corresponding to M_i. This mechanism prevents party M_i from disclosing its commitment before every other leader committed its randomness, thus preventing colluding. A miner whose commitment has been discovered by another leader is excluded from all the leader sets.
2. If the opening of $\sigma_{i,j}$ is not appended to any block, miner M_i loses its reward and fees. This mechanism prevents miners from not opening their commitments. Note that miners are incentivized to include all the valid openings, as discussed below.
3. If any of the previous conditions do not apply, party M_i receives an

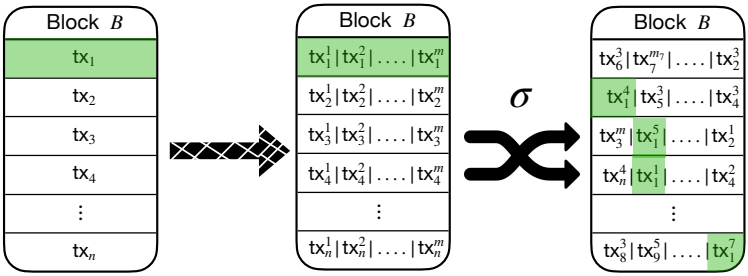


Figure 7.2. We illustrate the process followed by party M to deliver a block B. We denote by $tx_1, .., tx_n$ the n transaction that constitute B. In the first step, party M breaks each transaction tx_i into m transactions $tx_i^1, .., tx_i^m$ involving a smaller amount. These smaller transactions are later permuted according to the random permutation Σ. Lastly, party M delivers these small transactions in this new order.

> α fraction of the block reward and fees for $\alpha \in (0, 1)$, which is paid when M_i leaves the waiting phase. Each miner that appends the opening of M_i's commitments gets $\frac{(1-\alpha)\cdot w}{n_\ell}$ for each commitment.

In the remainder of this work we refer to the block reward and fees as simply *block reward*.

7.4.2 Chunking transactions

In all commit-and-open schemes, there exists the vulnerability that malicious parties decide to not open their commitments so as to bias the outcome. In our protocol, any coalition of k leaders can choose between 2^k ways to bias the final permutation. If one miner manages to create multiple blocks out of $B_{i-1}, \ldots, B_{i-n_\ell}$, this does not even require collusion with others.

The adaptive attacks mounted through withholding can be countered with a simultaneous broadcast abstraction [39], but realizing this is al-

most impossible in practice [74], especially in the blockchain domain. Alternatively, time-lock puzzles may negate the effect of the delay. But this technique costs computational effort, which may have a negative impact on the environment and possibly also on the protocol. In the particular case of generating a permutation, there is an alternative.

Chunking transfers in transactions. Let us assume that a block contains n_{tx} transactions, this means there exist $n_{tx}!$ possible permutations of them. A coalition of k leaders can choose between 2^k possible permutations among the $n_{tx}!$ total permutations. Furthermore, in the simplest case the coalition only aims to order the three transactions that constitute the sandwich attack, thus the fraction of advantageous permutations is $\frac{1}{6}$, the fraction of disadvantageous permutations is $\frac{1}{6}$ and the remaining ones are neutral. If k is big enough, the coalition could still extract enough value to compensate for the lost block rewards of those parties that do not open their commitment.

Therefore we want to increase the size of the permuted space. We assume here that every transaction consists of arbitrary code and a few specialized instructions that are *transfers* of coins or tokens. These may be the *native payment operation* of the blockchain or operations that involve a well-known *standard format for tokens*, which are emulated by smart contracts (such as the ERC-20 standard in Ethereum). We now divide every transfer generated by some transaction into m chunks.

For instance, suppose transaction tx_i consists of Alice paying Bob 1 ETH. In our protocol, each party would locally split tx_i into m chunks $tx_i^1, \ldots tx_i^m$, consisting of Alice transferring $1/m$ ETH to Bob each. After all transactions are chunked, the permutation will be applied to the larger set of transactions. There exist $(n_{tx}m)!$ permutations and the coalition would need to order the $3m$ chunks that constitute the involved transactions. Furthermore, for a given permutation with some chunks ordered beneficially, there will exist chunks ordered in a disadvantageous way, with overwhelming probability. The coalition needs to optimize the good ordering of some chunks while keeping the bad ordering under control. Obtaining a favorable ordering becomes extremely unlikely as number of chunks m grows (Section 7.5.2).

Execution of transactions and chunks. Transactions contain arbitrary code whose execution produces an ordered list of transfers, as introduced before. The process of chunking proceeds in two stages.

In the first stage, the party executes the code of all transactions contained in the block serially, in the order determined by the miner; this produces a list of transfers and the corresponding amounts for each transaction. Some transactions may turn out to be invalid, they are removed from the further processing of the block.

In the second stage, for each valid transaction tx a list of m transfer chunks $tx^1, \ldots tx^m$ is produced such that tx^1 contains the *code* executed by tx, and its transfers have their amount set to $1/m$ of the number computed by the code. The transaction chunks $tx^2, \ldots tx^m$ contain *only* the transfers, with their numbers set to $1/m$ of original amounts, but these transactions do not execute further code. If the code executed by tx^1 produces different transfers than in the first stage, the execution of tx^1 is aborted, also the execution of $tx^2, \ldots tx^m$. We consider two transfers to be the same if they transfer the same amount of coins or tokens from the same source address to the same destination address. Note the blockchain state produced by a transaction in the first stage can differ from the state produced by the same transaction in the actual execution according to the second stage execute; this is the case, for instance, when it interacts with a smart contract.

The block being permuted now contains up to m times as many chunks as the original block contained transactions, each of them transferring $1/m$ the value.

Notice that the permutation is not uniformly random across all choices, but needs to respect that tx^1 appears first within the set of chunks resulting from tx. However, this restriction in the permutation does not constitute any loss of generality since every chunk performs an identical transfer. An adversarial miner can utilize fine-grained conditions such as slippage to additionally control the conditional execution of transactions – and in our case transaction chunks – in a given block. The execution of transactions explained above guarantees atomicity: all chunks are executed or no chunk is executed. In Section 7.8 we present an in-depth analysis of how slippage could lead to higher expected revenue, which may also be of independent interest.

7.4.3 Details

In Algorithm 22 we show the pseudocode for protocol Π^3, which implements a block-based atomic broadcast (*bab*) primitive. The pseudocode assumes an underlying protocol Π, which is also modeled as a block-based atomic broadcast (*bab*) primitive, as defined in Section 7.3. The user or high-level application interacts with Π^3 by invoking Π^3-*broadcast*(*tx*) events. These are handled by invoking the corresponding Π-*broadcast*(*tx*) event on the underlying protocol Π (L 394-395).

Protocol Π outputs an event *bab-mined*(b, Q) whenever some party P_j mines a new block b (L 396). For Π^3, the mining of a new block at height i starts the opening phase for the block at height $i_{\text{open}} = i - \tau_1 - 1$ (L 397). Hence, party P_i loops through the n_ℓ blocks before i_{open} and checks whether it is the miner of each of them (L 398-400). If this is the case, P_i must provide a valid opening to the commitment related to block at height i_{open}. The opening is achieved by a specific type of transaction, for example through a call to a smart contract. In the pseudocode we abstract this into a function *Open*().

Protocol Π outputs an event Π-*deliver*(b) whenever a block b is delivered (L 401). According to the analysis of our protocol, this will allow Π^3 to deliver the block $\tau_1 + \tau_2$ positions higher than b, i.e., the block b_{del} at height $i_{\text{del}} = b.height - \tau_1 - \tau_2$. To this goal, Π^3 first reads the commitments related to b_{del} (L 403-404). By construction of Π^3, a commitment $c_{i,j}$, written on block b_i, is used to order the transactions in block b_{i+j}. Hence, the commitments related to b_{del} have been written on the n_ℓ blocks before b_{del}. Protocol Π^3 then reads the openings to these commitments (L 405-408). Again by construction of Π^3, the openings of the commitments related to b_{del} have been written on the blocks with height $i_{\text{del}} + \tau_1 + 1$ to $i_{\text{del}} + \tau_1 + \tau_2$. For each of these blocks, Π^3 loops through its transactions that contain an opening. L 407 then checks whether the opening is for a commitment related to block b_{del} and whether the opening is valid. Protocol Π^3 then calculates the final permutation Σ to be applied to block b (L 409-413). As presented in Section 7.4.1, $\Sigma = \mathit{PermFromRandBits}(G(\mathit{seed}))$, where *seed* is the XOR of all valid openings for block b, G is a pseudorandom generator, and *PermFromRandBits* an algorithm that derives a permutation from random bits. The remaining of this block chunks the transactions con-

Algorithm 22 Protocol Π^3. Code for party P_i.

Implements: Protocol Π^3
Uses: block-based atomic broadcast Π

State:
392: $\sigma[i,j] \leftarrow \perp$, for all $i \geq 1, j \in [n_\ell]$
393: $c[i,j] \leftarrow \perp$, for all $i \geq 1, j \in [n_\ell]$

394:**upon event** $\langle \Pi^3\text{-}broadcast, tx \rangle$ **do**
395: **invoke** $\langle \Pi\text{-}broadcast, tx \rangle$

396:**upon event** $\langle \Pi\text{-}mined, b, Q \rangle$ **do**
397: $i_{\text{open}} \leftarrow b.height - \tau_1 - 1$
398: **for** $i' \in [i_{\text{open}} - n_\ell - 1, i_{\text{open}} - 1]$ **do**
399: **if** $b.chain[i'].miner = P$ **then**
400: $Open(b.chain[i'].commitments[i_{\text{open}} - i'])$

401:**upon event** $\langle \Pi\text{-}deliver, b \rangle$ **do**
402: $i_{\text{del}} \leftarrow b.height - \tau_1 - \tau_2$
403: **for** $j \in [n_\ell]$ **do** // Read commitments for b
404: $c[i_{\text{del}}, j] \leftarrow b.chain[i_{\text{del}} - j].commitments[j]$
405: **for** $i' \in [i_{\text{del}} + \tau_1 + 1, i_{\text{del}} + \tau_1 + \tau_2]$ **do** // Read the openings for b
406: **for** $tx \in b.chain[i'].txs$ **such that** $tx = open(k, l, \sigma)$ **do**
407: **if** $k + l = i_{\text{del}}$ **and** $H(\sigma) = c[i_{\text{del}}, l]$ **then**
408: $\sigma[i_{\text{del}}, l] \leftarrow \sigma$
409: $seed \leftarrow 0^\lambda$
410: **for** $j \in [n_\ell]$ **do** // Compute final permutation for b
411: **if** $\sigma[i_{\text{del}}, j] \neq \perp$ **then**
412: $seed \leftarrow seed \oplus \sigma[i_{\text{del}}, j]$
413: $\Sigma \leftarrow PermFromRandBits(G(seed))$
414: $chunked\_txs \leftarrow [\,]$
415: **for** $tx \in b.txs$ **do** // Chunk and permute transactions in b
416: $chunked\_txs \leftarrow chunked\_txs \| Chunk(tx, m)$
418: $chunks \leftarrow Permute(\Sigma, chunked\_txs)$
419: $chunks \leftarrow SwapChunks(chunks)$ // swap the first chunk with tx^1
420: $b.txs \leftarrow chunks$
421: **invoke** $\langle \Pi^3\text{-}deliver, b \rangle$

Algorithm 23 Protocol Π^3. Code for party P_i.

```
422:function FB(txs) :
423:     data ← [ ]
424:     for tx ∈ txs do
425:          data ← data‖tx
426:     for j ∈ [n_ℓ] do
427:          σ ←$ {0,1}^λ
428:          c ← H(σ)
429:          data ← data‖c
430:     return data

431:function VB(b) :
432:     if (∃tx ∈ b.txs : ¬VT(tx)) ∨ (∃j ∈ [n_ℓ] : b.commitments[j] = ⊥) then
433:          return FALSE
434:     return TRUE
```

tained in b (L 414-416), applies Σ on the chunked transactions (L 417), and swaps the first permuted chunk of each of each transaction with the chunk containing the code (L 419). The function *Chunk*() is explained in Section 7.4.2. Finally, Π^3 delivers block b containing the chunked and permuted transactions through the Π^3-*deliver*(b) event (L 421).

The function *FB*() is an upcall from block-based atomic broadcast. It specifies how a block is filled with transactions and additional data. For simplicity, the pseudocode omits any detail specific to *bab*. It first writes all given transactions on the block, then picks uniformly at random n_ℓ bit-strings of length λ. These are the partial random seeds to be used in the permutation of the following n_ℓ blocks, if the block that is currently being built gets mined and delivered by *bab*. The commitments to these partial seeds are appended on the block.

Finally, the predicate *VB*() specifies that a block is valid if all its transactions are valid, as specified by *VT*(), and if it contains n_ℓ commitments. The predicate *VT*() is omitted, as its implementation does not affect Π^3.

7.5 Analysis

7.5.1 Security analysis

We model the adversary as an interactive Turing machine (ITM) that corrupts up to t parties at the beginning of the execution. Corrupted parties follow the instructions of the adversary and may diverge arbitrarily from the execution of the protocol. The adversary also has control over the *diffusion functionality*. That is, she can schedule the delivery of messages (within the Δ rounds), as well as read the $RECEIVE_i$ of every party at any moment of the execution and directly write in the $RECEIVE_i$ of any party.

We first show that the security of our construction is derived from the security of the original protocol. Given an execution of protocol Π^3, we define the equivalent execution in protocol Π as the execution in which every party follows the same steps but the commitment, opening, and randomization of transactions are omitted. We also recall the parameters τ_1 and τ_2 that denote the length (in blocks) of the silent and loud phase respectively.

Lemma 39. *The probability that an adversary can rewrite a block after any honest partial permutations have been opened is negligible in τ_1.*

Proof. Assume an adversary controlling up to t parties and a block B. We know that if $\tau_1 > d$, protocol Π would deliver block B, thus an adversary cannot revert the chain to modify the order of the transactions stored in B but with negligible probability. □

Lemma 40. *The probability that an adversary can rewrite a chain omitting the opening of some honest partial permutation is negligible in τ_2.*

Proof. The fairness quality of protocol Π states that for any consecutive N blocks, if $N \geq N_0$ the fraction of honest blocks is at least μ. Thus, if $\tau_2 \geq \max\{N_0, \frac{1}{\mu}\}$, there exists at least one honest block containing every opening that is not previously included in the chain. Since □

Our construction aims to turn any protocol into a protocol robust against sandwich attacks. However, there might be new vulnerabilities. Intuitively, our construction should not introduce any vulnerability because the only modified aspect is the order in which transactions are delivered. Theorem 41 formalizes this intuition.

Remark 5. Note that every Π-*delivered* block is also Π^3-*delivered* some block after (Line 401–421). Note also that every Π^3-*delivered* is also Π*delivered*. Furthermore, the blocks are delivered in the same order.

Theorem 41. *If protocol* Π *implements block-based atomic broadcast, then the Partitioned and Permuted Protocol* Π^3 *implements block-based atomic broadcast.*

Proof. According to Remark 5, the set of Π^3-*delivered* blocks is the same as the set of Π-*delivered* blocks.

Validity: Assume that an honest party Π^3-*broadcasts*(tx) transaction tx. The party first Π-*broadcasts*(tx) (L 394–395). The validity property of protocol Π guarantees that eventually a block b containing transaction tx is Π-*delivered*. According to Remark 5, the honest party eventually Π^3-*delivers* a block containing tx and Π^3 satisfies the validity property of block-based atomic broadcast.

No-duplication: Note that Π^3 delivers the same set of blocks as protocol Π, Remark 5. Thus, the no-duplication property of protocol Π^3 is inherited directly from the no-duplication of protocol Π.

Agreement: Consider two honest parties P_i and P_j such that party P_i Π^3-*delivers* block b. Remark 5 guarantees that P_i alsoΠ-*delivers* block b. The agreement property of protocol Π ensure that P_j eventually Π-*delivers* block b. Remark 5 guarantees that P_j eventually Π^3-*delivers* block b. Note that the block b delivered by both P_i and P_j may differ in how the transactions are chunked and permuted. However, Lemmas 39 and 40 guarantee all correct parties agree on the same permutation with all but negligible probability. Hence, we conclude that protocol Π^3 satisfies the agreement property.

Total order: Remark 5 guarantees that the order in which any honest party Π^3-*delivers* two block b_1 and b_2 is the same as it Π-*delivers* them. Thus, the total order property of protocol Π guarantees the total order property of protocol Π^3.

External validity: This follows from the external validity of Π.

Fairness: According to Remark 5 the same blocks and in the same order are both Π^3-*delivered* and Π-*delivered*. Hence, the fairness property of Π^3 is inherited from the fairness property of protocol Π.

□

After showing that Π^3 is as secure as the original protocol Π. We turn our attention to analyzing the behavior of Π^3 under sandwich attacks, in the upcoming section.

7.5.2 Game-theoretic analysis

Here, we aim to show that if we assume all miners are rational, i.e., they prioritize maximizing their own payoff, behaving honestly as according to our protocol Π^3 is a stable strategy.

Strategic games. For $N \in \mathbb{N}$, let $\Gamma = (N, (S_i), (u_i))$ be an N party game where S_i is a finite set of strategies for each party $i \in [N]$. Let $S := S_1 \times \cdots \times S_N$ denote the set of outcomes of the game. The utility function of each party i, $u_i : S \to \mathbb{R}$, gives the payoff of party i given an outcome of Γ. For any party i, a *mixed strategy* s_i is a distribution in $\mu(S_i)$. A *strategy profile* of Γ is $s := s_1 \times \cdots \times s_N$ where s_i is a mixed strategy of party i. The *expected utility* of a party i given a mixed strategy profile s is defined as $u_i(s) = \mathbb{E}_{a_1 \leftarrow s_1, \cdots, a_N \leftarrow s_N}[u_i(a_1), \cdots, u_i(a_N)]$. Finally, we note that if s_i is a Dirac distribution over a single strategy $a_i \in S_i$, we say s_i is a *pure strategy* for party i.

Notation. Let w denote the total reward for mining a block and q the negligible probability that a PPT adversary guesses a correct opening. Recall in Section 7.4.1 that the total block reward w is split between the miner of the block who gets $\alpha \cdot w$ and the miners that append the correct openings who get $\frac{(1-\alpha)\cdot w}{n_\ell}$ for each correct opening they append. For a given block, we denote by m the number of chunks for *each* transaction in the block, and by λ the utility of the sandwich attack on the block. Specifically, λ refers to the utility of a sandwich attack performed on

the original transactions in the order they are in *before* chunking and permuting them. We also denote the optimal sandwich utility by Λ, which is the maximum utility one can get by performing a sandwich attack. Finally, we denote by $\hat{\lambda}_i$ the *average* utility of the sandwich attack taken over all blocks on the chain for a specific miner M_i. This can be computed easily as the transaction mempool is public. We stress that it is important to look at the average sandwich utility for each miner separately and not the average over all miners as the utility a miner can derive from a sandwich attack depends on their available liquidity (i.e., how much assets they can spare to front-run and back-run the transactions).

Quasi-strong ε-Nash Equilibrium. In terms of game theoretic security, we want our protocols to be resilient to deviations of any subset of miners that form a coalition and deviate jointly. The security notion we want to achieve is that of an *quasi-strong ε-Nash Equilibrium* [13, 21, 36]. Let C denote the coalition of players. For any strategy profile s, we denote by $u_C(s)$ the expected utility of the coalition under s. We denote by $u_C(s'_C, s_{-C})$ the expected utility of the coalition when playing according to some other strategy profile s'_C given the other players that are not part of the coalition play according to s.

Definition 39. (quasi-strong ε-Nash Equilibrium) A quasi-strong ε-Nash Equilibrium is a mixed strategy profile s such that for any other strategy profile s'_C, $u_C(s) \geq u_C(s'_C, s_{-C}) - \varepsilon$ for some $\varepsilon > 0$.

The notion of a quasi-strong Nash Equilibrium is particularly useful in the context of blockchains as the coalition could potentially be controlled by a single miner with sufficient resources [36]. The notion of an ε-equilibrium is also important in cases where there could be a small incentive (captured by the ε parameter) to deviate from the protocol, and of course the smaller one can make ε, the more meaningful the equilibrium.

Subgame perfection. We also consider games that span several rounds and we model them as extensive-form games (see, e.g., [90] for a formal definition). Extensive form games can be represented as a game tree tx where the non-leaf vertices of the tree are partitioned to

sets corresponding to the players. The vertices belonging to each player are further partitioned into information sets I which capture the idea that a player making a move at vertex $x \in I$ is uncertain whether they are making the move from x or some other vertex $x' \in I$. A subgame of an extensive-form game corresponds to a subtree in tx rooted at any non-leaf vertex x that belongs to its own information set, i.e., there are no other vertices that are the set except for x. A strategy profile is a *quasi-strong subgame perfect ε-equilibrium* if it is a quasi-strong ε-Nash equilibrium for all subgames in the extensive-form game.

The induced game. Let us divide our protocol into epochs: each epoch is designed around a given block say B_i and begins with the generation of random seeds for B_i and ends with appending the openings for the committed random seeds for B_i (i.e., block $B_{i+\tau_1+\tau_2}$). We define the underlying game Γ induced by any given epoch of our protocol Π^3. Γ is a (τ_2+1)-round extensive form game played by $n_\ell + \tau_2$ parties (n_ℓ leaders comprising the leader set L_i for any block B_i and the τ_2 miners that mine the blocks $B_{i+\tau_1+1} \dots B_{i+\tau_1+\tau_2}$). Note that although we have $\binom{N}{\tau_2}$ sets of τ_2 miners to choose from (where N is the total number of miners in the chain) to be the miners of the blocks $B_{i+\tau_1+1} \dots B_{i+\tau_1+\tau_2}$, we can simply fix any set of τ_2 miners together with L_i to be the parties of Γ as we assume all miners are rational and so the analysis of the utilities of any set of τ_2 miners will be the same in expectation. We use A to denote the set of all miners in τ_2. In what follows, we assume an arbitrary but fixed ordering of the miners in A. Round 1 of Γ consists of only the parties in L_i performing actions, namely picking a random seed and committing to it. In rounds $2, \dots, \tau_2+1$ of Γ, each member of L_i can act by choosing to open their commitment or not. However, the moment a member of L_i opens its commitment in a given round, they lose the chance to open their commitment in any subsequent round. Only one miner from A and according to the imposed ordering acts in each round from round 2 to τ_2+1 of Γ. The choice of actions of the miner in any of these rounds are the subsets of the set of existing commitment openings (from members of L_i) to append to their block. Finally we note that the $L_i \cap A$ is not necessarily empty and thus miners in the intersection can choose to open and append their commitment in the same round.

Let us define the honest strategy profile as the profile in which all members of L_i choose to generate a random seed in round 1 of Γ, all members

of L_i open their commitments at round τ_2 (i.e., at block $B_{i+\tau_1+\tau_2-1}$), and each member of A appends all existing opened commitments that appear in the previous round. We denote the honest strategy profile by s. The security notion we want to achieve for our protocol is a quasi-strong subgame perfect ε-equilibrium (refer to Definition 39). Looking ahead, we will also prove that ε can be made arbitrarily small by increasing the number m of chunks.

Lemma 42. *The expected utility of an honest leader is at least* $(1-q)^{n_\ell}\alpha w$.

Proof. The expected utility for a user following the honest strategy comes from the sum of the block reward, the expected utility from the ordering of any of their transactions within the block, and appending valid openings of committed seeds (if any) to their blocks. The expected utility from the ordering of transactions is 0 due to symmetry: each possible order is equally likely, for each order that gives some positive utility, there exists a different order producing the same negative utility. The expected utility from the block reward is $(1-q)^{n_\ell}\alpha w$. Thus, the total expected utility of an honest miner is at least $(1-q)^{n_\ell}\alpha w$. □

We outline and analyze two broad classes of deviations or attacks any coalition can attempt in this setting. The first class happens at round 1 of Γ where the members of the coalition commit to previously agreed seeds to produce a specific permutation of the transactions. The coalition then behaves honestly from round 2 to $\tau_2 + 1$ of Γ. We call this attack the *chosen permutation attack* and denote this attack strategy by s_{CP}. In the second class, the coalition behaves honestly at round 1 of Γ, but deviates from round 2 onwards where some members selectively withhold opening or appending commitments to bias the final permutation. We call this attack the *biased permutation attack*, and denote it by s_{BP}.

Chosen permutation attack. Before we describe and analyze the chosen permutation attack (for say a block B_i), we first show that a necessary condition for the attack to be successful, that is, the coalition's desired permutation happens almost surely, is that at least all n_ℓ leaders in L_i have to be involved in the coalition (members of A can also

be involved in the coalition, however as we will show this will simply increase the cost). To do so, we let S denote the set of permutations over the list of transactions and their chunks, and we define what we mean by a protocol Π_{perm} (involving n parties) outputs random a permutation in S by the following indistinguishability game called *random permutation indistinguishability* played between a PPT adversary, a challenger, and a protocol Π_{perm}. First, the adversary corrupts up to $n-1$ parties. The adversary has access to the corrupted parties' transcripts. Then, the challenger samples σ_0 uniformly at random from S, and sets σ_1 to be the output of Π_{perm}. After that, the challenger flips a random bit b and sends σ_b to the adversary. The game ends with the adversary outputting a bit b'. If $b' = b$, the adversary wins the game. We say a protocol Π_{perm} outputs a random permutation if the the adversary wins the above game with probability $\frac{1}{2} + \varepsilon$ for some negligible ε. Let us define the output of a single round of Π^3 as the random permutation that is generated from the seeds generated from all leaders in the round according to the algorithm described in Section 7.4.1. The following lemma states that as long as a single leader is honest, the output of Π^3 is pseudorandom.

Lemma 43. *An adversary that corrupts at most $n_\ell - 1$ leaders in a single round of Π^3 can only win the random permutation indistinguishability game with negligible probability.*

Proof. The proof follows in the same way as introduced by M. Blum [22], with the addition of the PRG. □

Lemma 43 implies that launching the chosen permutation attack and thus choosing to deviate at round 1 of Γ comes with an implicit cost: either a single miner has to mine n_ℓ blocks in a row so the miner single-handedly forms the coalition, or *all* leaders in L_i have to be coordinated into playing according to a predefined strategy.

Lemma 44. *Given the underlying blockchain is secure, the expected utility of the single miner when playing according to s_{CP} is at most $\frac{\lambda}{2^{n_\ell}}$ more than the expected utility of following the honest strategy.*

Proof. Since the underlying blockchain is secure, a necessary condition is that a single miner cannot own more than $\frac{1}{2}$ of the total amount of

resources owned by all miners of the protocol. Thus, the probability of mining n_ℓ blocks in a row is strictly less than $\frac{1}{2^{n_\ell}}$. This means that the expected utility under the attack strategy $u_C(s_{CP}) < \frac{\lambda}{2^{n_\ell}} + n_\ell \alpha w$, which is at most $\frac{\lambda}{2^{n_\ell}}$ larger than the expected utility under the honest strategy which is $u_C(s) = n_\ell \alpha w$. □

The attack strategy of a coalition composed by more than one miner is more complex compared to the case where there is a single miner, as the coalition needs to ensure its members coordinate strategies. First, the coalition works with the miner of block B_i to select and fix a permutation generated by a specific PRG seed σ_i. Then, the coalition secret shares σ_i among its members[7]. After that, the coalition sets up some punishment scheme to penalize members that do not reveal their partial seeds[8]. Finally, the coalition commits and reveals these partial seeds in accordance to the protocol Π^3. Let $\mathcal{C}$ denote the expected cost of coordinating the whole chosen permutation attack for the coalition. For this attack to succeed, the expected coordination cost has to be smaller than the expected profit λ.

Lemma 45. *The chosen permutation attack fails to be profitable compared to the honest strategy if* $\mathcal{C} > \lambda$.

Proof. From Lemma 42, the expected revenue of an honest miner is $(1-q)^{n_\ell} \alpha w$, thus the expected revenue of the coalition when following the honest strategy is $u_C(s) = n_\ell \cdot (1-q)^{n_\ell} \alpha w$. The expected revenue for the chosen permutation attack strategy is $u_C(s_{CP}) = n_\ell \cdot (1-q)^{n_\ell} \alpha w + \lambda - \mathcal{C}$. Thus, assuming $\mathcal{C} > \lambda$, and since the expected revenue from a mixed strategy is a convex combination of the revenues of the honest and attack strategies, the pure honest strategy gives a strictly larger expected payoff compared to any mixed strategy. □

Remark 6. Computing, or even estimating, the coordination cost is non-trivial as it consists of several dimensions and also depends on a

[7] This not only prevents members from knowing the partial seeds of other members and hence stealing their block reward, but also additionally safeguards the partial seeds of the members against the miner of block B_i who cannot generate a partial seed of their block and hence has nothing to lose.

[8] This ensures that every member will reveal reveal their partial seeds and the permutation will be generated properly.

myriad of factors and assumptions. A few notable costs are, firstly, timing costs. The coalition has to convince and coordinate all the leaders to agree on a permutation and also commit and reveal them during a short interval of d blocks. This involves the cost of securely communicating with all the leaders and also the computational cost involved in setting up the secret sharing scheme. A second factor is the choice of the initial order of transactions, which the coalition would have to also agree on with the miner of the attacked block. Picking transactions greedily would be the simplest choice as finding the optimal set of transactions from the mempool is NP-hard [85]. Finally, the coalition has to set up a punishment scheme to penalize members that do not reveal their permutations. If we ignore the cost of setting up such a scheme, this can be implemented using a deposit scheme with the size of the deposit at least the value of the expected additional per user profit from the sandwich attack [102]. This implies an opportunity cost at least linear in $\frac{\lambda}{n_\ell}$, as well as the assumption that each member has at least $\frac{\lambda}{n_\ell}$ to spare to participate in the attack. Additionally, we note that the coalition could extend to miners from A which are outside the leader set L_i. However, since these miners do not contribute to generating the random seeds, they simply add to the communication cost of the coalition. Finally, we note that the coordination cannot be planned in advance due to the unpredictability of the block mining procedure.

Biased permutation attack. The intuition behind this attack is that any coalition that controls $k \leq n_\ell$ commitments can choose to select the ones to open or append, which allows the coalition to chose among 2^k possible permutations in order to bias the final ordering. This can be achieved in two situations: either k out of n_ℓ leaders of L_i form a coalition and decide which of their commitments to open, or some subset of miners in the loud phase (of size say $\tilde{k}$) form a coalition and end up controlling k openings, let $\kappa := \min\{k, \tilde{k}\}$. Unlike in the case of the chosen permutation attack, it suffices consider the case where we have a *single* miner that happens to either occupy k leader positions among the group of leaders L_i or mine the $\tilde{k}$ blocks that belong to the coalition in the loud phase. This is because the case where a coalition of distinct miners that collude only adds additional coordination cost. The probability that any such coalition gains any additional utility by performing the biased permutation attack compared to the honest strategy can be upper-bounded. Let *revenue* denote the utility the coalition would gain

from performing the biased permutation attack.

Lemma 46. *The probability that a coalition of κ members performing the biased permutation attack achieves utility of at least $\kappa w > 0$ is* $\mathsf{P}[\textit{revenue} \geq \kappa w] \leq 1 - (1 - e^{-\frac{2m\kappa w}{\lambda}})^{2^k}$.

Proof. Given a random permutation and a sandwich attack with original utility λ (utility if the order of the transactions were not randomized), denote by $\{X_i(\sigma)\}_{i=1}^m$ the utility produced by chunk i. The sum of these random variables $X(\sigma) = \sum_{i=1}^n X_i(\sigma)$ represents the total utility of a sandwich attack (after chunking and permuting). X takes values in $[-\lambda, \lambda]$, thus the variables $\{X_i(\sigma)\}$ take values in $[-\frac{\lambda}{m}, \frac{\lambda}{m}]$, are equally distributed and are independent. We define the random variables $Y_i(\sigma) = X_i(\sigma) + \frac{\lambda}{m} \in [0, \frac{2\lambda}{m}]$, and $Y(\sigma) = \sum_{i=1}^n Y_i(\sigma) \in [0, 2\lambda]$. Using lemma 42, $\mathrm{E}[Y_i(\sigma)] = \frac{\lambda}{m}$ and $\mathrm{E}[Y(\sigma)] = \lambda$. Applying Chernoff's bound [87] to Y,

$$\mathsf{P}[Y(\sigma) \geq (1+\delta)\mathrm{E}[Y(\sigma)]] \leq e^{\frac{-2\delta^2 \mathrm{E}[Y(\sigma)]^2}{m(\frac{\lambda}{m})^2}} = e^{-2m\delta^2} \tag{7.1}$$

for $\delta > 0$. We can rewrite Equation 7.1 as follows:

$$\begin{aligned}\mathsf{P}[\textit{revenue}(\sigma) \geq \delta\lambda] &= \mathsf{P}[\textit{revenue}(\sigma) + \lambda \geq (1+\delta)\lambda] \\ &= \mathsf{P}[Y(\sigma) \geq (1+\delta)\mathrm{E}[Y(\sigma)]] \leq e^{-2m\delta^2}.\end{aligned}$$

Using the law of total probability we obtain that $\mathsf{P}[\textit{revenue}(\sigma) \leq \delta\lambda] \geq 1 - e^{-2m\delta^2}$. Considering the maximum over the 2^k possible permutations σ and $\delta = \frac{\kappa\omega}{\lambda}$ we conclude that

$$\begin{aligned}\mathsf{P}[\textit{revenue} \geq \kappa w] &= 1 - \mathsf{P}[\textit{revenue} \leq \kappa w] = 1 - \mathsf{P}[\textit{revenue}(\sigma) \leq \kappa w] \\ &\leq 1 - (1 - e^{-\frac{2m\kappa w}{\lambda}})^{2^k}.\end{aligned}$$

□

Lemma 47. *The probability that a coalition of κ members has positive additional utility is:* $\mathsf{P}[\textit{revenue} \geq 0] \leq \max_{k' \leq \kappa} \left\{1 - (1 - e^{-\frac{2mk'w}{\lambda}})^{2^k}\right\}$.

Proof. Lemma 46 states a bound for the probability that a coalition of κ parties has a utility of at least $\kappa w > 0$, the penalty for not opening κ commitments. Thus, the general case for a coalition aiming to maximize profit is the maximum over $k' \leq \kappa$.

□

Recall that Λ is the maximal utility and let $p_{k,\lambda}$ denote $\max_{k' \leq \kappa}\{1 - (1 - e^{-\frac{2m(1-q)^{n_\ell} k' w}{\lambda}})^{2^k}\}$. Then, the expected additional utility from the biased permutation attack of a single miner controlling k leaders is no greater than $p_{k,\lambda}\Lambda$.

Lemmas 45,46 and 47 allow us to prove our main theorem.

Theorem 48. *Suppose $\mathcal{C} > \lambda$, then the honest strategy $s = ((\text{random seed})_{i=1}^{n_\ell}, (\text{open})_{i=1}^{n_\ell})$ is a quasi-strong subgame perfect ε-equilibrium in Γ for $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$.*

Proof. We first observe that the expected utility of a coalition that mixes both the chosen and biased permutation attack strategies is no greater than the expected utility of a coalition that performs the chosen permutation attack with a different chosen permutation that accounts for the biasing of the permutation in the second round of Γ. Hence, it suffices to analyze the expected utility of the coalition when implementing either of these strategies, i.e., deviating at round 1 of Γ or from rounds 2 onwards.

We first analyze the expected utility of a coalition when implementing the chosen permutation attack, which occurs at round 1 or Γ. Since we assume $\mathcal{C} > \lambda$, from Lemma 44 and Lemma 45, we see that any additional expected payoff of any coalition that deviates only at round 1 of Γ by implementing the chosen permutation attack compared to the expected revenue of behaving honestly is at most $\frac{\lambda}{2^{n_\ell}}$.

Now we analyze the expected utility of a coalition when implementing the biased permutation attack. From Lemmas 46 and 47, we see that the strategy that implements the biased permutation attack across all of rounds 2 to τ_2+1 of Γ only gives at most $p_{k,\lambda}\Lambda$ more payoff in expectation compared to following the honest strategy s in these rounds.

As such, if we set $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$ to be the largest difference in additional expected revenues between both strategies, we see that $s = ((\text{random seed})_{i=1}^{n_\ell}, (\text{open})_{i=1}^{n_\ell})$ is a quasi-strong ε-subgame perfect equilibrium of Γ.

□

Remark 7. Recall that ε bounds the additional expected utility an adversary can gain by deviating from the honest strategy profile s. The security of our protocol therefore improves as $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$ decreases. We observe that the first component $\frac{\lambda}{2^{n_\ell}}$ goes to 0 exponentially as the size of the leader set n_ℓ increases. As for the second component $p_{k,\lambda}$, we conduct an empirical analysis of sandwich attacks on Ethereum, Section 7.6, to estimate $p_{k,\lambda}$ and we show that this value approaches zero as the number of chunks m increases.

7.6 Case study: Ethereum MEV attacks

We validate the utility of our results with real-world data from Ethereum. Specifically, we estimate, using Lemma 47, the probability that a coalition of k parties obtains positive revenue, for various values of k, sandwich revenue λ, and chunks m. We conclude by analyzing the overhead incurred by protocol Π^3 as a function of m and its security-efficiency trade-offs.

Empirical security analysis. We obtain the data on the profit of sandwich attacks on Ethereum using the Eigenphi tool[9] for October 2022. We considered October 2022 because it is the month with the highest amount of Sandwich attacks[10], thus its data represents better the state of the art of sandwich attacks. To convert between ETH and USD we use the price of ETH as of October 31st, approx. $1,570$ USD. The block reward at this time is 2 ETH[11], or approx. $3,140$ USD. In Figure 7.3 we show the number of attacks in bins of increasing profit, as returned by Eigenphi. From this data we make use of two facts.

[9] https://eigenphi.io/mev/ethereum/sandwich
[10] https://eigenphi.io/mev/research
[11] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1234.md

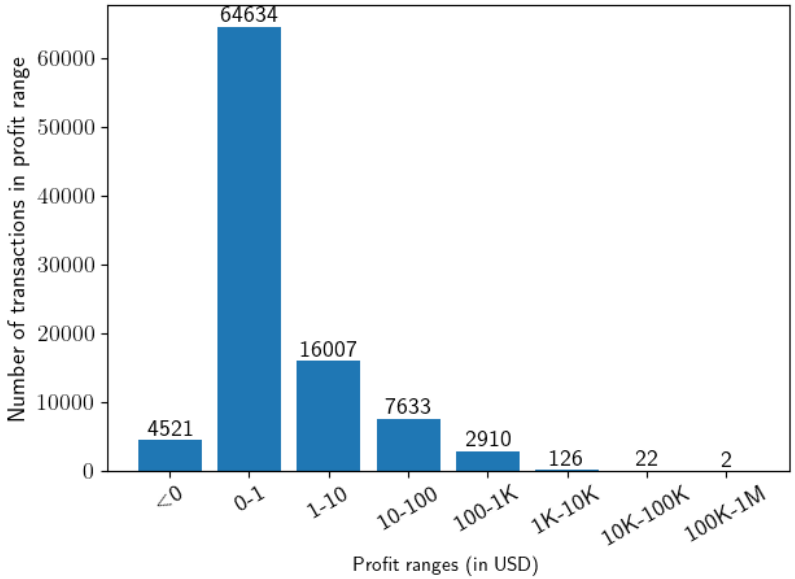
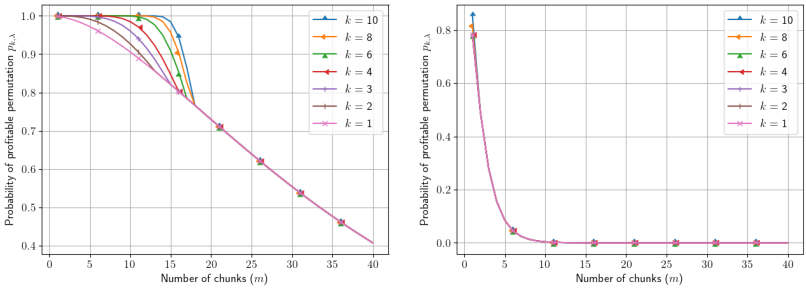


Figure 7.3. Detected sandwich attacks on Ethereum in October 2022, grouped by their profit range. It can be observed that the majority of the detected attacks had a profit of at most 1 USD, and that 99.97% of them had a profit of at most 10K USD.

First, 99.97% of the attacks had profit lower than 10K USD, or approx. 6.37 ETH, and second, the most profitable sandwich attack had a profit of 170,902.35 USD, or approx. 109 ETH. Hence, we define $\lambda_{99.97} = 6.37$ and $\lambda_{\max} = 109$. In Figure 7.4 we plot an upper bound for the probability of positive revenue for a coalition of k leaders, considering a sandwich revenue of $\lambda_{\max}$ (Figure 7.4a) and $\lambda_{99.97}$ (Figure 7.4b). We observe that, even for the largest observed sandwich revenue, $\lambda_{\max}$, the probability of a profitable attack drops below 0.5 for $m = 33$. For $\lambda_{99.97}$, the probability is low even for small values of m. For example, already for $m = 2$ we get $p_{k,6.37} \simeq 0.49$, and for $m = 10$ we get $p_{k,6.37} \simeq 0.0038$, for all $k \geq 1$. We also remark that Lemma 47 states upper bound for the adversary to have *some positive* revenue.



(a) $\lambda = 109$ ETH **(b)** $\lambda = 6.37$ ETH

Figure 7.4. The upper bound for the probability of positive revenue $p_{k,\lambda}$ (according to Lemma 47) for a coalition of k leaders, versus the number of chunks m. Fig. 7.4a considers $\lambda = 109$ ETH, the *maximum* value among all sandwich attacks detected on Ethereum in October 2022, and Fig. 7.4b considers $\lambda = 6.37$ ETH, the 99.97-*th percentile* of the values. Fig. 7.4a shows that for small number of chunks a larger coalition has a noticeable advantage over a single party, while for bigger values of m this advantage fades away, as the number of possible permutations grows factorially in m, reducing the possible bias of a coalition. In Fig. 7.4b, on the other hand, we observe a low probability of positive revenue, even for small values of m (e.g., 0.0038 for $m = 10$). Notice that this is the case for 99.97% of the attacks. In both figures, parts of the plots coincide. This is because a large m (which implies a factorial increase in the number of possible permutations), combined with the value of the sandwich revenue make it unprofitable to sacrifice more than one block reward.

Overhead. In terms of space, each block contains exactly n_ℓ commitments and on average n_ℓ openings of partial seeds. A commitment to a partial seed takes 256 bits of space. Assuming openings are implemented as a call $open(i, j, \sigma_{i,j})$ to a smart contract, where i and j are 16-bit integers and an address is 160 bits long, then each opening consumes 468 bits on the block. In total, Π^3 incurs on average an overhead of $724n_\ell$ bits per block. As an example, for $n_\ell = 10$, this results in an average overhead of less than 1 KB per block. We remark that chunking of transactions happens locally, and hence adds no space overhead on the block. Concerning execution, there are two main sources of overhead in Π^3. First, when a block is delivered parties compute the final permutation of its $n_t m$ transactions using *PermFromRandBits*(), which has linear-logarithmic bit complexity. The overhead is thus $O(n_t m \cdot \log(n_t m))$. Moreover, parties execute $n_t m - n_t$ more transactions, which incurs an overhead of $O(n_t m)$. In total, considering n_t to be a constant and m a parameter to Π^3, the execution overhead scales as $O(m \log m)$. We remark here that the vast majority of computational resources is used in the mining mechanism, and this is not changed from Π. Finally, Π^3 incurs an increased latency when delivering transactions. While Π has a latency of d blocks, Π^3 has a latency of $\tau_1 + \tau_2 + d$ blocks. As discussed earlier, a smaller τ_1 certainly decreases the latency of the protocol, but it also increases the probability of rewriting a block after the commitments that order its transactions have been opened. Similarly, a smaller τ_2 decreases the latency but gives the miners a shorter time frame to open their commitments. We stress that *not* opening a commitment does not impact the latency of our protocol at all. The only impact it has is on the block reward of the miner who owns the commitment. We also stress that the computational overhead involved in generating the partial seeds in minimum compared with other traditional solutions such as time-lock puzzles [98].

Security-efficiency tradeoffs. We first observe a tradeoff between security of Π^3 and computational overhead. On the one hand, increasing the number of chunks improves the security of Π^3. Recall that $\varepsilon = \max\{\frac{\lambda}{2^{n_\ell}}, p_{k,\lambda}\Lambda\}$, and in Remark 7 we highlight that $p_{k,\lambda}\Lambda$ goes to zero as the number of chunks m increases. In Figure 7.4 we see how $p_{k,\lambda}$ changes with m, based on historical data. Specifically, for the vast majority of observed sandwich attacks (in Figure 7.4b we use the 99.97-th percentile) the probability of a coalition to succeed drops expo-

nentially with m. On the other hand, the execution overhead increases as $O(m \log m)$. Moreover, the size of leader set n_ℓ leads to the following tradeoff. In Remark 7 we show that the security of Π^3 against rational adversaries improves exponentially with n_ℓ. However, the size of the leader set also determines the number of leader sets each miner needs to be a part of, and hence the length of time each miner has to wait until it receives its block reward.

7.7 Sandwich MEV attacks

Decentralized exchanges. Decentralized exchanges (DEXes) allow users to exchange various cryptocurrencies in a decentralized manner (i.e., in a peer-to-peer fashion without a central authority). Some examples of DEXes on the Ethereum blockchain are Uniswap[12] and Sushiswap[13] DEXes typically function as constant product market makers (CPMMs) [11], i.e., the exchange rate between any two underlying assets is automatically calculated such that the product of the amount of assets in the inventory remains constant.

As an example, consider the scenario where a user at time t wants to swap δ_X of asset X for asset Y in the $X \rightleftharpoons Y$ liquidity pool, and suppose the pool has X_t and Y_t amount of assets X and Y in its inventory at time t. The user would receive

$$\delta_Y = Y_t - \frac{X_T \cdot Y_T}{X_t + (1-f)\delta_X} = \frac{Y_t(1-f)\delta_X}{X_t + (1-f)\delta_X}$$

amount of asset Y for δ_X amount of asset X, where f is a fee charged by the pool [62]. We can thus compute the exchange rate of X to Y at time t as

$$\rho_t^{XY} := \frac{\delta_X}{\delta_Y} = \frac{X_t + (1-f)\delta_X}{Y_t(1-f)} \tag{7.2}$$

Sandwich attack. As transactions in a block are executed sequentially, the exchange rate for a swap transaction could depend on where

[12] https://uniswap.org
[13] https://sushi.com

the transaction is located in the block. From Equation (7.2), we note that the exchange rate from X to Y increases with the size of the trade δ_X. Thus, if a transaction that swaps X for Y occurs *after* several similar X to Y swap transactions, the exchange rate for this particular transaction would increase. Consequently, the user which submitted this transaction would pay more per token of Y as compared to if the transaction occurred *before* the other similar transactions. In a sandwich attack, the adversary (usually miner) manipulates the order of the transactions within a block such that they can profit from the manipulated exchange rates. Specifically, the adversary is given the list $\mathcal{T}$ of all transactions that can be included in a block and can make two additional transactions tx_1 and tx_2: transaction tx_1 exchanges some amount (say δ_X) of asset X for asset Y, and tx_2 swaps the Y tokens from the output of tx_1 back to X. Let us denote the amount of tokens of X the adversary gets back after tx_2 by δ'_X. The goal of the adversary is to output a permutation over $\mathcal{T} \cup \{tx_1, tx_2\}$ such that $\delta'_X - \delta_X$ is maximized. A common technique is to front-run all transactions exchanging X to Y in the block, i.e., place transaction tx_1 before all transactions exchanging X to Y and tx_2 after [61]. We note that users can protect themselves by submitting a slippage bound $sl > 0$ together with each transaction. However, this protection is only partial.

7.8 Sandwich attacks with random permutation

Here we outline a way an adversary can still launch a sandwich attack even when the transactions in a block are randomly permuted by carefully specifying slippage bounds.

Background for attack. The setting of the attack is as follows: suppose there is a particularly large transaction t^* (that hence impacts the exchange rates) swapping X for Y in the list of transactions $\mathcal{T}$ in a block, and suppose the adversary is aware of t^* (maybe due to colluding with the miner of the block). We make two further simplifying assumptions: first, that all other transactions are small and hence have negligible impact on the $X \rightleftharpoons Y$ exchange rates, and second, that the

swap fee f is negligible. Like in the case of the classic block sandwich attack, the adversary can create 2 transactions tx_1 and tx_2 (together with slippage bounds), where tx_1 exchanges some amount of X for Y, and tx_2 exchanges the Y tokens back to X. Unlike in the case of the classic sandwich attack, the adversary has no control over the final order of the transactions in the block as the transactions will be randomly permuted. An advantageous permutation for the adversary would be any permutation such that tx_1 comes before t^* and t^* comes before tx_2 (hereafter we use the notation $a \prec b$ to denote a "comes before" b for two transactions a and b). Permutations that would be disadvantageous to the adversary would be any permutation such that $tx_2 \prec t^* \prec tx_1$, as this is the precise setting where the adversary would lose out due to unfavorable exchange rates. Any other permutations outside of these are acceptable to the adversary.

Utilities. Suppose both tx_1 and tx_2 are executed. We assume the utility of the adversary is $\alpha \in \mathbb{R}^+$ if the resulting permutation is advantageous, $-\alpha$ if the resulting permutation is disadvantageous, and 0 for all other permutations. We assume the utility of the adversary is 0 if both their trades did not execute, as fees are negligible. We further assume the following utilities if only 1 trade executes: if only tx_1 executes, the utility of the adversary is 0 if $t^* \prec tx_1$ and $0 \leq \beta < \alpha$ if $tx_1 \prec t^*$. The intuition behind this is that if tx_1 executes before t^* in this block, there is a chance that when tx_2 executes in the next block the adversary can still benefit from the favorable exchange rates due to advantageous permutation (albeit split over more than 1 block, thus the discount in utility). In the same vein, if only tx_2 executes, the utility of the adversary is 0 if $t^* \prec tx_2$ and $-\gamma < -\alpha$ if $tx_2 \prec t^*$. The reason why $\gamma > \alpha$ is to not only take into account the potential loss to the adversary from the disadvantageous permutation, but also the opportunity cost of waiting more than 1 block for tx_1 to execute.

Here, we denote by s the strategy where the adversary simply wants both transactions to execute and thus does not care about the slippage to be the strategy where the slippage bound for both transactions are set to ∞. It is clear that the expected utility of the adversary under strategy s is 0 due to the fact that both advantageous and disadvantageous permutations occur with equal probability.

Sandwich attack by controlling slippage. We first describe the first attack where the adversary can gain positive expected utility just by being more precise in specifying slippage bounds. The intuition behind this attack is that by specifying the slippage bounds to be extremely precise, the adversary can ensure that transaction tx_1 always executes before t^*, and tx_2 only executes if tx_1 and t^* have executed.

The attack strategy (denoted by s_{slip}) is as follows:

- The adversary computes the current exchange rate of X to Y, denoted by $\rho^{XY}_{tx_0}$, and sets the slippage bound on tx_1 to be $\rho^{XY}_{tx_0} + \varepsilon_1$ for some $\varepsilon_1 > 0$.
- The adversary computes the hypothetical exchange rate of X to Y *after* an execution of t^*. We denote this rate by $\rho^{XY}_{t^*}$. The adversary also computes the hypothetical exchange rate of Y to X *after* an execution of tx_1, t^*, and *both* tx_1 and t^*. We denote these rates by $\rho^{XY}_{tx_1}$, $\rho^{XY}_{t^*}$, and $\rho^{XY}_{tx_1+t^*}$ respectively.
- The adversary sets the slippage bound for tx_2 to be $\rho^{YX}_{t^*} + \varepsilon_2$ for some other $\varepsilon_2 > 0$.

Theorem 49. *If* $\varepsilon_1 < \rho^{XY}_{t^*} - \rho^{XY}_{tx_0}$ *and* $\varepsilon_2 < \min(\rho^{YX}_{tx_1}, \rho^{YX}_{t^*}) - \rho^{YX}_{tx_1+t^*}$, *the expected utility of* μ_{slip} *is* $\frac{\alpha}{6} + \frac{\beta}{3} > 0$.

Proof. Let σ be a random permutation over $\mathcal{T} \cup \{tx_1, tx_2\}$. We denote by $[\sigma_{r,i}]$ the position/index of the ith transaction after the permutation.

We will proceed case by case for each of the 6 different orderings of tx_1, tx_2, t^*.

- $tx_1 \prec t^* \prec tx_2$: both tx_1 and tx_2 would be executed. The utility of the adversary is α in this case.
- $tx_1 \prec tx_2 \prec t^*$: tx_1 will be executed. However, tx_2 will not be executed as $\rho^{YX}_{tx_1+t^*} + \varepsilon_2 < \rho^{YX}_{tx_1}$. Since $tx_1 \prec t^*$, the utility of the adversary is β in this case.
- $t^* \prec tx_1 \prec tx_2$: tx_1 will not be executed as $\rho^{XY}_{tx_0} + \varepsilon_1 < \rho^{XY}_{t^*}$. tx_2 will be executed. Since $t^* \prec tx_2$, the utility of the adversary in this case is 0.
- $t^* \prec tx_2 \prec tx_1$: both tx_1 and tx_2 will not execute as $\rho^{XY}_{tx_0} + \varepsilon_1 < \rho^{XY}_{t^*}$

and $\rho^{YX}_{tx_1+t^*} + \varepsilon_2 < \rho^{YX}_{t^*}$. Since both trades will not execute, the utility of the adversary in this case is 0.

- $tx_2 \prec tx_1 \prec t^*$: tx_1 will execute but tx_2 will not execute as $\rho^{YX}_{tx_1+t^*} + \varepsilon_2 < \rho^{YX}_{tx_0}$. Since $tx_1 \prec t^*$, the utility of the adversary is β in this case.
- $tx_2 \prec t^* \prec tx_1$: both tx_1 and tx_2 will not execute as $\rho^{XY}_{tx_0} + \varepsilon_1 < \rho^{XY}_{t^*}$ and $\rho^{YX}_{tx_1+t^*} + \varepsilon_2 < \rho^{YX}_{tx_0}$. Since both trades will not execute, the utility of the adversary in this case is 0.

Since each ordering is equally likely to occur, the expected utility of the adversary is $\frac{\alpha}{6} + \frac{\beta}{3} > 0$. □

Long-range sandwich attacks. Long-range sandwich attacks are attacks where an adversary aims to front and back run transactions over multiple blocks. This can happen when the adversary mines more than 1 block in a row. However, as the probability of mining more than 1 consecutive block is very small (and grows exponentially smaller in the number of consecutive blocks), the success probability of such an approach is similarly low.

An approach that would lead to a larger probability of success would be for the adversary to create two transactions tx_1 and tx_2 and split tx_1 and tx_2 into separate blocks such that tx_2 only conditionally executes upon tx_1 being on the chain. Formally, instead of adding both tx_1 and tx_2 to the transaction list $\mathcal{T}$ like in the above attack setting, the adversary now only adds tx_1 to $\mathcal{T}$ and waits until tx_1 is on the chain to add tx_2 to the transaction mempool. We note that this can be done by wrapping transactions into smart contracts, which can handle the conditional execution of transactions based on some state of the blockchain. This can also be done in Bitcoin-like blockchains by ensuring that the UTXO of tx_1 is given as input to tx_2.

The attack strategy (denoted by $s_{longslip}$) is as follows:

- The adversary computes the current exchange rate of X to Y, denoted by $\rho^{XY}_{tx_0}$, and sets the slippage bound on tx_1 to be $\rho^{XY}_{tx_0} + \varepsilon_1$ for some $\varepsilon_1 > 0$.
- The adversary waits until tx_1 has been executed (i.e. when the block which contains tx_1 is gossiped), then either wraps tx_2 into

a smart contract that checks if tx_1 is on the blockchain and if so executes tx_2, or ensures that the UTXO of tx_1 is given as input to tx_2.

Recall that if $tx_1 \prec t^*$ in a block and tx_2 executes in some block after the block containing tx_1, the utility of the adversary is $0 \leq \beta < \alpha$, and that the utility of the adversary is 0 if both trades do not occur. We now show that the expected utility under $s_{longslip}$ is also positive.

Theorem 50. *If $\varepsilon_1 < \rho^{XY}_{t^*} - \rho^{XY}_{tx_0}$, the expected utility of $s_{longslip}$ is $\frac{\beta}{2}$.*

Proof. We note that the orderings $tx_1 \prec t^*$ and $t^* \prec tx_1$ are equally likely to occur. If $tx_1 \prec t^*$, tx_1 would be executed together with t^* and thus tx_2 would also be executed in some block after the block containing tx_1. The expected utility of the adversary is β in this case. If $t^* \prec tx_1$, tx_1 would not be executed $\rho^{XY}_{tx_0} + \varepsilon_1 < \rho^{XY}_{t^*}$. Since tx_1 did not execute, tx_2 would not be executed and thus the utility of the adversary in this case is 0. □

Remark 8. We can make the computation of expected utilities more precise by assuming that the utility of the adversary if tx_2 is executed one block after tx_1 is β, and multiply β by δ^d for some discount factor $\delta < 1$ if tx_2 is executed d blocks after tx_1. However, this would require detailed assumptions about the probability of transactions being selected from the mempool which can depend on fees and other factors. This is beyond the scope of our work, thus we leave this as an interesting direction of future work.

7.9 Conclusion

In this chapter we introduced a new construction that can be implemented on top of any blockchain protocol with three main properties. First, the construction does not add any vulnerability to the old protocol, i.e., the security properties remain unchanged. Secondly, performing sandwich attacks in the new protocol is no longer profitable. Thirdly, the construction incurs in minimal overhead with the exception of a minor increase in the latency of the protocol. Our empirical study of sandwich attacks on the Ethereum blockchain also validates the design

principles behind our protocol, demonstrating that our protocol can be easily implemented to mitigate sandwich MEV attacks on the Ethereum blockchain.

Chapter 8

Conclusion

> This is how liberty dies ... with thunderous applause.
>
> Padmé Amidala

From the advent of Nakamoto's groundbreaking Bitcoin protocol, the pursuit of enhanced throughput and reduced latency while upholding principles of security and decentralization has been at the forefront of research and development. The thesis embarked on an exploration of diverse consensus protocols, each striving to overcome limitations inherent in earlier systems. Nakamoto consensus, while revolutionary, faces challenges primarily concerning its throughput. The subsequent development of protocols like GHOST aimed to expand upon Nakamoto consensus by considering off-chain blocks. We have introduced a family of protocols called Medium that allows the study of both Nakamoto consensus and GHOST in a common framework. The Medium protocol can achieve throughputs between the throughput of Nakamoto and GHOST while being robust against the balance attack, a major vulnerability of GHOST.

The analysis of Avalanche introduced a departure from traditional mechanisms based on chains to a metastable approach built on a directed

acyclic graph. Avalanche showcased exceptional throughput and low latency but imposed stricter security constraints. The analysis uncovered vulnerabilities, leading the Glacier modification to fortify the protocol. Moreover, the Avalanche showcased the potential of DAG-based consensus protocols on permissionless settings, emphasizing the performance improvements they offer. This thesis formalized this intuition with a construction transforming a blockchain protocol into a DAG protocol with better throughput, same or lower latency, and preserving the same security guarantees. This construction also allows to determine the set of potentially optimal DAG protocols. We also introduced a new DAG protocol based on the common core that notoriously improves the state-of-the-art latency of DAG protocol. Furthermore, this protocol belongs to the set of possibly optimal protocols.

Addressing critical concerns in blockchain security, this thesis culminated in adapting methods originally designed for the scalability of consensus to prevent sandwich attacks, offering decentralized solutions that mitigate transaction order control within blocks.

In a nutshell, the results encapsulated within this thesis offer a meticulous understanding of blockchain technology's intricacies and hopefully serve as a critical step forward in realizing more efficient, scalable, and secure decentralized systems.

Bibliography

[1] "Distributed randomness beacon," 2023. https://drand.love/.

[2] "Ethereum," 2023. https://ethereum.org/en/whitepaper/.

[3] "MEV over time," 2023. https://explore.flashbots.net/.

[4] "Random ordering of equally-priced transactions incentivises competitive spam," 2023. https://github.com/ethereum/go-ethereum/issues/21350.

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

[6] O. Alpos, I. Amores-Sesar, C. Cachin, and M. Yeo, "Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks," vol. abs/2307.02954, 2023.

[7] I. Amores-Sesar and C. Cachin, "We will DAG you," *CoRR*, vol. abs/2311.03092, 2023.

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

[9] I. Amores-Sesar, C. Cachin, and A. Parker, "Generalizing weighted trees: a bridge from bitcoin to GHOST," in *AFT*, pp. 156–169, ACM, 2021.

[10] I. Amores-Sesar, C. Cachin, and E. Tedeschi, "When is spring coming? A security analysis of avalanche consensus," in *OPODIS*, vol. 253 of *LIPIcs*, pp. 10:1–10:22, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[11] G. Angeris and T. Chitra, "Improved price oracles: Constant function market makers," in *AFT*, pp. 80–91, ACM, 2020.

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

[13] R. J. Aumann, "Acceptable points in general cooperative n-person games," in *Contributions to the Theory of Games (AM-40), Volume IV* (A. W. Tucker and R. D. Luce, eds.), pp. 287–324, Princeton: Princeton University Press, 1959.

[14] Ava Labs, Inc., "Avalanche documentation." https://docs.avax.network/.

[15] Ava Labs, Inc., "Node implementation for the Avalanche network." https://github.com/ava-labs/avalanchego.

[16] V. K. Bagaria, S. Kannan, D. Tse, G. C. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *CCS*, pp. 585–602, ACM, 2019.

[17] L. Baird and A. Luykx, "The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers," in *COINS*, pp. 1–7, IEEE, 2020.

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

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

[20] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *CCS*, pp. 62–73, ACM, 1993.

[21] B. Bernheim, B. Peleg, and M. D. Whinston, "Coalition-proof nash equilibria i. concepts," vol. 42, pp. 1–12, 1987.

[22] M. Blum, "Coin flipping by telephone a protocol for solving impossible problems," vol. 15, (New York, NY, USA), p. 23–27, Association for Computing Machinery, jan 1983.

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

[24] M. Bowman, D. Das, A. Mandal, and H. Montgomery, "On elapsed time consensus protocols," in *INDOCRYPT*, vol. 13143 of *Lecture Notes in Computer Science*, pp. 559–583, Springer, 2021.

[25] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, 1985.

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

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

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

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

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

[31] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptol.*, vol. 13, no. 1, pp. 143–202, 2000.

[32] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *STOC*, pp. 42–51, ACM, 1993.

[33] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, pp. 173–186, USENIX Association, 1999.

[34] Chainlink Labs, "Chainlink 2.0: Next steps in the evolution of decentralized oracle networks." Whitepaper, 2021. https://research.chain.link/whitepaper-v2.pdf.

[35] T. H. Chan, N. Ephraim, A. Marcedone, A. Morgan, R. Pass, and E. Shi, "Blockchain with varying number of players," *IACR Cryptol. ePrint Arch.*, p. 677, 2020.

[36] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, "Probabilistic smart contracts: Secure randomness on the blockchain," in *IEEE ICBC*, pp. 403–412, IEEE, 2019.

[37] T. Chitra, "Towards a theory of maximal extractable value II: uncertainty," *CoRR*, vol. abs/2309.14201, 2023.

[38] K. Choi, A. Arun, N. Tyagi, and J. Bonneau, "Bicorn: An optimistically efficient distributed randomness beacon," *IACR Cryptol. ePrint Arch.*, p. 221, 2023.

[39] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract)," in *FOCS*, pp. 383–395, IEEE Computer Society, 1985.

[40] B. Cohen and K. Pietrzak, "The chia network blockchain." Whitepaper, 2019. https://docs.chia.net/assets/files/Precursor-ChiaGreenPaper-82cb50060c575f3f71444a4b7430fb9d.pdf.

[41] "Coinmarketcap: Today's cryptocurrency prices by market cap." https://coinmarketcap.com/, 2022.

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[43] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains - (A position paper)," in *Financial Cryptography Workshops*, vol. 9604 of *Lecture Notes in Computer Science*, pp. 106–125, Springer, 2016.

[44] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *IEEE Symposium on Security and Privacy*, pp. 910–927, IEEE, 2020.

[45] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient BFT consensus," in *EuroSys*, pp. 34–50, ACM, 2022.

[46] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT (2)*, vol. 10821 of *Lecture Notes in Computer Science*, pp. 66–98, Springer, 2018.

[47] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO*, vol. 435 of *Lecture Notes in Computer Science*, pp. 307–315, Springer, 1989.

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

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

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

[51] G. Eisenstein, "Über die Irreductibilität und einige andere Eigenschaften der Gleichung, von welcher die Theilung der ganzen Lemniscate abhängt," vol. 39, pp. 160–179, 1850.

[52] Encyclopedia Britannica, "Topic: Medium (occultism)," 2021. https://www.britannica.com/topic/medium-occultism.

[53] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse, "Bitcoining: A scalable blockchain protocol," in *NSDI*, pp. 45–59, USENIX Association, 2016.

[54] I. Eyal and E. G. Sirer, "Majority is not enough: bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[55] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[56] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *EUROCRYPT (2)*, vol. 9057 of *Lecture Notes in Computer Science*, pp. 281–310, Springer, 2015.

[57] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol with chains of variable difficulty," in *CRYPTO (1)*, vol. 10401 of *Lecture Notes in Computer Science*, pp. 291–323, Springer, 2017.

[58] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *SOSP*, pp. 51–68, ACM, 2017.

[59] N. Halidias, "On the absorption probabilities and mean time for absorption for discrete markov chains," *Monte Carlo Methods Appl.*, vol. 27, no. 2, pp. 105–115, 2021.

[60] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *USENIX Security Symposium*, pp. 129–144, USENIX Association, 2015.

[61] L. Heimbach and R. Wattenhofer, "Eliminating sandwich attacks with the help of game theory," in *AsiaCCS*, pp. 153–167, ACM, 2022.

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

[63] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *PODC*, pp. 165–175, ACM, 2021.

[64] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, "Cordial miners: Fast and efficient consensus for every eventuality," in *DISC*, vol. 281 of *LIPIcs*, pp. 26:1–26:22, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

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

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

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

[68] J. Kelsey, L. T. A. N. Brandão, R. Peralta, and H. Booth, "Nistir 8213. a reference for randomness beacons: Format and protocol version 2," tech. rep., 2011.

[69] A. Kiayias and G. Panagiotakos, "On trees, chains and fast transactions in the blockchain," in *LATINCRYPT*, vol. 11368 of *Lecture Notes in Computer Science*, pp. 327–351, Springer, 2017.

[70] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO (1)*, vol. 10401 of *Lecture Notes in Computer Science*, pp. 357–388, Springer, 2017.

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

[72] L. Kiffer, R. Rajaraman, and A. Shelat, "A better method to analyze blockchain consistency," *IACR Cryptol. ePrint Arch.*, p. 601, 2022.

[73] M. Kim, Y. Kwon, and Y. Kim, "Is stellar as secure as you think?," in *EuroS&P Workshops*, pp. 377–385, IEEE, 2019.

[74] G. Kol and M. Naor, "Games for exchanging information," in *STOC*, pp. 423–432, ACM, 2008.

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

[76] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[77] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[78] A. K. Lenstra and B. Wesolowski, "Trustworthy public randomness with sloth, unicorn, and trx," *Int. J. Appl. Cryptogr.*, vol. 3, no. 4, pp. 330–343, 2017.

[79] C. Li, P. Li, W. Xu, F. Long, and A. C. Yao, "Scaling nakamoto consensus to thousands of transactions per second," *CoRR*, vol. abs/1805.03870, 2018.

[80] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C. Yao, "A decentralized blockchain with high throughput and fast confirmation," in *USENIX Annual Technical Conference*, pp. 515–528, USENIX Association, 2020.

[81] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with stellar," in *SOSP*, pp. 80–96, ACM, 2019.

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

[83] H. N. Mamache, G. Mazué, O. Rashid, G. Bu, and M. Potop-Butucaru, "Resilience of IOTA consensus," in *ICC*, pp. 5694–5699, IEEE, 2022.

[84] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: High-throughput atomic broadcast," *Comput. J.*, vol. 60, no. 6, pp. 866–882, 2017.

[85] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, and A. Shakiba, "Optimal mining: Maximizing bitcoin miners' revenues from transaction fees," in *Blockchain*, pp. 266–273, IEEE, 2022.

[86] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *CCS*, pp. 31–42, ACM, 2016.

[87] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[88] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." Whitepaper, https://bitcoin.org/bitcoin.pdf, 2009.

[89] C. Natoli and V. Gramoli, "The balance attack or why forkable blockchains are ill-suited for consortium," in *DSN*, pp. 579–590, IEEE Computer Society, 2017.

[90] M. J. Osborne and A. Rubinstein, *A course in game theory*. Cambridge, USA: The MIT Press, 1994. electronic edition.

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

[92] T. P. Pedersen, "Cps, certificate practice statement," in *Encyclopedia of Cryptography and Security* (H. C. A. van Tilborg, ed.), Springer, 2005.

[93] F. Pedone and A. Schiper, "Generic broadcast," in *DISC*, vol. 1693 of *Lecture Notes in Computer Science*, pp. 94–108, Springer, 1999.

[94] A. Penzkofer, B. Kusmierz, A. Capossele, W. Sanders, and O. Saa, "Parasite chain detection in the IOTA protocol," in *Tokenomics*, vol. 82 of *OASIcs*, pp. 8:1–8:18, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[95] S. Popov, O. Saa, and P. Finardi, "Equilibria in the tangle," *Comput. Ind. Eng.*, vol. 136, pp. 160–172, 2019.

[96] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?," in *IEEE Symposium on Security and Privacy*, pp. 198–214, IEEE, 2022.

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

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

[99] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless BFT consensus through metastability," *CoRR*, vol. abs/1906.08936, 2019.

[100] S. Ross, *Stochastic Processes*. Wiley, second ed., 1996.

[101] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *ICDCN*, vol. 7129 of *Lecture Notes in Computer Science*, pp. 153–167, Springer, 2012.

[102] N. I. Schwartzbach, "Deposit schemes for incentivizing behavior in finite games of perfect information," *CoRR*, vol. abs/2107.08748, 2021.

[103] Y. Sompolinsky, S. Wyborski, and A. Zohar, "PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021," in *AFT*, pp. 57–70, ACM, 2021.

[104] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *Financial Cryptography*, vol. 8975 of *Lecture Notes in Computer Science*, pp. 507–527, Springer, 2015.

[105] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *CCS*, pp. 2705–2718, ACM, 2022.

[106] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *IEEE Symposium on Security and Privacy*, pp. 444–460, IEEE Computer Society, 2017.

[107] B. Wang, Q. Wang, S. Chen, and Y. Xiang, "Security analysis on tangle-based blockchain through simulation," vol. abs/2008.04863, 2020.

[108] Q. Wang, J. Yu, Z. Peng, V. C. Bui, S. Chen, Y. Ding, and Y. Xiang, "Security analysis on dbft protocol of NEO," in *Financial Cryptography*, vol. 12059 of *Lecture Notes in Computer Science*, pp. 20–31, Springer, 2020.

[109] A. Yanai, "Blinderswap: MEV meets MPC." https://www.youtube.com/watch?v=KQ4xK79YkFE&ab_channel=IC3InitiativeforCryptocurrenciesandContracts, 2021. Accessed 03/08/23.

[110] M. Yin, *Scaling the Infrastructure of Practical Blockchain Systems*. PhD thesis, Cornell University, USA, 2021.

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

[112] H. Zhang, L. Merino, Z. Qu, M. Bastankhah, V. Estrada-Galiñanes, and B. Ford, "F3B: A low-overhead blockchain architecture with per-transaction front-running protection," in *AFT*, vol. 282 of *LIPIcs*, pp. 3:1–3:23, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.