Scaling permissionless blockchains via sharding

A Thesis submitted in fulfilment of the requirements for

the candidature of

Doctor of Philosophy

By

Runchao Han

Supervisor: Jiangshan Yu

Co-supervisor: Joseph Liu and Shiping Chen

Department of Software Systems and Cybersecurity

Faculty of Information Technology

Monash University

August 29, 2022

Abstract

Sharding is a promising approach to scale permissionless blockchains. In a blockchain sharding protocol, participants are divided into different shards and each shard processes transactions concurrently. Despite its wide adoption in permissioned systems, transferring such success to permissionless blockchains is still an open problem. To the best of our knowledge, no blockchain sharding protocol has been proven secure or practical, and formal security analysis on blockchain sharding protocols and their primitives is still missing.

This thesis bridges this gap by systematically studying blockchain sharding protocols and their primitives. We formalise blockchain sharding protocols and evaluate existing proposals, which reveals multiple security concerns and design trade-offs overlooked by existing research. Most notably, we identify two primitives that are necessary for blockchain sharding protocols but are overlooked by existing studies, namely *shard allocation* and *decentralised randomness beacon*. We formalise these two primitives, and propose constructions that are proven secure and more efficient. Our proposals can serve as drop-in replacements in any blockchain sharding protocols, and can be of independent interest. In addition, we identify two security issues that exist in sharded blockchains and all non-sharded proof-of-work-based blockchains, namely the cross-chain 51% attacks and the optionality of the Atomic Swap protocol. We formally study their impact on the related security properties and suggest countermeasures against them.

All of the results are supported by formal security proofs and experimental evaluations.

Copyright notice

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: [illegible]

Print Name: Runchao Han

Date: August 29, 2022

List of publications

The following papers containing contents in this thesis have been published.

- Runchao Han, Haoyu Lin, and Jiangshan Yu. "RandChain: A Scalable and Fair Decentralised Randomness Beacon. The 4th ACM Conference on Advances in Financial Technologies (AFT 2022)."
  (Full version at https://eprint.iacr.org/2020/1033)

- Runchao Han, Zhimei Sui, Jiangshan Yu, Joseph Liu, and Shiping Chen. "Fact and Fiction: Challenging the honest majority assumption of permissionless blockchains." The 16th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2021).
  (Full version at https://eprint.iacr.org/2019/752)

- Runchao Han, Haoyu Lin, and Jiangshan Yu. "On the optionality and fairness of Atomic Swaps." The First ACM Conference on Advances in Financial Technologies (AFT 2019).
  (Full version at https://eprint.iacr.org/2019/896)

The following papers containing contents in this thesis are in submission.

- Runchao Han and Jiangshan Yu. "Fair delivery of decentralised randomness beacons."

- Runchao Han, Jiangshan Yu, and Ren Zhang. "Analysing and Improving Shard Allocation Protocols for Sharded Blockchains."
  (Full version at https://eprint.iacr.org/2020/943)

- Runchao Han, Jiangshan Yu, Haoyu Lin, Shiping Chen, and Paulo Esteves-Veríssimo. "On the Security and Performance of Blockchain

Sharding."
(Full version at https://eprint.iacr.org/2021/1276)

I am also fortunate to collaborate with other blockchain researchers. The following papers are beyond the scope of this thesis.

- Runchao Han, Haoyu Lin, and Jiangshan Yu. "VRF-Based Mining: Simple Non-Outsourceable Cryptocurrency Mining." International Workshop on Cryptocurrencies and Blockchain Technology (CBT@ESORICS 2020).
  (Full version at https://github.com/DEX-ware/vrf-mining/blob/master/paper/main.pdf)

- Zhichun Lu, Runchao Han, and Jiangshan Yu. "General Congestion Attack on HTLC-Based Payment Channel Networks." 3rd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2021)
  (Full version at https://eprint.iacr.org/2020/456)

Acknowledgements

I would like to thank Monash University and CSIRO's Data61 for providing scholarships for my PhD. I also would like to thank The Department of Software Systems and Cybersecurity, Faculty of Information Technology, Monash University, for the additional support, especially during the COVID outbreak.

I am grateful to my supervisor Jiangshan Yu, Joseph Liu and Shiping Chen for guiding me towards research and providing me full academic freedom to explore different areas. The experience at Monash and CSIRO's Data61 was wonderful and is key to achieve my goals. In addition, I am also grateful to Vincent Gramoli, Haiwu He, and Christos Kotselidis for the trust and the initial guidance when I had no research or professional experience.

I am fortunate to work with many great researchers and engineers during my PhD, including (chronologically) Nikos Foutris, Gary Shapiro, Haoyu Lin, Ren Zhang, Zhichun Lu, Carsten Rudolph, Marthie Grobler, James Boorman, Paulo Verissimo, and Jianyu Niu. I learned a lot from them and would like to thank them for their contributions.

My PhD journey would be much less enjoyable without the friendship from many colleagues at Monash University and CSIRO's Data61, including Maxime Buser, Ahmad Salehi Shahraki, Qin Wang, Dimaz Wijaya, Zhimei Sui, Viet Vo, and everyone that I cannot have chance to meet often due to the COVID.

Finally, I want to thank my partner Xin Lin and my family for the constant support. This thesis would not be possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Blockchain, first introduced by Bitcoin [1], has gained significant attention in the recent years. It allows a set of networked participants to jointly maintain a tamper-resistant and censorship-resistant ledger, even when a subset of them are compromised (aka Byzantine). Its tamper- and censorship-resistance properties enables and reinforces numerous applications in different industries, such as finance, supply chain management, and Internet of Things (IoT).

A concrete example of blockchain-based applications would be currency exchange. Consider two parties want to trade their currencies at a certain exchange rate. In traditional centralised exchanges, they have to assume the central server to be uncompromised in order to complete the trade. Otherwise, if the central server is compromised, it will be possible that one party transfers its money while the other party does not. Meanwhile, in blockchains where all valid transactions will be eventually executed, such trades can be made always successful, by using various on-blockchain protocols such as Atomic Swap and Automated Market Maker.

There are different types of blockchains in terms of the membership, such as permissioned blockchains and permissionless blockchains. While permissioned blockchains specify a set of known participants, permissionless blockchains allow anyone on the Internet to join the system. For example, Bitcoin [1] is the first permissionless blockchain. Compared to permissioned blockchains, permissionless blockchains relax the trust assumption on memberships and are open to anyone, thus are more trusted and widely adopted by the community. To date, there have been more than 20,000 blockchains that constitute 1 trillion USD market cap in total [2].

However, permissionless blockchains suffer from a fundamental *blockchain trilemma* that obstacles their mass adoption. Informally, the blockchain trilemma states that any blockchain can achieve at most two out of the following three properties:

- **Decentralisation:** Each participant in the blockchain only has access to a constant amount of resource in computation and communication.
- **Scalability:** The blockchain can process an increasing number transactions per time unit with more participants.
- **Security:** The blockchain can achieve all necessary security properties even when a predefined portion of participants are malicious.

Each of the three properties is important for the mass adoption of permissionless blockchains. Without decentralisation, a blockchain will be controlled by a small set of participants, making it less trustworthy. Without scalability, a blockchain can only process limited transactions per time unit, limiting the number of concurrent users. Without security, transactions in a blockchain will no longer be trustworthy.

There have been numerous permissionless blockchain proposals that make different trade-offs over the three properties. A promising approach among these proposals is *sharding*. It aims at achieving scalability and decentralisation while slightly compromising the security level to a reasonable level. In a sharded blockchain, as depicted in Figure 1.1, participants are divided into different shards, each of which maintains its own ledger and processes transactions concurrently. Therefore, the sharded blockchain can process more concurrent transactions with increasing number of shards, leading to better scalability. In addition, a node only needs to store the trasnactions in a single shard rather than all transactions, leading to better decentralisation.

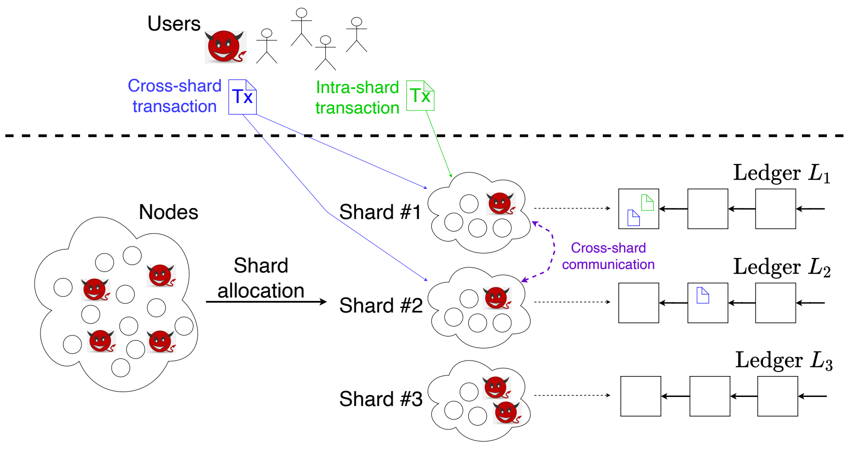


Figure 1.1: An example sharded blockchain with three shards. A set of nodes (of which a subset are Byzantine and marked as red) are divided into three shards. Nodes in each shard maintain their own ledger. Users (of which a subset are also Byzantine and marked as red) submit transactions to shards, including intra-shard transactions (in green) and cross-shard trasnactions (in blue) that are related to multiple shards.

Several academic proposals [3]–[7] and industry projects [8], [9] on permissionless sharded blockchains have been proposed. Yet, it still remains unknown whether existing blockchain sharding protocols are secure or practical. Existing proposals either lack formal security analysis [7]–[9], or only provide analysis on some primitives such as distributed randomness generation [3], [4], shard allocation [5], and cross-shard transactions [4], [6]. Existing papers on analysing these proposals either merely summarise designs [10] or focus on their certain primitives such as shard allocation [11] and cross-shard transactions [12]. Therefore, a systematic security analysis of permissionless sharded blockchains is still missing, and building a sharded blockchain that is secure and practical remains an open challenge.

1.1 Contributions

To fill this gap, this thesis analyses and improves permissionless blockchain sharding protocols and their underlying primitvies. We start from formalising blockchain sharding protocols (Chapter 2) and evaluating existing proposals (Chapter 3), which reveals multiple security concerns and design trade-offs. Through our evaluation, we identify two primitives that are implicitly and informally studied by existing research, namely *shard allocation* and *decentralised randomness beacon* (DRB). The shard allocation protocol allows nodes to be allocated into different shards securely, randomly and dynamically w.r.t. the node churn. DRB allows nodes to generate unpredictable and unbiasible random outputs periodically. We formalise these two primitives, evaluate existing proposals, and propose constructions that are proven secure and more efficient (Chapter 4-5). We also identify a new security property *delivery-fairness* overlooked by existing research in DRB. We provide the first formal definition to delivery-fairness, prove its lower bound, and suggest modifications on existing DRB protocols to achieve optimal delivery-fairness (Chapter 6). In addition, we identify two security issues that are critical for both sharded blockchains and non-sharded blockchains, namely the 51% attacks with external mining power and the optionality of Atomic Swap protocols. We formalise the two issues, quantify their impacts on the related security properties, and suggest countermeasures against them (Chapter 7-8).

Systematic evaluation of blockchain sharding protocols. To evaluate sharded blockchains, we deconstruct the blockchain sharding problem into four foundational layers with orthogonal functionalities. The four layers include *data layer* that defines how the ledger is formatted and divided into different shards; *membership layer* that defines how nodes are allocated to different shards; *intra-shard layer* that defines how each shard processes local transactions; and *cross-shard layer* that defines how shards process cross-shard transactions. For each protocol layer, we suggest and analyse the required security properties and performance metrics for sharded blockchains w.r.t. their design objectives and possible attacks. Of independent interest, this can serve as an evaluation framework assisting the future development for sharded blockchains.

We select seven state-of-the-art sharded blockchains, deconstruct them into the four layers, and evaluate the layer according to our definitions. The evaluated sharded blockchains include five academic proposals (Elastico [3], Omniledger [4], Chainspace [6], RapidChain [5], and Monoxide [7]) and two industrial projects (Zilliqa [8] and Ethereum 2.0 [9]). We perform the layer-wise evaluation, then perform the system-level evaluation for the selected sharded blockchains. The evaluation shows that these protocols fail to satisfy certain correctness properties, or meet them at the price of strong assumptions or at least $O(n^2)$ communication complexity. Along the way, we identify a number of new attacks, overlooked primitives and design trade-offs in blockchain sharding protocols.

Two key primitives: Shard allocation and decentralised randomness beacon. Based on the systematic analysis, we identify two primitives that are necessary for building a sharded blockchain but are overlooked by existing research. The two identified primitives are *shard allocation* and *decentralised randomness beacon (DRB)*. Shard allocation allows nodes to be allocated into different shards securely and dynamically w.r.t. the node churn. DRB allows nodes to produce unpredictable and unbiasible random outputs periodically. We formally study them and propose constructions that are proven secure and more efficient. These constructions serve as drop-in replacements to existing blockchain sharding protocols.

For shard allocation, we provide the first formal study on this primitive, and evaluate the shard allocation protocols in the seven permissionless sharded blockchains above. Our results show that *none of them is fully correct or achieves satisfactory performance*. Based on the evaluation, we observe and prove a fundamental security-performance trade-off in blockchain sharding protocols, then identify and define a new property *memory-dependency* that is necessary for shard allocation protocols to parameterise this trade-off. These insights allow us to construct WORMHOLE, a secure and efficient shard allocation protocol that is the first to allow parameterisation over this trade-off.

For DRB, we observe that existing approaches cannot scale due to a fundamental design: Participants contribute their local inputs and aggregate them into a single output. In order to collaborate, participants should continuously broadcast messages to and synchronise with each other. The former incurs at least $O(n^2)$ communication complexity, and the latter requires round synchronisation. To address the inherent limitations in the collaborative design, we suggest a new design space for DRBs called *competitive DRBs*, and propose RANDCHAIN, a scalable decentralised randomness beacon protocol that belongs to this design space. The central building block of the RANDCHAIN protocol is a new primitive *Sequential Proof-of-Work (SeqPoW)*, a cryptographic puzzle that takes a random and unpredictable number of sequential steps to solve. SeqPoW is also of independent interest in other protocols such as leader election and Proof-of-Stake (PoS)-based consensus.

We also observe a new security property in DRBs called *delivery-fairness*, which is overlooked by existing literatures. Delivery-fairness concerns the advantage that some participants learn random outputs earlier than other participants in DRBs. We formalise the delivery-fairness property, prove lower bound results of them in system settings aligned to the real-world environment, and suggest modifications over existing DRB protocols to achieve optimal delivery-fairness.

Two cross-chain security issues. We also study two security issues that exist in sharded blockchains and all non-sharded PoW-based blockchains, namely the cross-chain 51% attacks and the optionality of the Atomic Swap protocol. We formalise these two issues, quantify their impact on the related security properties, and suggest countermeasures against them.

In the 51% attack in PoW-based blockchains, the adversary mines a blockchain longer than the honest blockchain secretly, and later publishes its blockchain to revert the honest blockchain. The 51% attack can utilise external mining power from other blockchains or cloud mining services such as NiceHash [13]. The adversary may obtain more profits by launching such cross-chain 51% attacks compared to using such mining power to mine a blockchain honestly. We study two cases of the cross-chain 51% attacks, namely the *mining power migration attack* and the *cloud mining attack*. In the mining power migration attack, the adversary migrates mining power from a stronger blockchain containing more mining power to attack a weaker blockchain containing less mining power. In the cloud mining attack [14], the adversary rents mining power from cloud mining services (e.g., Nicehash[13]) to attack a blockchain. We formally analyse two variants of 51% attacks using externally available mining power, and show that 51% attacks are feasible and more profitable than honestly mining for most blockchains. Our analysis implies two results. First, the honest majority assumption does not hold for these blockchains. Second, instead of encouraging miners to mine honestly, the incentive mechanism encourages miners to launch 51% attacks and break "honest majority".

Atomic Swap [15] is a protocol where two parties on two blockchains atomically exchange their cryptocurrencies without trusted third party. It is known to have the *optionality*: The swap initiator can choose to proceed or abort the swap before the timelock (with the default value of 24 hours) expires. This allows the adversary, as the swap initiator, to speculate without any penalty: When the timelock is about to expire, if the price of the swap participant's asset rises, the swap initiator will proceed the swap so that he will profit; and if the price of the swap participant's asset drops, the swap initiator can abort the swap, so that he will not lose money. We show that an atomic is equivalent to a premium-free American Call Option, and quantify the unfairness of the Atomic Swap by using the Cox-Ross-Rubinstein option pricing model [16]. Our results show that, in the default setting, the premium should be approximately 2% for Atomic Swaps with cryptocurrency pairs, while the premium is approximately 0.3% for American Call Options with stocks and fiat currencies. We then suggest improvements to implement the premium mechanism in the Atomic Swap protocol to make it fair.

Chapter 2

Background and Model

In this chapter, we formally define the blockchain protocol and the blockchain sharding protocol.

2.1 System setting

We start from specifying the system setting of blockchain protocols. The system setting of a sharded blockchain concerns two aspects, namely network synchrony and fault tolerance degree.

Network synchrony concerns the timing guarantees of message deliveries. We consider three network models: *Synchrony* where messages are delivered within a known finite time bound; *partial synchrony* [17] where messages are delivered within a known finite time bound plus an unknown global stabilisation time (GST); and *asynchrony* where messages are delivered eventually but without a known time bound.

Fault tolerance degree concerns the resilience of the protocol, in terms of the level of threat it can cope with while remaining correct. Given the permissionless settings, we consider Byzantine faults. The fault tolerance degree is quantified as the least percentage of *voting power* that the Byzantine adversary should control to break the protocol. Different protocols quantify *voting power* using different metrics, such as computing power in PoW-based consensus and deposited cryptocurrency in PoS-based consensus.

2.2 System components

The blockchain protocol contains three components, namely nodes, users and the ledger.

Nodes are participants who jointly maintain the ledger for the system. The system is permissionless: Anyone can participate in the system as a node, and nodes can join and leave the system at any time. Each node p_i has a pair of secret key and public key (sk_i, pk_i), and is identified by its public key pk_i in the system.

Users are clients that create transactions and send them to nodes. Nodes verify incoming transactions continuously. If an incoming transaction is valid, then the node moves it to its *memory pool*, i.e., the set of pending transactions. For each epoch, nodes in each shard sample some transactions from their memory pools to agree on.

Ledger is the collection of system states jointly maintained by nodes. We model the ledger following the approach of Chainspace [6]. Specifically, let "object" be the irreducible unit of system states. The ledger consists of a number of objects, or a number of transactions recording changes of objects. Each object is owned by a user, and has a unique identifier and a value (i.e., the amount of coins). Each transaction consists of input objects and output objects, plus some transaction fee. Each object can only be the output of a single transaction. An object is inactive if the current ledger includes a transaction with this object as input. An object is active if the current ledger includes a transaction creating this object and no transaction takes this object as input. Note that this model captures both the UTXO model (e.g., in Bitcoin) where each active object is an unspent transaction output (UTXO) and the account model (e.g., in Ethereum) where each account controls a number of objects and a transaction modifies multiple objects atomically.

2.3 Blockchain protocol

We formally define blockchains based on well-established models [18]. In a blockchain, nodes jointly maintain a ledger formed as a blockchain, i.e., a chain of blocks. Each block records a number of transactions. Each transaction records transition of some states. To abstract the process of verifying transactions, we define a transaction verification oracle $V(tx, L, \ell)$ that, given transaction tx, ledger L and height ℓ, outputs 1 if tx is a valid transaction at height ℓ of ledger L, otherwise 0. The transaction format and definition of $V(\cdot)$ depends on the data layer design, which we will analyse in §3.4.

Same as distributed ledger protocols [18], a blockchain has to satisfy *persistence* and *Liveness*. To distinguish properties between a non-sharded blockchain and a sharded blockchain, we label persistence and liveness of a non-sharded blockchain as chain-persistence and chain-liveness, respectively. Chain-Persistence formally states the tamper-resistance of blockchains. It specifies that if a valid transaction tx becomes d-deep (i.e., is followed by d consecutive blocks) in the blockchain of a correct node, then it will be "stable": All correct nodes will include tx in the same position of their blockchains, and the adversary cannot revert tx in their blockchains.

Definition 2.3.1 (d-Chain-Persistence)**.** A blockchain satisfies d-Chain-Persistence if the following holds. If a correct node in the blockchain includes a valid transaction tx at height ℓ on its local ledger which is at least $(\ell + d)$-long, then any correct node in the blockchain includes tx at the same position as the node's ledger.

Chain-Liveness formally states the censorship-resistance of blockchains. It specifies that if a valid transaction tx is submitted to correct nodes for a certain time range of generating u new blocks, then tx will eventually be stable (i.e., become d-deep in blockchains of all correct nodes), and the adversary cannot censor tx, i.e., preventing tx from being included.

Definition 2.3.2 ((u, d)-Chain-Liveness)**.** A blockchain satisfies (u, d)-Chain-Liveness if the following holds. If a valid transaction tx is given as input to all correct nodes in a blockchain for u consecutive blocks, then all correct nodes in the blockchain will include tx in a block that is at least d-deep on its local ledger.

2.4 Sharded blockchain

Sharded blockchain aims at scaling the blockchain by maintaining multiple blockchains in parallel. In a sharded blockchain, nodes are partitioned into a fixed number of m *shards*, and each node only belongs to a single shard. Each shard maintains its own ledger: Nodes in each shard execute consensus to agree on some new transactions; pack agreed transactions into a block; and append the block to the ledger.

A sharded blockchain has to satisfy three properties, namely persistence, liveness, and validity. First, a sharded blockchain satisfies persistence and liveness if all shards in the sharded blockchain satisfy persistence and liveness, respectively.

Definition 2.4.1 (d-Persistence)**.** A sharded blockchain satisfies d-Persistence iff all shards satisfy d-Chain-Persistence.

Definition 2.4.2 ((u, d)-Liveness)**.** A sharded blockchain satisfies (u, d)-Liveness iff all shards satisfy (u, d)-Chain-Liveness.

In addition, a sharded blockchain has to satisfy validity, meaning that no transaction in the ledger is conflicted with each other. There are two types of transactions, namely *intra-shard* transactions whose input and output objects belong to a single shard and *cross-shard transactions* whose input and output objects belong to different shards. Nodes in different shards need to communicate with each other to process cross-shard transactions, in order to guarantee that cross-shard transactions are not conflicted with each other.

To abstract the process of resolving conflicts introduced by cross-shard transactions, we define oracle $C(tx_x, tx_y)$ that, given two transactions tx_x

and tx_y, outputs 1 if tx_x and tx_y are conflicted, otherwise 0. The definition of transaction conflicts and the specification of $C(\cdot)$ depend on the data layer design, which we will analyse in §3.4. Cross-shard communication may be required in $C(\cdot)$.

Definition 2.4.3 (Validity)**.** For any two correct nodes from any two shards k_x and k_y ($x = y$ is possible), if they include transaction tx_x and tx_y in the blocks at height ℓ_x and ℓ_y in their local ledgers L_x and L_y, respectively, then the following holds:

- $V(tx_x, L_x, \ell_x) = 1$,
- $V(tx_y, L_y, \ell_y) = 1$, and
- $C(tx_x, tx_y) = 0$.

Chapter 3

Evaluation of sharded blockchains

3.1 Introduction

In this chapter, we perform a systematic evaluation of permissionless sharded blockchains based on the formalisation in Chapter 2. Our evaluation identifies a considerable number of new attacks, design trade-offs and open challenges. Most notably, we identify an important design choice that is overlooked by existing sharded blockchains, namely the coherence of system settings across layers. Our study will help designers and developers to better understand the sharding system and design principles and pitfalls, assisting the future development of secure and practical sharding techniques for permissionless blockchains.

Systematic evaluation. To evaluate sharded blockchains, we deconstruct the blockchain sharding problem into four foundational layers with orthogonal functionalities. The four layers, summarised in §3.2, include *data layer* that defines how the ledger is formatted and divided into different shards; *membership layer* that defines how nodes are allocated to different shards; *intra-shard layer* that defines how each shard processes local transactions; and *cross-shard layer* that defines how shards process cross-shard transactions. The functionality of data layer is captured by a set of verification rules; the functionality of membership layer is captured by *shard allocation*, which will be formally studied in Chapter 4; the functionality of intra-shard layer is captured by *leader election* [19] and *consensus*; and the functionality of cross-shard layer is captured by *Concurrency Control* [20] and *Atomic Commit* [21]. For each protocol layer, we suggest and analyse the required security properties and performance metrics for sharded blockchains w.r.t.

Table 3.1: Evaluation of existing blockchain sharding proposals.

	System setting		Correctness			Performance
	Net. sync.	Fault tolerance	Persistance	Liveness	Validity	Comm. compl.
Elastico	Sync.	0	✓	✓	✓	$O(n^2)$
Omniledger	Sync.	1/3	✓	✗	✗	$O(n^2)$
RapidChain	Sync.	0†	✓	✓	✗	$O(n^2)$
Chainspace	Part. Sync.	1/3	✓	✓	✗	$O(n^2)$
Monoxide	Sync.	1/2	✓	✓	✗	$O(n)$
Zilliqa	Sync.	1/3	✓	✓	✓	$O(n^2)$
Ethereum 2.0	Sync.	1/3	✓	✗	✓	$O(n)$

† RapidChain's shard allocation protocol cannot tolerate any fault, which will be analysed in Chapter 4.

their design objectives and possible attacks. Of independent interest, this can serve as an evaluation framework assisting the future development for sharded blockchains.

We select seven state-of-the-art sharded blockchains, deconstruct them into the four layers, and evaluate the layer according to our definitions. The evaluated sharded blockchains, summarised in §3.3, include five academic proposals (Elastico [3], Omniledger [4], Chainspace [6], RapidChain [5], and Monoxide [7]) and two industrial projects (Zilliqa [8] and Ethereum 2.0 [9]).

Evaluation results. We perform the layer-wise evaluation in §3.4-3.6, then perform the system-level evaluation in §3.7 for the selected sharded blockchains. The evaluation, summarised in Table 3.1, shows that these protocols fail to satisfy certain correctness properties, or meet them at the price of strong assumptions or at least $O(n^2)$ communication complexity. Most notably, a single Byzantine node can stall the leader election protocol (Table 3.4a) and the shard allocation protocol [11], breaking the liveness of Elastico and RapidChain, respectively. In addition, a single Byzantine user can create conflicted views of cross-shard transactions for different shards, breaking the validity of Omniledger, RapidChain and Monoxide (Table 3.6).

Along the way, we identify a number of new attacks, overlooked subprotocols and design trade-offs in blockchain sharding protocols. For the data layer (§3.4), we identify five design choices and analyse two design trade-offs parameterised by how the ledger is partitioned among shards and how transactions are ordered. For the intra-shard layer (§3.5), we show that existing proposals overlook the leader election protocol, which in fact is challenging to design and relies on strong assumptions. For the cross-shard layer (§3.6), we relate the distributed transaction problem with the cross-shard communication problem, which consists of two protocols, namely Concurrency Control [22]–[24] and Atomic Commit [25]. We narrow down the design space of Concurrency Control for sharded blockchains and analyse the trade-offs in different designs. We identify three new attacks on Atomic Commit in sharded blockchains, and show that Non-Blocking Atomic Commit (NB-AC) [25], a variant of Atomic Commit, is necessary to resist these attacks. Our evaluation in the NB-AC model shows that cross-shard communication requires either a centralised coordinator or shared information among shards, which is consistent with a recent work [12]. We also analyse the trade-off between communication overhead and *timing of consistency* [26] for cross-shard communication.

Insights on the coherence of system settings. Based on the system-level evaluation, we identify an overlooked design choice that greatly affects a sharded blockchain's security and performance, namely *the coherence of system settings across layers*. All evaluated sharded blockchains assume different system settings for different protocol layers, without corresponding architectural guarantees. Unless a hybrid architecture were used, assuming different system settings across layers will weaken security and/or reduce performance [27]. We show that for each evaluated sharded blockchain, replacing subprotocols to those with consistent assumptions will improve its security and/or performance.

Table 3.2: The protocol stack of blockchain sharding protocols.

Protocol layer	**Sub-protocol**	**Functionality**
Cross-shard layer	Atomic Commit	Protecting correctness of cross-shard transactions
	Concurrency Control	Protecting correctness of concurrent transactions
Intra-shard layer	Consensus	Agreeing on transactions within each shard
	Leader election	Electing a leader for each shard
Membership layer	Shard allocation	Partitioning nodes into different shards
Data layer	-	Defining the ledger format

3.2 Protocol stack

In this section, we deconstruct the sharded blockchain into four protocol layers focusing on orthogonal functionalities. As summarised in Table 3.2, the protocol stack consists of four layers, including 1) *data layer* that defines the ledger's format; 2) *membership layer* that partitions nodes into different shards; 3) *intra-shard layer* that agrees on transactions within a shard; and 4) *cross-shard layer* that processes cross-shard transactions.

3.2.1 Data layer

Data layer concerns the ledger format, i.e., how system states are represented, evolved, and stored. Apart from concerns that exist in non-sharded blockchains, the data layer raises some additional concerns on cross-shard transactions, and thus affects the cross-shard layer design in sharded blockchains. First, how transactions are ordered affects the frequency of resolving conflicts between concurrent transactions. In addition, how ledgers are partitioned among shards affects the communication overhead of verifying cross-shard transactions. Concurrency Control and Atomic Commit, the two protocols addressing these two concerns, constitute the cross-shard layer. We will analyse the design spaces and their trade-offs of the data layer in §3.4.

3.2.2 Membership layer

Membership layer is responsible for allocating nodes into different shards. This functionality is informally studied in existing blockchain sharding proposals. We formally study this primitive which we call *shard allocation* in Chapter 4, and assume all shard allocation protocols are secure in this chapter. Summarising the major results of analysis in Chapter 4, we identify a trade-off between the resistance against the *single-shard takeover attack* and the overhead of *reshuffling* in shard allocation protocols, and propose WORMHOLE, a new shard allocation protocol that allows parameterisation over this trade-off.

3.2.3 Intra-shard layer

Intra-shard layer is responsible for processing intra-shard transactions in this shard and cross-shard transactions involving this shard. The intra-shard layer protocol acts the same way as a non-sharded blockchain, where nodes jointly maintain a ledger of transactions and keep agreeing on new transactions. The non-sharded blockchain protocol usually involves two subprotocols, namely *leader election* and (leader-based) *consensus*. For each epoch in a shard, nodes run the *leader election* protocol to elect a leader (aka *primary node*). The leader samples a subset of valid transactions from its memory pool, packs them into a block, and broadcasts this block to other peers in this shard. Nodes in this shard then execute the *consensus* protocol to agree on these transactions and update states of the ledger accordingly. We will analyse leader election and consensus for sharded blockchains in §3.5.

Although blockchain protocols can be leaderless, we still include leader election in the framework for two reasons. The first reason is the completeness: all our evaluated sharded blockchains use leader election protocols, except for Chainspace. The second reason is that without leader election, blockchain protocols have to employ *leaderless consensus*, which remains more of theoretical interest due to the high communication complexity and strong assumptions [28].

3.2.4 Cross-shard layer

Cross-shard layer is responsible for processing cross-shard transactions that involve multiple shards. Processing cross-shard transactions faces two major challenges, namely 1) resolving conflicts between concurrent cross-shard transactions and 2) including cross-shard transactions "atomically": Eventually, a cross-shard transaction is either included or omitted in the ledgers of all involved shards. Both challenges also exist in *distributed transactions* that involve multiple computers. Existing distributed systems research [29] suggests to handle the two tasks by using *Concurrency Control* (CC) [20] and *Atomic Commit* (AC) [22], respectively.

To explain CC and AC, we will use an example cross-shard transaction tx. Let $tx = \{a^1 \to b^2\}$ be a cross-shard transaction that takes object a^1 on shard #1 as input and outputs object b^2 on shard #2. As tx involves both shard #1 and #2, it has to be included in both shards.

Concurrency Control. In non-sharded blockchains, a transaction is included in the ledger instantly once the block including it is included in the ledger. In sharded blockchains, including tx is likely to take multiple epochs, as tx are processed by two different shards that are executing independently. Within this time gap, there might be concurrent conflicted transactions attempting to access a^1 and b^2. To avoid anomalies caused by conflicted transactions, the cross-shard layer has to achieve a property called *isolation* ("I" in ACID [30]), which specifies how and when changes made by a transaction become visible to other transactions. Concurrency Control (CC) [20] is a family of protocols that achieve *isolation* by properly scheduling concurrent-but-conflicted transactions. We will analyse CC for sharded blockchains in §3.6.1.

Atomic Commit. Shard #1 and #2 should have the same decision on tx: Eventually, both shards should include or discard tx. Otherwise, if shard #1 includes but shard #2 omits tx, then a^1 is locked (i.e., cannot be spent) forever. If shard #1 omits but shard #2 includes tx, then a^1 is used as input twice, leading to a double-spending attack. Atomic Commit (AC) is the family of protocols to ensure a transaction is included or discarded in all involved shards. We will analyse AC for sharded blockchains in §3.6.2.

3.3 Existing sharded blockchains

In this section, we summarise the design of sharded blockchains that we will evaluate. As the analysis focuses on permissionless settings, we choose to evaluate seven state-of-the-art permissionless sharded blockchains, including five academic proposals Elastico [3], Omniledger [4], RapidChain [5], Chainspace [6] and Monoxide [7], and two industry projects Zilliqa [8] and Ethereum 2.0 [31].

Elastico and Zilliqa. In Elastico [3], the ledger is formed as a single blockchain. Each node maintains the entire ledger. For each epoch, nodes in a special shard called *final committee* execute a Distributed Randomness Generation (DRG) protocol [32] to generate a random output. Each node solves a PoW with the random output and its public key as input. The node is allocated to a shard according to the prefix of its PoW solution. The first shard becomes the *final committee*. In each shard, nodes execute *Monarchy* [33] to elect a leader, and execute PBFT [34] to agree on the block proposed by the leader. The *final committee* gathers all blocks, merges them into a single block, and appends it to the ledger.

Zilliqa [8] is an industry project that adapts the Elastico protocol with three main optimisations. First, the random output is derived from the last block's hash rather than generated from DRG. Second, instead of using Monarchy for leader election, the node with the smallest PoW solution in a shard is elected as leader. Third, Zilliqa incorporates Collective Signing [35] with PBFT, in order to reduce communication complexity from $O(n^2)$ to $O(n)$.

Omniledger. In Omniledger [4], the ledger is partitioned into different shards. Each part of the ledger is structured as a Directed Acyclic Graph (DAG) of blocks. Each block consists of a number of objects (rather than transactions). Objects are distributed to different shards according to their IDs. For each epoch, all nodes execute the RandHound [36] DRG protocol to produce a random output. Nodes are allocated to shards randomly by a centralised identity authority. In each shard, each node runs Verifiable Random Function (VRF) over the random output and its identity, and the node with the smallest VRF output is elected as leader. The leader proposes a block, and nodes execute ByzCoinX – an optimised version of the ByzCoin [35] consensus protocol – to agree on the block.

Omniledger employs the *Atomix* protocol to process cross-shard transactions. We describe Atomix by using the cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$ in §3.2.4 as an example. First, the user sends tx to shard #1 and requests shard #1 to lock a^1. Nodes in shard #1 verify tx, perform a ByzCoinX consensus on locking a^1, and send a *proof-of-acceptance* of a^1 to the user. The user then constructs a *unlock-to-commit* transaction consisting of the *proof-of-acceptance* of a^1, tx, and b^2, then sends this transaction to shard #2. Nodes in shard #2 execute ByzCoinX consensus over the *unlock-to-commit* transaction. If including this transaction, nodes in shard #2 execute another ByzCoinX consensus on making b^2 an active object. If rejecting this transaction, the leader of shard #2 sends a *proof-of-rejection* to the user, then the user constructs a *unlock-to-abort* transaction consisting of the *proof-of-rejection* to shard #1. Upon the *unlock-to-abort* transaction, nodes in shard #1 perform a ByzCoinX consensus to unlock a^1.

RapidChain. In RapidChain [5], the ledger is partitioned into different shards. Each part of the ledger is formed as a blockchain. Each block consists of a set of transactions. Each transaction is allocated to a shard according to the input object's ID. Each node should solve a PoW to join the system. Nodes are allocated to different shards following the Commensal Cuckoo rule [37]. For each epoch, nodes in each shard elect a leader (RapidChain does not specify the leader election protocol), and execute a synchronous BFT consensus protocol [38] to agree on the block proposed by the leader. The leader is also responsible for coordinating cross-shard transactions. Given cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$, the leader of shard #2 splits tx to two intra-shard transactions: $tx_a^1 = \{a^1 \rightarrow x\}$ and $tx_b^2 = \{x \rightarrow b^2\}$, then sends tx_a^1 and tx_b^2 to nodes in shard #1 and shard #2, respectively. After nodes in shard #1 agree on tx_a^1, the leader of shard #2 proposes tx_b^2, and nodes in shard #2 executes consensus to agree on tx_b^2. As long as tx is valid, tx will eventually be included by both shards (without being rolled back).

Chainspace. Chainspace [6] is a sharded smart contract platform. In Chainspace, the ledger is partitioned into different shards. Each part of the ledger consists of a Directed Acyclic Graph (DAG) of objects, as well as transaction hashes. Objects are distributed to shards according to their IDs. Chainspace does not specify how newly joined nodes are allocated to different shards. If an existing node wants to move to another shard, then it should send a request to the system by using a transaction, and nodes vote to approve its request. Chainspace does not elect leaders for intra-shard consensus. Instead, once a node receives a transaction, this node will initiate a consensus within its shard (plus a cross-shard AC if the transaction is cross-shard) on that transaction.

To process cross-shard transactions, Chainspace employs the *S-BAC* protocol, which combines an optimistic concurrency control (OCC) protocol and an AC protocol. Unlike in other sharded blockchains, S-BAC requires input shards to communicate with each other. Thus, we describe S-BAC using a transaction with multiple inputs in different shards. Let $tx = \{a^1, b^2 \rightarrow c^3\}$ be a transaction with two inputs a^1 on shard #1 and b^2 on shard #2, and an output c^3 on shard #3. In S-BAC, the user first sends $tx = \{a^1, b^2 \rightarrow c^3\}$ to shard #1 and #2, and the two shards verify tx with an intra-shard consensus. If valid, the two shards exchange a^1 and b^2, and execute another consensus to inactivate a^1 and b^2. Note that if a^1 or b^2 is inactivated by another transaction before tx, shard #1 and #2 will agree to abort tx and roll back all operations in tx. After that, the two shards send tx to shard #3, and meanwhile exchange the status of a^1 and b^2 and respond to the user. When shard #3 receives tx, shard #3 performs an intra-shard consensus on creating c^3.

Monoxide. Monoxide [7] partitions the ledger into different shards. Each part of the ledger is formed as a blockchain. Each block contains a number of transactions. Each transaction can only have one input object and one output object. A transaction is allocated to a shard according to its input object's ID. Nodes can join any shards. Similar to Bitcoin, Monoxide employs PoW-based leader election and Nakamoto consensus. Nodes are allowed to do Chu-ko-nu mining, i.e., mine on multiple shards simultaneously. Monoxide takes a similar approach with RapidChain for cross-shard transactions: Each cross-shard transaction is split to multiple intra-shard transactions that are submitted individually. Nodes are incentivised to process cross-shard transactions, as they want to earn the fee in these transactions. Monoxide refers to this guarantee as *eventual atomicity*.

Ethereum 2.0. Ethereum 2.0 is the next generation of the Ethereum project [39], aiming at scaling Ethereum via sharding. In Ethereum 2.0 [31], the ledger is partitioned into different shards, including a beacon chain and a number of shard chains. The beacon chain is the main chain that stores cross-shard transactions, manages validators who produce and verify blocks, and generates random outputs periodically using the RANDAO protocol [40]. Nodes in shard chains execute consensus and append blocks independently. Transactions belong to different smart contracts, and smart contracts are allocated to different shards according to their IDs. Each node stores a part of the ledger as well as block headers of the entire ledger. Each shard samples a subset of nodes called *validators* via the "custody game", a deposit-based weighted sortition. Nodes first deposit some coins in a special *deposit contract* to join the validator registry. Given the latest random output, the deposit contract randomly samples a number of validators and a leader from all nodes. The leader proposes a block, and validators execute the Casper [41] consensus protocol to agree on the block. To process cross-shard transactions, users submit transactions on both input shards and output shards. Given cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$ between a sender and a receiver, the sender splits tx to two intra-shard transactions: $tx_a^1 = \{a^1 \rightarrow x\}$ and $tx_b^2 = \{x \rightarrow b\}$. Then, the sender sends tx_a^1 to shard #1. Once tx_a^1 is included, the shard will create a receipt consisting of the block's Merkle branch with tx_a^1, a^1, and the receiver of tx_b^2. With this receipt and block headers of shard #1, any node can verify the status of a^1. The receiver verifies the receipt, and sends tx_b^2 together with the receipt to shard #2. Shard #2 then validates tx_b^2 using the receipt, and will include tx_b^2 if valid.

3.4 Data layer

We consider the following design choices for the data layer.

- *Ledger unit*: The irreducible unit in a ledger (e.g., transaction or object).
- *Unit allocation*: How a ledger unit is allocated to a shard.

Table 3.3: Design choices of the data layer.

	Ledger unit	Unit alloc.	Consensus unit	Partitioning	Ordering
Elastico	Tx	Arbitrary	Block	Replicated	Total
Omniledger	Object	ID	Block	Sharded	Partial
RapidChain	Tx	Input ID	Block	Sharded	Partial
Chainspace	Tx	Input ID	Tx	Sharded	Partial
Monoxide	Tx	Input ID	Block	Sharded	Partial
Zilliqa	Tx	Arbitrary	Block	Replicated	Total
Ethereum 2.0	Tx	SC. ID	Block	Sharded	Partial

- *Consensus unit*: The item appended to the ledger for each consensus epoch (e.g., block or transaction).
- *Ledger partitioning*: Whether each node stores a part of the ledger (sharded) or the entire ledger (replicated).
- *Ordering*: How ledger units are ordered (e.g., partial ordering or total ordering).

Table 3.3 summarises design choices of the data layer made by blockchain sharding proposals, and we analyse the design trade-offs in ledger partitioning and ordering.

Sharded v.s. replicated. While Elastico and Zilliqa fully replicate the ledger among shards, other protocols divide the ledger into different shards. The ledger partitioning has a direct impact on the construction of oracle $C(tx_x, tx_y)$. Replicating ledgers is similar to *parallel chains* [42]–[45], where the ledger consists of multiple shards, and nodes execute consensus on these shards in parallel. With all shards, nodes can verify cross-chain transactions locally, and $C(tx_x, tx_y)$ does not involve cross-shard communication [12], which we will show is challenging to solve in §3.6. However, replicating ledgers inevitably introduces $O(m)$ more overhead on communication and storage, where m is the number of shards.

This implies a trade-off between the overhead of cross-shard communication and storing/synchronising ledgers, parameterised by the portion of the ledger maintained by a shard, similar to distributed databases. Ethereum 2.0 employs an in-between solution that minimises the overhead and does not require cross-chain communication: Each node stores a part of the ledger as well as block headers of the entire ledger, so that any node can verify cross-shard transactions locally. We consider other possibilities over this trade-off as future work.

Total ordering v.s. partial ordering. Except for Elastico and Zilliqa that enforce total ordering on transactions, other protocols only enforce partial ordering. This is consistent to the design principle of sharding: Trading ordering requirements for parallel execution. In blockchains, some transactions involve different sets of accounts that are independent of each other, and thus can remain unordered. To enforce stronger ordering requirements, the sharded blockchain will require shards to synchronise with each other, which inevitably introduces extra communication overhead. For example, Elastico and Zilliqa allow nodes in a special shard called *final committee* to merge blocks from all shards to a single one. Consequently, the final committee works as a *barrier* [46]: The sharded blockchain stalls only after all shards send their blocks to the final committee.

3.5 Intra-shard layer

In this section, we evaluate the intra-shard layer. The evaluation shows that leader election, while being overlooked by existing designs, is in fact challenging to design and relies on strong assumptions.

3.5.1 Leader election

System setting. Apart from aspects mentioned in §2.1, we also evaluate the *weight* for leader election. A node's *weight* is in proportion to the chance

Table 3.4: Evaluation of intra-shard layer. N/A means the protocol is not specified, and symbol “-” means the protocol is not needed.

(a) Leader election.

	Protocol	System setting			Correctness					Performance
		Network sync.	Fault tolerance	Weight	Public verif.	Uniqueness	Unpredictability	Fairness	Termination	Comm. compl.
Elastico	Monarchy [33]	Sync.	0[‡]	-	✓	✓	✓	✓	✓	$O(n^2)$
Omniledger	VRF-based	Sync.	1	-	✓	✓	✓	✓	✓	$O(n^2)$
RapidChain	N/A	N/A	N/A	-	N/A	N/A	N/A	N/A	N/A	N/A
Chainspace	-	-	-	-	-	-	-	-	-	-
Monoxide	PoW-based	Sync.	1/2	Comp.	✓	✗[*]	✓	✗[†]	✓	$O(n)$
Zilliqa	VRF-based	Sync.	1	Comp.	✓	✓	✓	✓	✓	$O(n^2)$
Ethereum 2.0	PoS-based	N/A	N/A	Stake	✓	N/A	N/A	N/A	N/A	$O(1)$

‡ In *Monarchy* [33], a Byzantine node to withhold all messages, so that no leader will be elected.
* The PoW-based leader election does not provide uniqueness. Instead, Nakamoto consensus ensures only one of blocks at the same height is eventually included.
† With selfish mining [47], the adversary can increase its chance of being elected as leader.

(b) Consensus protocols.

	Protocol	System setting		Correctness				Performance
		Network sync.	Fault tolerance	Agreement	Validity	Termination	Finality	Comm. compl.
Elastico	PBFT	Part. Sync.	1/3	✓	✓	✓	✓	$O(n^2)$[†]
Omniledger	ByzCoinX	Part. Sync.	1/3	✓	✓	✗[‡]	✓	$O(n)$[†]
RapidChain	[38]	Sync.	1/2	✓	✓	✓	✓	$O(n^2)$
Chainspace	PBFT	Part. Sync.	1/3	✓	✓	✓	✓	$O(n^2)$[†]
Monoxide	Nakamoto	Sync.	1/2	✓	✓	✓	✗	$O(n)$
Zilliqa	PBFT + CoSi	Part. Sync.	1/3	✓	✓	✓	✓	$O(n)$
Ethereum 2.0	Casper FFG	Sync.	1/3	✓	✓	✗[*]	✓	$O(n)$

† The communication complexity here is the best-case one. In the worst case where the leader is Byzantine, nodes need to invoke view change protocols, which incur higher communication complexity (e.g., $O(n^3)$ in PBFT).
‡ In ByzCoinX consensus, when the elected leader is Byzantine, then it can break liveness by not generating any block [48].
$*$ Casper FFG cannot terminate when no block reaches the finalisation threshold at a certain height [49], [50].

of being elected as leader. For example, the weight can be computational power and financial stake in PoW-based and PoS-based leader election, respectively.

Correctness and performance metrics. We model the leader election for sharded blockchains based on existing models [19], [51]. Leader election for blockchain sharding should satisfy the following five properties:

- *Public verifiability*: Given a node's public key, its leadership proof and the system state, anyone can verify whether this node is the leader at this system state.
- *Uniqueness*: After election, only one node (in a shard) can provide a valid leadership proof.
- *Unpredictability*: For any epoch t, the probability of making an accurate guess on the leader of any shard at the $(t+1)$-th epoch is in proportion to the ratio between the guessed node's weight and its shard's total weight at epoch t.
- *Fairness*: No node can manipulate its probability of being elected.
- *Termination*: For every epoch, eventually, there will be a node elected as leader.

Public verifiability allows nodes to verify the leader's identity. Uniqueness ensures that only a single node can become the leader and initiate the consensus protocol. Unpredictability and fairness prevent the adversary from corrupting the leader throughout the protocol execution. Termination prevents nodes in the sharded blockchain to lose liveness forever. The performance metric of the leader election protocol is the communication complexity.

Evaluation and analysis. Table 3.4a summarises our evaluation results. Our evaluation shows that leader election is overlooked by existing proposals. Specifically, RapidChain and Ethereum 2.0 require leader election protocols but do not provide detailed specifications. Ethereum 2.0's leader election protocols samples the leader randomly according to the nodes' stake, which can be queried from the ledger and thus is verifiable. Elastico's leader election protocol *Monarchy* [33] incurs $O(n^2)$ communication complexity, and a single Byzantine node can stall the protocol by withholding messages. Monoxide's PoW-based leader election and consensus protocols require synchronous networks [52], [53]. Zilliqa elects the node with the smallest PoW solution in a shard as leader. The process requires an all-to-all broadcast with $O(n^2)$ communication complexity, and requires synchronous networks in order to determine the timeout for the period of broadcasting messages. Omniledger's leader election protocol requires nodes to broadcast their VRF outputs, leading to $O(n^2)$ communication complexity.

Previous studies show that leader election is challenging in permissionless settings. Calzado et al. [51] model and evaluate leader election protocols under crash faults and dynamic mobile networks, and show that no protocol resists against all failures. Boneh et al. [19] formalise Single Secret Leader Election (SSLE) and propose two protocols, but both protocols rely on randomness beacon and complex cryptographic primitives such as Fully Homomorphic Encryption and Indistinguishable Obfuscation.

3.5.2 Consensus

Given the rich literature in consensus [54]–[57], we summarise correctness properties and performance metrics for consensus protocols, and mention previously identified issues of consensus protocols used by these blockchain sharding protocols. We consider system settings mentioned in §2.3, and evaluate consensus protocols against the following properties [21]:

- *Agreement*: No two honest nodes decide differently.

- *Validity*: The value decided must be a value proposed.
- *Termination*: All honest nodes eventually decide.
- *(Optional) Finality* [58]: If a correct node appends a block B before another block B' to its local blockchain, then no correct node appends B' before B to its local blockchain.

We consider a single performance metric, namely the *communication complexity*. Table 3.4b summarises our evaluation on consensus. Note that fault tolerance capacity is quantified by voting power, of which the definition depends on protocol designs. It shows that all consensus protocols either require $O(n^2)$ communication complexity or fail to satisfy *termination*, except for Monoxide's Nakamoto consensus which assumes synchronous networks. To summarise, Omniledger's ByzCoinX consensus protocol does not satisfy termination, as it can lose liveness when the leader is Byzantine, as pointed out by Yu et al. [48]. Ethereum 2.0's Casper FFG consensus protocol executes upon a special smart contract, where nodes (aka validators) post their votes to agree on the next block. Casper FFG assumes synchronous networks **neu2021ebb**, and does not satisfy termination, as it cannot terminate when none of the blocks at a certain height reaches the finalisation threshold **neu2021ebb**, [49], [50].

3.6 Cross-shard layer

In this section, we evaluate the cross-shard layer. For CC, we find that existing sharded blockchains do not separate Concurrency Control (CC) and Atomic Commit (AC), and focus less on CC compared to AC. We evaluate CC based on well-defined models in database literature [22]–[24], and analyse the design space of CC for sharded blockchains. For AC, we show that cross-shard transactions require a variant of AC called Non-Blocking AC (NB-AC) [25]. Our evaluation in the NB-AC model shows that cross-shard communication requires either a centralised coordinator or shared information among shards, which is consistent with a recent work [12]. We also analyse a trade-off between communication overhead and *timing of consistency* [26].

3.6.1 Concurrency Control

System setting. CC does not rely on any of the system setting aspects discussed in §2.3. A CC protocol specifies a set of rules that should be followed when appending a transaction to the ledger. Any node can verify whether a transaction satisfies these rules for a ledger, without relying on communication with other nodes or trusted third parties.

Correctness and performance. CC's correctness includes *safety* and *liveness*. *Safety* defines the *isolation* guarantee of transactions. There are various isolation levels [22]. With higher isolation level, concurrent transactions have less impact on each other, but fewer transactions can be executed concurrently. There are two widely adopted isolation levels, namely *serialisability (I-S)* and *snapshot isolation (I-SI)*. Serialisability is the strongest isolation guarantee, and snapshot isolation is a relatively weaker one.

- *Serialisability (I-S)* [23]: For any set of transactions (that might be executed concurrently) and their execution result r, there exists a sequence of them such that its execution result is equivalent to r.

- *Snapshot isolation (I-SI)* [24]: For any transaction tx, 1) all read operations on an object in tx return the same result (e.g., the result of the first read operation), and 2) iff no other concurrent transactions modify objects read by tx, tx will commit, otherwise tx will rollback.

The key difference between them is that snapshot isolation does not prevent the *write skew* anomaly, where two transactions (tx_1, tx_2) simultaneously read the same set of objects a and b, simultaneously make disjoint

Table 3.5: Evaluation of Concurrency Control.

	Protocol	Correctness		Performance
		Safety	Liveness	No cross-shard comm.
Elastico	Coordinator	I-S[†]	✓	✗
Omniledger	PCC	I-S	✓*	✓
RapidChain	PCC	I-S	✓*	✓
Chainspace	OCC	I-SI[‡]	✓	✗
Monoxide	PCC	I-S	✓*	✓
Zilliqa	Coordinator	I-S	✓	✗
Ethereum 2.0	PCC	I-S	✓*	✓

* Users have incentive to finish cross-shard transactions.
† I-S means serialisability. ‡ I-SI means snapshot isolation.

updates (e.g., tx_1 updates a and tx_2 updates b), and simultaneously commit, without noticing the latest updates made by each other. Serialisability is usually achieved by pessimistic CC (PCC), while snapshot isolation is usually achieved by optimistic CC (OCC).

Liveness is defined the same way as *termination* [59]: Transactions will eventually terminate rather than halting halfway. To conclude, we evaluate CC against the following two properties:

- *Safety*: The protocol guarantees a certain isolation level.
- *Liveness*: The execution of any transaction will eventually terminate.

For CC's performance, we evaluate whether nodes in different shards need to communicate with each other.

Taxonomy of CC. Existing studies [60], [61] categorise CC protocols into four types, namely coordinator-based CC, timestamp-based CC (T/O, aka deterministic scheduling) pessimistic CC (PCC), and optimistic CC (OCC). Coordinator-based CC employs a centralised coordinator to order all transactions and resolve conflicts between them. Timestamp-based CC employs a global clock to timestamp transactions, and processes transactions chronologically. PCC and OCC both rely on locks, but make different assumptions on the conflict rate of transactions. PCC assumes most transactions are conflicted, and follows the two-phase locking (2PL) approach: A transaction locks its accessed objects, then modifies objects, and finally releases locks on the objects. OCC assumes few transactions are conflicted, and follows the modify-validate-commit/rollback approach: A transaction modifies objects while saving a copy of original objects, then verifies if other transactions modify these objects, and finally commits modifications if no other transaction does this, otherwise rolls back modifications.

Evaluation and analysis. Table 3.5 summarises our evaluation results on CC. Elastico and Zilliqa use coordinated-based CC that achieves serialisability and liveness, and requires cross-shard communication. The final committee receives blocks from all shards, and merge all blocks to a single one where transactions are ordered. Omniledger, RapidChain, Monoxide and Ethereum 2.0 use PCC that achieves serialisability, and requires no cross-shard communication. They achieve liveness by using *incentive*: In order to receive coins, users have to submit their cross-shard transactions to output shards. Chainspace's S-BAC protocol implements an OCC protocol that achieves snapshot isolation and liveness. The OCC protocol forbids concurrent write operations, and detecting them requires cross-shard communication.

Design space of CC. According to Table 3.5, existing sharded blockchains use any of them except for T/O. The reason why T/O is not used is that there is no global clock in permissionless networks. Without a global clock, timestamps cannot be reliably verified. Consequently, the adversary can create transactions with any timestamps to re-order transactions arbitrarily, breaking the isolation guarantee.

Coordinator-based CC is less complex to design than PCC and OCC, as the coordinator has a complete view of all transactions. However, coordinator-based CC achieves lower throughput limit than PCC and OCC. In coordinator-based CC, the coordinator has to wait for all shards to produce blocks, then merge all blocks to a single block and broadcast it. The waiting process reduces the concurrency level, and thus the throughput limit.

PCC and OCC make different assumptions on the rate of conflicted transactions: With more conflicted transactions, PCC outperforms OCC, and vice versa. Apart from the conflict rate assumption, OCC requires shards to communicate with each other in order to detect conflicted write operations before including transactions, while PCC does not require cross-shard communication. In addition, OCC allows to roll back transactions, and the rolling back mechanism incurs more complexity in protocol design.

3.6.2 Atomic Commit

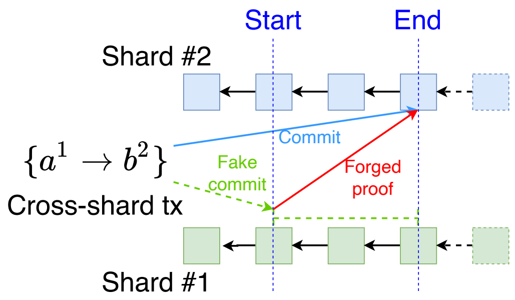
System setting. We consider aspects mentioned in §2.3.

Correctness. Apart from the previously known replay attack [62], we identify three new attacks on the Atomic Commit (AC) for sharded blockchains, allowing us to introduce the required correctness properties. The four attacks are as follows.

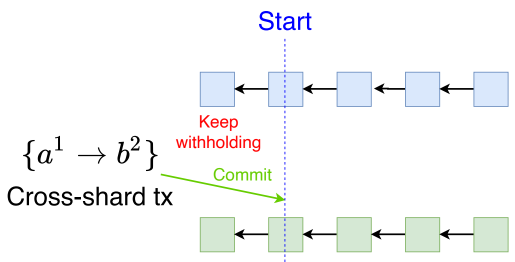
1) Equivocation (Figure 3.1a) The adversary submits two conflicted cross-shard transactions to two shards (e.g., aborting and committing a cross-shard transaction on shard #1 and #2, respectively). Without cross-shard communication or the knowledge of the shard's ledger, two transactions will both be committed, breaking the agreement property.

2) Message withholding attack (Figure 3.1b) The adversary, who might be a user or a node, withholds some messages that should be sent to shards, in order to stop cross-shard transactions from being processed.

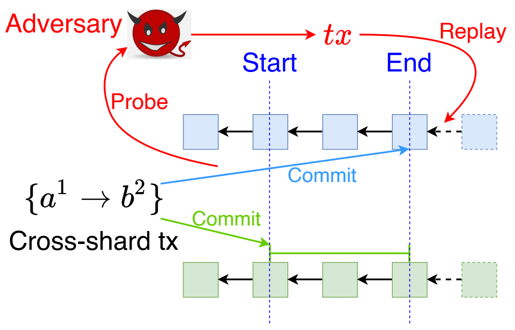
3) Replay attack [62] (Figure 3.1c) The adversary probes a cross-shard transaction, then replays it to the involved shards. In permissionless networks, nodes cannot distinguish whether their received messages are



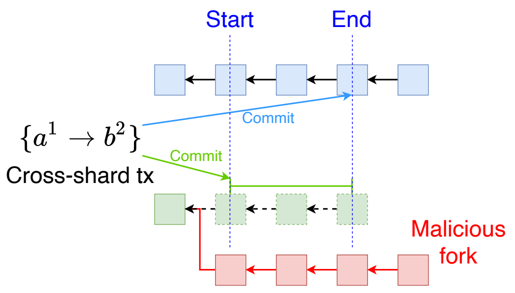
(a) **Equivocation.** The adversary fakes tx that does not exist on shard #1, and submits tx to shard #2 with a fake proof that shard #2 includes tx.



(b) **Messaging withholding.** After submitting tx to shard #1, the adversary keeps withholding tx on shard #2 so that tx cannot terminate.



(c) **Replay.** The adversary probes and replays tx on shard #2. Without proper AC, object b^2 can be locked, and/or object a^1 can be double-spent.



(d) **Publish-revert.** After submitting tx to both shards, the adversary creates a longer fork that reverts tx on shard #1, so that he can take object a^1 as input again.

Figure 3.1: Four possible attacks on Atomic Commit (AC). We use cross-shard transaction $tx = \{a^1 \rightarrow b^2\}$ as an example.

honest but delayed, or malicious. Such replayed messages can lead to two scenarios. The first scenario (e.g., in Chainspace) is that the victim shard considers the replayed transaction to be malicious so rejects it. The second scenario (e.g., in Omniledger, Rapidchain, and Monoxide) is that shards will have conflicted views on the replayed transaction.

4) Publish-revert attack (Figure 3.1d) In probabilistic consensus protocols such as Nakamoto consensus, a transaction might be included first and reverted later. Given a sharded blockchain with probabilistic consensus, it is possible that a cross-shard transaction is included in output shards but is reverted in input shards. Consequently, shards have conflicted views on the transaction.

To resist against these attacks, AC in sharded blockchains should provide the same guarantee as Non-Blocking AC (NB-AC) [25], an AC variant that additionally satisfies termination. NB-AC satisfies the following properties.

- *Agreement*: For any cross-shard transaction, all involved shards have the same decision on it.
- *Termination*: For any cross-shard transaction, all involved shards eventually decide on it.
- *Abort-validity*: A cross-shard transaction will be aborted iff at least one involved shard votes to abort it.
- *Commit-validity*: A cross-shard transaction will be included iff all involved shards vote to include it.

Agreement, abort-validity and commit-validity jointly provide resistance against transaction forging attacks. Termination provides resistance against message withholding attacks. Agreement and termination jointly provide resistance against replay attacks. Agreement provides resistance against publish-revert attacks.

Performance. We evaluate two performance metrics, namely *communication complexity* and *timing of consistency*. We consider three levels of timing of consistency [63]:

- *Strict consistency*: Transactions will be seen by all nodes once included;
- *Casual consistency*: For every two casually related transactions $tx_x < tx_y$ where tx_y is valid only when tx_x is included, then tx_y will not be included unless tx_a is included.
- *Eventual consistency*: Transactions will be seen by relevant nodes but without any timing guarantee.

Table 3.6: Evaluation of Atomic Commit. Symbol "-" means the protocol has no name.

		System setting		**Correctness**				**Performance**	
	Protocol	**Network sync.**	**Fault tolerance**	**Agreement**	**Termination**	**Abort-Validity**	**Commit-Validity**	**Comm. compl.**	**Timing**
Elastico	-	Part. Sync.	1/3	✓	✓	✓	✓	$O(n^2)$	Strict
Omniledger	Atomix	Async.	1/3	✗^{er}	✗^{w}	✓	✓	$O(n)$	Eventual
RapidChain	-	Async.	-	✗^{er}	✗^{w}	✓	✓	$O(1)$	Eventual
Chainspace	S-BAC	Part. Sync.	1/3	✓	✓	✓	✗^{r}	$O(n^2)$	Casual
Monoxide	-	Async.	-	✗^{erp}	✗^{w}	✓	✓	$O(1)$	Eventual
Zilliqa	-	Part. Sync.	1/3	✓	✓	✓	✓	$O(n)$	Strict
Ethereum 2.0	-	Async.	-	✓	✗^{w}	✓	✓	$O(1)$	Eventual

^{e} vulnerable to equivocations. ^{w} vulnerable to message withholding attacks, but with limited impacts on security.
^{r} vulnerable to replay attacks. ^{p} vulnerable to publish-revert attacks, can be fixed by a simple countermeasure.

Evaluation of correctness properties. Table 3.6 summarises our evaluation results on AC. Omniledger, RapidChain and Monoxide are vulnerable to equivocations: The adversary, who acts as a user, can commit a cross-shard transaction on a shard while aborting this cross-shard transaction on the other shard. As they do not require shards to communicate with each other when verifying cross-shard transactions, the cross-shard trasnaction will be committed on a shard and aborted on the other shard, breaking agreement.

As analysed in the paper [62], in Chainspace the replay attack can make a cross-shard transaction to be rejected even all shards commit it, breaking the commit-validity; and in Omniledger, RapidChain and Monoxide the replay attack can create conflicted views on cross-shard transactions for different shards, breaking agreement.

The original Monoxide AC protocol is vulnerable to the publish-revert attack: The adversary can allow cross-shard transactions to be included to output shards while reverting them in input shards, breaking agreement. We suggest a simple verification rule as the countermeasure: A cross-shard transaction is valid in output shards only when it becomes deep enough (and thus irreversible) in input shards. The deepness can be verified by attaching the block headers after a cross-shard transaction in input shards.

Omniledger, RapidChain, Monoxide and Ethereum 2.0's AC protocols are vulnerable to message withholding attacks: The adversary, who acts as a user, can withhold cross-shard transactions on output shards, breaking termination. Nevertheless, such withholding only affects the adversary's coins and does not affect other nodes or the protocol execution. In order to receive coins, users are incentivised to finish cross-shard transactions.

Evaluation of performance metrics. Elastico's AC requires nodes in the final committee to execute a PBFT consensus, and Chainspace's S-BAC protocol requires nodes to execute PBFT consensus for multiple times, both leading to $O(n^2)$ communication complexity. Omniledger and Zilliqa's AC protocols require nodes in involved shards to execute the CoSi-based PBFT consensus, leading to $O(n)$ communication complexity. RapidChain, Monoxide and Ethereum's AC protocols do not need nodes to execute extra intra-shard consensus, leading to $O(1)$ communication complexity.

In Elastico and Zilliqa, after all shards send their blocks to the final committee, the final committee merges all blocks and broadcasts the merged block to all nodes, leading to strict consistency. In Chainspace, each cross-shard transaction invokes an AC among only involved shards, leading to casual consistency. In Monoxide and Chainspace, cross-shard transactions are split into and processed by different shards independently without cross-shard communication, leading to eventual consistency.

Challenges of achieving agreement. Our evaluation shows that achieving *agreement* is challenging, due to equivocations and replay attacks. In fact, when shards maintain different parts of the ledger, achieving *agreement* between shards is proven to be *impossible* without a trusted third party [12]. In a nutshell, when blockchains control disjoint sets of information, solving cross-chain communication, i.e., making blockchains to agree on something, is equivalent to solve fair exchange, which is proven impossible without a trusted third party [64]. The impossibility also applies to cross-shard communication when shards maintain different parts of the ledger.

To workaround the impossibility, sharded blockchains have to either employ a trusted party or enforce shards to share some information. Elastico and Zilliqa employ a final committee that stores the entire ledger and coordinates all cross-shard transactions. This requires the final committee to be trustworthy, and introduces non-negligible storage and communication overhead. Ethereum 2.0 requires nodes to store block headers and work as lightweight clients of all shards, so that every node can verify all cross-shard transactions.

Timing of consistency v.s. communication overhead. Elastico and Zilliqa achieve *strict consistency* by enforcing all shards to synchronise with each other in every epoch. Chainspace achieves *casual consistency* by enforcing shards to synchronise with each other upon each cross-shard transaction involving them. Consequently, only a subset of shards needs to synchronise with each other in every epoch, leading to less communication overhead than Elastico and Zilliqa. Monoxide achieves *eventual consistency* by allowing shards to process cross-shard transactions independently and concurrently, without communicating with each other. This is consistent with the consistency-performance trade-off in traditional databases [65] and distributed systems [66], [67], where enforcing higher consistency level incurs more overhead and reduces performance.

3.7 System-level analysis

In this section, we provide the system-level evaluation on sharded blockchains based on the results in §3.4-3.6. The evaluation results in Table 3.1 show that these protocols fail to satisfy certain correctness properties, or meet them at the price of strong assumptions or at least $O(n^2)$ communication complexity. We also identify an overlooked design choice: The coherence of system settings across layers. Namely, different protocol layers make different assumptions on the system settings. We show that for each evaluated sharded blockchain, replacing subprotocols to those with consistent assumptions will improve its security and/or performance.

3.7.1 Evaluation

System setting. If a system consists of multiple protocols, then it remains secure only when all assumptions made by its protocols hold, otherwise some protocols cannot achieve all correctness properties [27], compromising the entire system. Therefore, the system remains secure under the strongest assumptions made by its protocol layers. Given this observation, we derive the system settings of the sharded blockchains from the individual layer evaluation.

According to the evaluation in §3.4-3.6, all evaluated sharded blockchains assume different system settings on different layers. In addition, Elastico and RapidChain cannot tolerate a single Byzantine node, otherwise the liveness will be broken. Specifically, if the adversary can corrupt a single node in Elastico or RapidChain, then it can stall the leader election protocol or the Feldman Verifiable Secret Sharing protocol [68] in its shard allocation protocol in a shard, respectively. Consequently, the shard will stall and all transactions involved in this shard will not be processed, breaking liveness of the sharded blockchain.

Correctness properties. We consider correctness properties defined in Chapter 2, namely *persistence*, *liveness*, and *validity*. All sharded blockchains satisfy persistence. In Omniledger and Ethereum 2.0, as the adversary can break the consesnus' termination to stall shards according to analysis in §3.5.2, they do not satisfy *liveness*. To fix the liveness issue, one has to choose a consensus protocol that satisfies all correctness properties. Note that such consensus protocols require at least $O(n^2)$ communication complexity according to the well-known lower bound result [69]. In Omniledger, RapidChain, Chainspace and Monoxide, as the adversary can break the Atomic

Commit's agreement to create conflicted cross-shard transactions in different shards according to analysis in §3.6.2, they do not satisfy *validity*. To fix the validity issue, one has to choose an Atomic Commit protocol that satisfies all correctness properties and works around the impossibility result [12] as analysed in §3.6.2.

3.7.2 Coherence of system settings

Based on the evaluation, we observe a design choice that is overlooked by existing proposals, namely the coherence of system settings across layers. In particular, different protocol layers make different system settings, without corresponding architectural guarantees. Consequently, unless a hybrid architecture were used, assuming different system settings across layers will weaken security and/or reduce performance [27]. In the context of blockchain sharding, to remain correct on all layers, the sharded blockchain can only work in the environment that satisfies the strongest system setting made by protocol layers. Otherwise, some protocol layers cannot be fully correct, and the entire sharded blockchain can be compromised.

If the sharded blockchains are deployed in the strongest system settings assumed in Table 3.1, then by replacing protocols relying on weaker system settings with those relying on the strongest system setting, the sharded blockchain can achieve better performance without compromising security. Elastico, Omniledger and Zilliqa can improve the performance by replacing the partially synchronous PBFT consensus protocol with synchronous ones. If Omniledger instantiates the Sybil-resistance mechanism by using an identity authority as described in the paper [4], then Omniledger can also use it to coordinate cross-shard transactions in order to improve the timing of consistency and the communication overhead of Atomic Commit. Monoxide and Ethereum 2.0 can improve the performance by replacing the asynchronous shard allocation and Atomic Commit protocols with synchronous ones.

If the sharded blockchains are deployed in weaker system settings than those in Table 3.1, then they cannot achieve some correctness properties. For example, if the network is partially synchronous, then some sharded blockchains lose some correctness properties. Specifically, Elastico, Omniledger, and Zilliqa's leader election protocols cannot achieve termination, breaking the system's liveness. RapidChain, Monoxide and Ethereum 2.0's consensus protocol cannot achieve agreement, breaking the system's persistence.

Chapter 4

Analysing and improving shard allocation protocols for sharded blockchains

4.1 Introduction

Our evaluation in Chapter 3 shows that existing permissionless sharded blockchains overlook the design of allocating nodes into different shards. While this problem can be solved by various approaches in traditional permissioned settings [70]–[74], it is challenging in permissionless sharded blockchains. First, the adversary can launch *single-shard takeover attacks* (aka *1% attacks*) [75], [76] by gathering its nodes to a single shard and compromise a shard's consensus. As voting power is split among shards, launching such attacks requires much fewer nodes compared to 51% attacks in non-sharded blockchains. To resist against single-shard takeover attacks, sharded blockchains should 1) prevent nodes from choosing shards freely, and 2) achieve *load balance* where each shard contains a comparable number of nodes. Otherwise, shards with fewer nodes can be compromised with less effort. Without a global view of the network and centralised membership management, a common solution is to randomly allocate nodes into shards.

On the other hand, the permissionless setting inherently has *node churn* [77], where nodes may join or leave the system at any time. To achieve load balance under node churn, permissionless sharded blockchains need to adaptively re-balance nodes over time. An intuitive solution is to randomly shuffle all nodes for every epoch. However, when a node is allocated to a new shard, it needs to synchronise the new shard's ledger and find new peers, which introduces non-negligible overhead and makes the node temporarily

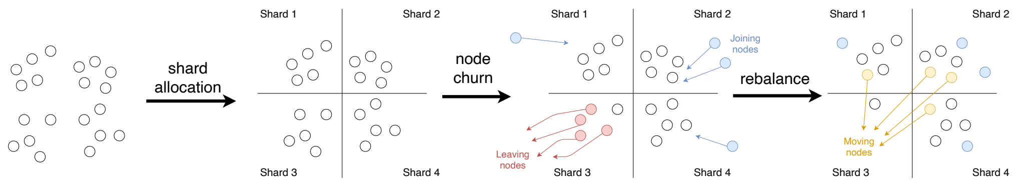


Figure 4.1: An example of shard allocation. New nodes (in blue) may join the system and existing nodes (in red) may leave the system. After a state update, a subset of nodes (in yellow) may be relocated.

unavailable. The blockchain community recognises this issue as the *reshuffling* problem [78], [79].

To address the above issues, permissionless sharded blockchains should employ a mechanism that allocates nodes into shards securely, randomly, and dynamically. We refer to such primitive as *shard allocation*, of which the intuition is depicted in Figure 4.1. Five nodes are allocated in shard #3 and four of them later left the system. To prevent the only node in shard #3 from becoming a single point of failure, the system has to allocate some nodes to shard #3 to re-balance the shards.

A systematic study on shard allocation is still missing. Existing works on permissionless sharded blockchains focus on either the system-level design [3]–[8], [10], [31], [80] or other components such as ledger structure [81] and cross-shard communication [12]. Other peer-to-peer protocols such as distributed hash tables [82], [83] and distributed slicing [84]–[87] cannot be directly adapted for this primitive, as they usually assume rational adversary and choose liveness over safety.

Contributions. This chapter provides the first study on shard allocation, the overlooked core component for permissionless shared blockchains. In particular, we formalise the shard allocation protocol, evaluate the shard allocation protocols of existing blockchain sharding protocols, observe insights and propose WORMHOLE, a correct and efficient shard allocation protocol for permissionless blockchains. Our contributions are summarised as follows.

1. We **provide the first study on formalising the shard allocation protocol for permissionless blockchains (§4.2)**. The formalisation includes the syntax, correctness properties and performance metrics, and can be used as a framework for evaluating shard allocation protocols.

2. Based on our framework, we **evaluate the shard allocation protocols in seven state-of-the-art permissionless sharded blockchains (§4.3)**, including five academic proposals Elastico [3] (CCS'16), Omniledger [4] (S&P'18), RapidChain [5] (CCS'19), Chainspace [6] (NDSS'19), and Monoxide [7] (NSDI'19), and two industry projects Zilliqa [8] and Ethereum 2.0 [9]. Our results show that *none of these protocols is fully correct or achieves satisfactory performance*.

3. We **observe and prove the impossibility of simultaneously achieving optimal *self-balance* and *operability* (§4.4.1).** Self-balance represents the ability to re-balance the number of nodes in different shards; and operability represents the system performance w.r.t. the cost of re-allocating nodes to a different shard. While this impossibility has been conjectured [78], [79] and studied informally [4], we formally prove it is impossible to achieve optimal values on both, and quantify the trade-off between them. All existing sharded blockchains except for Omniledger make extreme choices on either self-balance or operability, leading to serious security or performance issues.

4. We **identify and define a property *memory-dependency* that is necessary for shard allocation protocols to parameterise the trade-off between self-balance and operability (§4.4.2)**. *Memory-dependency* (aka non-memorylessness in signal processing literatures [88]) specifies that the shard allocation relies on both the current and previous system states. The parameterisation support opens a new in-between design space and makes the system configurable for different application sce-

narios. We formally prove the necessity of *memory-dependency* for supporting such parameterisation.

5. We **propose** WORMHOLE**, a correct and efficient shard allocation protocol (§4.5), and analyse how to integrate** WORMHOLE **into sharded blockchains (§4.6)**. We formally prove that WORMHOLE achieves all correctness properties, and supports parameterisation of self-balance and operability. We also classify existing sharded blockchains, and analyse how to integrate WORMHOLE into each type of them.

6. We **implement** WORMHOLE**, and evaluate its overhead and performance metrics in real-world settings (§4.7).** We implement WORMHOLE in Rust, and evaluate the overhead of integrating WORMHOLE into different designs of sharded blockchains. We simulate WORMHOLE with 128 shards and 32768 nodes, and evaluate the dynamic load balance and operability under different churn conditions. The results show that WORMHOLE achieves consistent load balance and operability with our theoretical analysis, and can recover quickly from load imbalance.

4.2 Formalising shard allocation

This section defines shard allocation protocol, including its system model, syntax, correctness properties, and performance metrics.

4.2.1 System model

A sharded blockchain consists of a fixed number of m shards, each of which maintains a ledger formed as a blockchain, and processes transactions concurrently. Each node i in the system has a pair of secret key sk_i and public key pk_i, and is identified by pk_i. The sharded blockchain proceeds in epochs. For each epoch t, new nodes and existing nodes execute the shard allocation protocol to obtain a new shard membership w.r.t. the current system state st_t. Nodes find peers in the same shard by exchanging shard memberships/proofs, then execute consensus with peers to agree on new blocks. Each block includes the block proposer's shard membership and proof, apart from other data common in non-sharded blockchains. Let n_k^t be the number of nodes in shard $k \in [m]$ and $n^t = \sum_{k=1}^{m} n_k^t$ be the total number of nodes in epoch t, where $[m] = \{1, 2, \ldots, m\}$.

Epoch and global system state. An epoch t begins when a new global and unique system state st_t is available. How and when a system state is generated depends on the concrete protocol design. For example, Elastico [3], Omniledger [4], RapidChain [5] and Ethereum 2.0 [9] use a decentralised randomness beacon protocol to generate random outputs as system states; Zilliqa [8] merges blocks from all shards in an epoch, then extracts a global system state from the merged block.

Most sharded blockchains demand that the system state can be accessed by nodes securely and synchronously. We make the same assumption in line with these proposals. To focus on analysing the shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$, we assume the system state generation protocols are secure.

Sybil resistance. To defend against Sybil attacks where the adversary spawns numerous nodes to compromise the consensus, permissionless sharded blockchains must employ a Sybil-resistant mechanism. For example, Elastico, RapidChain, and Zilliqa require nodes solving PoW puzzles to obtain shard memberships; Monoxide and Ethereum 2.0 employ PoW-based and Proof-of-Stake (PoS)-based Nakamoto-style consensus; and Omniledger supports any Sybil-resistant mechanisms, and instantiates it with a trusted identity authority. Among these Sybil resistance mechanisms, Nakamoto-style consensus requires network synchrony and certain fault tolerance capacity [89], affecting the sharded blockchain's system model.

Node churn. *Node churn* [77] happens at any point of the protocol execution: some new nodes join and some existing nodes leave the sharded blockchain. As we study shard allocation across epochs, we consider node churn happens at the end of each epoch for simplicity. Let α and β be the joining rate and leaving rate, respectively. At the end of epoch t, $\alpha_t n^t$ new nodes will join and $\beta_t n^t$ existing nodes will leave the sharded blockchain. For two consecutive epochs t and $t+1$, $n^{t+1} = (1 - \beta_t + \alpha_t)n^t$.

Network model. The network model concerns the timing guarantee of delivering messages. Depending on different proposals' settings, the network model is either synchrony, partial synchrony [17], or asynchrony. A network is synchronous if the adversary can delay a message up to a known finite time bound Δ; is asynchronous if the adversary can delay a message arbitrarily without any known time bound; and is partially synchronous [17] if it is asynchronous before an unknown Global Stabilisation Time (GST) and becomes synchronous after GST.

We say $\Pi_{\mathsf{ShardAlloc}}$ is synchronous if the adversary can break its safety by delaying messages beyond Δ; is partially synchronous if safety is guaranteed before GST and both safety and liveness are guaranteed after GST; and is asynchronous if a correct node can calculate its shard membership locally without communicating with other nodes (assuming synchronous access to system states).

Adversary. The adversary aims to break some of $\Pi_{\mathsf{ShardAlloc}}$'s correctness properties that we will define in §4.2.3. Let ϕ be the fault tolerance capacity of $\Pi_{\mathsf{ShardAlloc}}$, where ϕ is no bigger than the consensus protocol's fault tolerance capacity Ψ. Otherwise, even when the adversary's nodes are evenly distributed among shards, the adversary can compromise every shard. The adversary is adaptive: at any time, it can corrupt any set of less than ϕn^t nodes, i.e., make these nodes Byzantine, where t is the epoch number. The adversary can read internal states of corrupted nodes, and direct corrupted nodes to arbitrarily forge, modify, delay, and/or drop messages from them. The adversary can read and/or delay messages from correct nodes. The delay period is subjected to the network model assumed by the sharded blockchain.

4.2.2 Syntax

We formally define the shard allocation protocol as follows.

Definition 4.2.1 (Shard allocation $\Pi_{\mathsf{ShardAlloc}}$)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ is a tuple of polynomial time algorithms

$$\Pi_{\mathsf{ShardAlloc}} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Join}, \mathsf{Update}, \mathsf{Verify})$$

$\mathsf{Setup}(\lambda) \rightarrow pp$: On input the security parameter λ, outputs the public parameter pp.

$\mathsf{Gen}(pp) \rightarrow (sk, pk)$: A probabilistic function, which on input public parameter pp, produces a secret key sk and a public key pk.

$\mathsf{Join}(pp, sk_i, st_t) \rightarrow (k, \pi_{i,st_t,k})$: On input secret key sk_i, public parameter pp and state st_t, outputs the ID k of the shard assigned for node i, the proof $\pi_{i,st_t,k}$ of assigning i to k at st_t. The input may also be public key pk_i of node i, depending on concrete constructions. This also applies to $\mathsf{Update}(\cdot)$.

$\mathsf{Update}(pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \rightarrow (k', \pi_{i,st_{t+1},k'})$: On input the public parameter pp, secret key sk_i, state st_t, shard index k, proof $\pi_{i,st_t,k}$ and the next state st_{t+1}, outputs the identity k' of the newly assigned shard for i, a shard assignment proof $\pi_{i,st_{t+1},k'}$.

$\mathsf{Verify}(pp, pk_i, st_t, k, \pi_{i,st_t,k}) \rightarrow \{0, 1\}$: Deterministic. On input public parameter pp, i's public key pk_i, system state st_t, shard index k and shard assignment proof $\pi_{i,st_t,k}$, outputs 0 (false) or 1 (true).

Algorithm 1: Typical execution of shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ in a sharded blockchain, from node i's perspective.

```
$(k_t, \pi_{i,st_t,k_t}) \leftarrow \Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(pp, sk_i, st_t)$    // Join in epoch $t$
$st_*, k_*, \pi_* \leftarrow st_t, k_t, \pi_{i,st_t,k_t}$    // State, shard and proof in epoch $t$
repeat
    Wait for a new state $st_+$
    // Update shard membership and proof
    $(k_*, \pi_{i,st_*,k_*}) \leftarrow \Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp, sk_i, st_*, k_*, \pi_{i,st_*,k_*}, st_+)$
    $st_* \leftarrow st_+$
    // Messages may attach $k_*$ and $\pi_{i,st_*,k_*}$
    Execute consensus with peers in shard $k_*$
until node $i$ leaves the system
```

Algorithm 1 describes the typical execution of $\Pi_{\mathsf{ShardAlloc}}$ in a sharded blockchain, from a node i's perspective. $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Setup}(\lambda)$ is executed once at the beginning of the protocol execution. To join the system in epoch t, node i executes $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ to obtain a shard membership k_* and the associated membership proof π_{i,st_*,k_*}, so that it can execute consensus with peers in shard k_*. Upon epoch $t+1$, node i needs to execute $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ to update its shard membership. Other nodes can execute $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Verify}(\cdot)$ to verify whether node i has a valid and updated shard membership.

4.2.3 Correctness properties

We consider three correctness properties for $\Pi_{\mathsf{ShardAlloc}}$, namely *liveness*, *allocation-randomness*, and *unbiasibility*, plus an optional property *allocation-privacy*.

Liveness. This property ensures that correct nodes can obtain valid shard memberships *timely*: Given a system state, all correct nodes will finish computing $\mathsf{Update}(\cdot)$ (or $\mathsf{Join}(\cdot)$ if the node newly joins the system) before the next epoch. Otherwise, nodes cannot find their shards or participate in consensus, and consequently, the block producing is stalled.

Definition 4.2.2 (Liveness). A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ satisfies liveness iff for every epoch t, every correct node i will finish computing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp, sk_i, st_{t-1}, k_{t-1}, \pi_{i,st_{t-1},k_{t-1}}, st_t)$ (or $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(pp, sk_i, st_t)$

if $t = 1$) before epoch $t+1$ such that $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Verify}(pp, pk, +i, st_t, k_t, \pi_{i,st_t,k_t}) = 1$, where pp is the public parameter, sk_i is node i's secret key, $(st_{t-1}, k_{t-1}, \pi_{i,st_{t-1},k_{t-1}})$ and $(st_t, k_t, \pi_{i,st_t,k_t})$ are the system state, node i's allocated shard and node i's shard membership proof in epoch $t-1$ and t, respectively.

Allocation-randomness. This property ensures that every node is allocated to a random shard [3], [4], [8]. Otherwise, if the adversary can predict shard allocation results, then it can launch the *single-shard takeover attack* by corrupting nodes that will be allocated to a specific shard. We stress that allocation-randomness specifies the shard allocation process for every node independent of others, rather than specifying a global permutation of all nodes' shard allocation results, which is impossible when node churn exists and nodes have no global view on the network. Such independent decisions may lead to some extreme cases where some shards are almost empty, but with negligible probability **Example3.10**, [90].

We consider two parts of allocation-randomness, namely *join-randomness* and *update-randomness*. Join-randomness specifies that a newly joined node is assigned to each shard with equal probability.

Definition 4.2.3 (Join-randomness)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards satisfies join-randomness iff for any secret key sk_i, public parameter pp and state st_t, the probability that node i is allocated to shard k is

$$\Pr\left[k = k' \middle| \ (k', \pi_{i,st_t,k'}) \leftarrow \mathsf{Join}(pp, sk_i, st_t) \ \right] = \frac{1}{m} \pm \epsilon$$

where $k, k' \in [m]$, and ϵ is a negligible value.

Update-randomness specifies the probability distribution of existing nodes' shard allocation. To remain balanced under churn, existing nodes may need to move to other shards upon state update. Moving to a new shard is computation- and communication-intensive, as a node needs to synchronise and verify the new shard's ledger, which can take hundreds

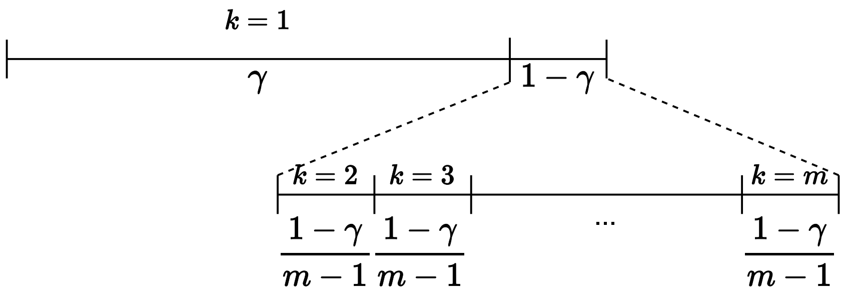


Figure 4.2: Update-randomness. After executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, the probability that a node stays in its shard (say shard 1) is γ, and the probability of moving to each other shard is $\frac{1-\gamma}{m-1}$.

of Gigabytes [4], [91]–[93]. If a large portion of nodes move to other shards upon each state update, then this introduces non-negligible overhead and may make the system unavailable for a long time. To avoid this, only a small subset of nodes should be moved within each state update. We define γ as the probability that a node stays in the same shard after a state update. We define update-randomness as follows.

Definition 4.2.4 (Update-randomness)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards satisfies update-randomness iff there exists $\gamma \in [0,1)$ such that for any $k \in [m]$, secret key sk_i and public parameter pp, the probability that node i updates its shard membership from shard k at state st_t to shard k' at state st_{t+1} is

$$\Pr\left[k = k' \middle| \begin{array}{c} (k', \pi_{i,st_{t+1},k'}) \leftarrow \\ \mathsf{Update}(pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \end{array} \right] = \begin{cases} \gamma \pm \epsilon & (k' = k) \\ \frac{1-\gamma}{m-1} \pm \epsilon & (k' \neq k) \end{cases} \tag{4.1}$$

where $k' \in [m]$, and ϵ is a negligible value.

When $\gamma = \frac{1}{m}$, $\Pi_{\mathsf{ShardAlloc}}$ achieves optimal update-randomness, as all nodes are shuffled randomly under uniform distribution. This definition is intuitively depicted in Figure 4.2.

Definition 4.2.5 (Allocation-randomness)**.** A shard allocation protocol satisfies allocation-randomness if it satisfies join-randomness and update-randomness.

Unbiasibility. This property ensures that the adversary cannot manipulate the shard allocation results. While allocation-randomness defines the probability distribution of shard allocation, *unbiasibility* rules out attacks on manipulating the probability distribution, e.g., the join-leave attack [94], [95].

Definition 4.2.6 (Unbiasibility)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ satisfies unbiasibility iff given a system state, no node can manipulate the probability distribution of the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, except with negligible probability.

Allocation-privacy. This property ensures that no one can learn a node's shard membership without the node providing them by itself. Compared to allocation-randomness, *allocation-privacy* further prevents the adversary from computing a node's membership if the adversary has no access to the node's secret key. We consider allocation-privacy to be optional, as it has both advantages and disadvantages. On the positive side, allocation-privacy is necessary for the sharded blockchain to resist against the adaptive adversary: If the adversary cannot learn others' shard memberships, then it cannot corrupt nodes in a specific shard, but only a random set of nodes scattered across shards. On the negative side, allocation-privacy makes nodes difficult to find peers in the same shard. If the sharded blockchain employs a consensus protocol that requires broadcasting operations, then nodes have to execute an extra peer finding protocol [3], [8] before executing consensus, introducing non-negligible communication overhead. Thus, if the sharded blockchain is not required to resist against an adaptive adversary, then $\Pi_{\mathsf{ShardAlloc}}$ does not need to achieve allocation-privacy.

Definition 4.2.7 (Join-privacy)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards provides join-privacy iff for any secret key sk_i, public parameter pp, and state st_t, without the knowledge of $\pi_{i,st_t,k}$ and sk_i, the probability of making a correct guess k' on k is

$$\Pr\left[k' = k \middle| (k, \pi_{i,st_t,k}) \leftarrow \mathsf{Join}(pp, sk_i, st_t) \right] = \frac{1}{m} \pm \epsilon$$

where $k, k' \in [m]$, and ϵ is a negligible value.

Definition 4.2.8 (Update-privacy)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards provides update-privacy iff for some $\gamma \in [0, 1)$, any $k \in [1, m]$, secret key sk_i, public parameter pp, and two consecutive states st_t and st_{t+1}, without the knowledge of $\pi_{i,st_{t+1},k'}$ and sk_i, the probability of making a correct guess k'' on k' is

$$\Pr\left[k'' = k' \middle| \begin{array}{c} (k', \pi_{i,st_{t+1},k'}) \leftarrow \\ \mathsf{Update}(pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}) \end{array} \right] = \begin{cases} \gamma \pm \epsilon & (k'' = k) \\ \frac{1-\gamma}{m-1} \pm \epsilon & (k'' \neq k) \end{cases} \tag{4.2}$$

where $k', k'' \in [m]$, and ϵ is a negligible value.

Definition 4.2.9 (Allocation-privacy)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ satisfies allocation-privacy iff it satisfies both join-privacy and update-privacy.

4.2.4 Performance metrics

We consider three performance metrics, namely communication complexity, self-balance and operability.

Communication complexity. Communication complexity is the amount of communication (measured by the number of messages) required to complete a protocol [96]. For shard allocation, we consider the *communication complexity* of all correct nodes obtaining shard memberships when joining, and updating shard memberships upon a new epoch. The communication of synchronising new shards is omitted.

Self-balance. Nodes should be uniformly distributed among shards. Otherwise, the fault tolerance threshold of shards with fewer nodes and the performance of shards with more nodes may be reduced [7], [97]. Due to node churn and lack of a global view, reaching global load balance is impossible for permissionless networks. Instead, the randomised self-balance approach – where a subset of nodes move to other shards randomly – provides the optimal load balance guarantee. We quantify the *self-balance* as the ability that $\Pi_{\mathsf{ShardAlloc}}$ recovers from load imbalance.

Definition 4.2.10 (Self-balance)**.** When executing $\Pi_{\mathsf{ShardAlloc}}$ on m equal-sized shards in epoch t (i.e., $n_i^t = n_j^t$ for all $i, j \in [m]$), $\Pi_{\mathsf{ShardAlloc}}$ is μ-self-balanced iff

$$\mu = 1 - \max_{\forall i,j \in [m]} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t} \tag{4.3}$$

Value μ measures the level of imbalance among shards after an epoch. When $\mu = 1$, $\Pi_{\mathsf{ShardAlloc}}$ achieves the optimal self-balance: Regardless of how many nodes join or leave the system during the last epoch, the system can balance itself within an epoch.

Operability. To balance shards, $\Pi_{\mathsf{ShardAlloc}}$ should move some nodes to other shards upon each state update. As mentioned, moving nodes to other shards introduces non-negligible overhead and may make the system unavailable for a long time. *Operability* was introduced to measure the cost of moving nodes [4]. We define operability as the probability that a node stays at its shard upon a state update. If $\Pi_{\mathsf{ShardAlloc}}$ satisfies update-randomness with γ (Definition 4.2.4), then its operability is γ, i.e., γ-operable. When $\gamma = 1$, $\Pi_{\mathsf{ShardAlloc}}$ is most operable: Nodes will never move after joining the network.

4.3 Evaluating existing protocols

In this section, we model shard allocation protocols of seven state-of-the-art sharded blockchains and evaluate them based on our framework. For simplicity, we refer a sharded blockchain's shard allocation protocol as

Table 4.1: Evaluation of seven permissionless shard allocation protocols. Red indicates strong assumptions, unsatisfied correctness properties, and relatively weaker performance. Yellow indicates moderate assumptions and partly satisfied correctness properties. Green indicates weak assumptions, satisfied correctness properties, and better performance.

		System model			Correctness					Performance metrics			
	State update	Network model	Trusted components	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy^o	Join comm. compl.	Update comm. compl.	Self-balance	Operability
Elastico	New block	Sync.	-	1/3	✓	✓	✓	✗	✓	$O(n^f)$	$O(n^f)$	1	$\frac{1}{m}$
Omniledger	New block	Part. sync.	-	1/3	✓	✓	✓	✓	✗	$O(n)$	$O(n) \sim O(n^3)$	$1 - \frac{(2m-3)\beta_t}{3m-3}$	$\frac{2}{3}$
RapidChain	Nodes joining	Sync.	-	0	✗	✓	✓	✓	✗	$O(n^2)$	$O(n^2)$	$1 - \beta_t$	$max(1 - \kappa\alpha_t n, 0)$
Chainspace	-	Async.	Smart contracts	Ψ	✓	✓	✗	✗	✗	-	$O(n)$	$1 - \beta_t$	-
Monoxide	-	Async.	-	Ψ	✓	✓	✗	✓	✗	0	0	$1 - \beta_t$	1
Zilliqa	New block	Async.	-	Ψ	✓	✓	✓	✗	✓	$O(n)$	$O(n)$	1	$\frac{1}{m}$
Ethereum 2.0	-	Async.	-	Ψ	✓	✓	✗	✓	✗	0	0	$1 - \beta_t$	1
WORMHOLE (§4.5)	New rand.	Async.	Rand. Beacon*	Ψ	✓	✓	✓	✓	✓	$O(n)$	$O(n)$	$1 - \beta_t + \frac{\beta_t}{2^{op}}$	$1 - \frac{m-1}{m \cdot 2^{op}}$

^{o} Optional property.
$*$ WORMHOLE can rely on an external randomness beacon, or allow a group of nodes to run a decentralised randomness beacon protocol similar to Elastico, Omniledger, RapidChain and Ethereum 2.0.
Ψ is the fault tolerance capacity of the sharded blockchain's consensus protocol.

the sharded blockchain's name. Our evaluation (summarised in Table 4.1) shows that none of them is fully correct or achieves satisfactory performance.

4.3.1 Evaluation criteria

The evaluation framework includes the system model, correctness properties (§4.2.3), and performance metrics (§4.2.4). The system model concerns the network model and fault tolerance capacity in §4.2.1, plus the trusted components that some proposals assume in order to guarantee the correctness. The node churn and adversary's goals in §4.2.1 are common in all proposals, and thus are omitted. As the evaluation framework focuses on shard allocation, other subprotocols in sharded blockchains – e.g., system state generation, consensus and cross-shard communication – are assumed secure.

We stress that shard allocation's fault tolerance capacity is no bigger than the consensus protocol's fault tolerance capacity Ψ. For example, if all correctness properties in the shard allocation protocol hold even when all nodes are Byzantine (e.g., guaranteed by a trusted third party), the shard allocation protocol achieves the fault tolerance capacity of Ψ.

4.3.2 Overview of evaluated proposals

We choose seven state-of-the-art sharded blockchains, including five academic proposals Elastico [3], Omniledger [4], Chainspace [6], RapidChain [5], and Monoxide [7], and two industry projects Zilliqa [8] and Ethereum 2.0 [9]. We briefly describe their shard allocation protocols below.

Elastico, Omniledger and RapidChain rely on distributed randomness generation (DRG) protocols for shard allocation. In Elastico, nodes in a special shard called *final committee* run a commit-and-reveal DRG protocol [98] to produce a random output. Each node then solves a PoW puzzle derived from the random output and its identity, and will be assigned to a shard according to its PoW solution. In Omniledger, all nodes in the network execute a synchronous leader election protocol based on a verifiable random function. The leader then initiates the *RandHound* [36] DRG protocol with the other nodes to generate a random output. If the leader election fails for five times, then nodes fallback to run an asynchronous DRG protocol [99]. Given the latest random output, nodes derive a unique permutation of them, and $\frac{1}{3}$ nodes in the beginning of the permutation are shuffled to other shards randomly. In RapidChain, nodes in a special shard called *reference committee* execute a Feldman Verifiable Secret Sharing (VSS) [68]-based DRG protocol to generate a random output. To join the system, a node needs to solve a PoW puzzle parameterised by the random output. The puzzle serves no other purpose than allowing the node to join the system. The *reference committee* then executes the *Commensal Cuckoo* rule [37] as follows. Interval $[0, 1)$ is equally divided into different fragments, each representing a shard. Each new node is mapped to an ID $x \in [0, 1)$ based on its identity, and is allocated to the shard whose interval includes x. Existing nodes with IDs close to x are "pushed" to other shards randomly.

Chainspace, Monoxide, Zilliqa and Ethereum 2.0 do not rely on DRG protocols for shard allocation. In Chainspace, a node can apply to move to another shard at any time, and other nodes vote to decide on the applications. The voting works over a special smart contract `ManageShards`, whose execution is assumed to be correct and trustworthy. Monoxide and Ethereum 2.0 allocate nodes into different shards according to their addresses' prefixes. Zilliqa is built upon Elastico, but it uses the last block's hash value as the current epoch's random output.

4.3.3 System model

Network model. A shard allocation protocol is synchronous if the adversary can break its safety by delaying messages beyond the latency upper bound Δ; is partially synchronous if such $> \Delta$ delay only affects liveness but not safety, and liveness is resumed once the network becomes synchronous; and is asynchronous if a correct node can calculate its shard membership locally without communicating with other nodes. Note that we assume in §4.2.1 that nodes have secure and synchronous access to system states, and other subprotocols of the sharded blockchain (including system state generation) are secure. Elastico and RapidChain are synchronous, as they employ the synchronous DRG protocols [68], [98]. Omniledger is partially synchronous, as it employs the partially synchronous *RandHound* DRG protocol. Chainspace, Monoxide, Zilliqa and Ethereum 2.0 are asynchronous: In Chainspace, a node submits a smart contract transaction to obtain or update a shard membership, and the liveness is achieved once the transaction is received by the smart contract; Monoxide and Ethereum 2.0 allow nodes to calculate shards locally without communicating with others; and Zilliqa replaces the DRG [98] in Elastico by using block hashes as system states that can be accessed synchronously by assumption, and nodes can calculate their shards locally given the system states.

Trusted components. These protocols assume no trusted component, except for Chainspace that assumes trusted smart contracts.

Fault tolerance capacity. Elastico and Omniledger achieve the fault tolerance capacity of $\phi = \frac{1}{3}$, which is inherited from their DRG protocols. RapidChain cannot tolerate any faults, as one faulty node can make the Feldman VSS lose liveness by withholding shares. In Chainspace, Monoxide, Zilliqa, and Ethereum 2.0, all correctness properties of shard allocation are guaranteed when all nodes are Byzantine. For Monoxide and Ethereum 2.0, computing shards is offline. Chainspace assumes trusted smart contracts. For Zilliqa, blocks are produced correctly by assumption and shard computation is offline. Thus, their shard allocation protocols achieve fault tolerance capacity of Ψ.

4.3.4 Correctness properties

Public verifiability. All of these shard allocation protocols achieve public verifiability except for RapidChain. RapidChain's shard allocation is not publicly verifiable, as the deployed Commensal Cuckoo protocol is not publicly verifiable.

Liveness. All shard allocation protocols satisfy liveness.

Allocation-randomness. Elastico, Omniledger, RapidChain, and Zilliqa satisfy allocation-randomness, as all nodes are shuffled for each epoch. Chainspace does not satisfy allocation-randomness, as nodes can choose which shard to join. Monoxide and Ethereum 2.0 do not satisfy allocation-randomness, as nodes can choose their preferred shards by choosing addresses.

Unbiasibility. Elastico and Zilliqa do not fully achieve unbiasibility. Compared to the PoW puzzles in Bitcoin-like systems, the PoW puzzles in Elastico and Zilliqa are less challenging to solve, allowing the adversary to solve multiple puzzles within an epoch and choose a preferred shard to join. Chainspace does not achieve unbiasibility, as it does not satisfy allocation-randomness and nodes are free to choose shards.

Allocation-privacy. Elastico and Zilliqa satisfy allocation-privacy, as the allocated shard remains secret if the node does not reveal its PoW solution. Therefore, Elastico and Zilliqa employ an extra peer finding mechanism called "overlay setup", where a special shard called "directory committee" collects and announces nodes' allocated shards. Omniledger, RapidChain, and Chainspace do not satisfy allocation-privacy as memberships can be queried at the identity blockchain, the reference committee and the `ManageShards` smart contract, respectively. Monoxide and Ethereum 2.0 do not satisfy allocation-privacy, as nodes' addresses are publicly known.

4.3.5 Performance metrics

Communication complexity. Elastico's shard allocation requires $O(n^f)$ messages per epoch, where n and f are the number of nodes and faulty nodes, respectively. For each epoch, the final committee needs to run the DRG protocol, which consists of a vector consensus [32] with communication complexity $O(n^f)$. Ideally, the final committee in Elastico has $\frac{n}{m}$ nodes, and the communication complexity is $O(\frac{n}{m}^{\frac{f}{m}}) = O(n^f)$ (m is constant). For Omniledger, $\mathsf{Join}(\cdot)$ requires $O(n)$ communication, as each new node requests to a node for joining the system. The communication complexity of $\mathsf{Update}(\cdot)$ is $O(n)$ or $O(n^3)$: The best case of $\mathsf{Update}(\cdot)$ is that the leader election and RandHound are both successful, leading to $O(n)$ messages; and the worst case is that nodes fallback to run the asynchronous DRG [99] with communication complexity $O(n^3)$. RapidChain's shard allocation requires $O(n^2)$ messages per epoch, which is inherited from Feldman VSS [68]. Monoxide and Ethereum 2.0 requires no communication for shard allocation, as nodes decide their shards locally. Zilliqa requires $O(n)$ messages per epoch as each node needs to retrieve the latest block.

Self-balance. In Elastico and Zilliqa, all nodes are shuffled for each epoch, leading to the self-balance of 1 with negligible bias. In Omniledger, $\frac{1}{3}$ nodes are shuffled for each epoch, leading to operability $\gamma = \frac{2}{3}$. By Lemma 4.4.1 (introduced later in §4.4.1), Omniledger's self-balance will be $\mu = 1 - \frac{(2m-3)\beta_t}{3m-3}$. The self-balance of RapidChain, Chainspace, Monoxide and Ethereum 2.0 is $1-\beta_t$. In the worst case where no nodes newly join the system and βn^t nodes in the same shard leave the system, self-balance becomes $\frac{n-\beta_t n}{n} = 1 - \beta_t$.

Operability. The operability of Elastico and Zilliqa are $\frac{1}{m}$, as all nodes are shuffled for each new epoch. The operability of Omniledger is $\gamma = \frac{2}{3}$, as $\frac{1}{3}$ nodes are shuffled for each epoch. The operability of RapidChain is $\max(1-\kappa\alpha_t n, 0)$, where $\kappa \in [0, 1]$ is the size of the interval in which nodes should move to other shards, and α_t is the join churn rate in epoch t. In epoch t, there are $\alpha_t n$ nodes joining the network, and each newly joined node causes the reallocation of κn other nodes. The operability then becomes $1 - \frac{\alpha_t n \cdot \kappa n}{n} = 1 - \kappa\alpha_t n$. As operability cannot be smaller than 0 in reality, operability is $\max(1 - \kappa\alpha_t n, 0)$. We cannot determine the operability of Chainspace, as Chainspace does not specify how many nodes can propose to change their shards. Monoxide and Ethereum 2.0 have the operability of 1, as nodes in Monoxide and Ethereum 2.0 never move to other shards.

4.4 Observation and insights

Table 4.1 shows that no shard allocation protocols achieves optimal self-balance and operability simultaneously. We formally prove that achieving optimal values on both of them is impossible. We then identify a new property *memory-dependency* that enables parameterising the trade-off between them, opening a new in-between design space configurable for different application scenarios.

4.4.1 Impossibility and trade-off

According to Table 4.1, except for Omniledger and RapidChain, self-balance μ is either $1-\beta_t$ or 1, and operability γ is either 1 or $\frac{1}{m}$. In fact, achieving optimal self-balance and operability simultaneously still remains as an open problem, and has been extensively discussed in the blockchain community [78], [79]. We prove that, however, this is *impossible* for any correct shard allocation protocol. The proof starts from analysing the relationship between self-balance μ and operability γ. Lemma 4.4.1 formally states the relationship.

Lemma 4.4.1. *If a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards satisfies update-randomness with γ, the self-balance of $\Pi_{\mathsf{ShardAlloc}}$ is $\mu = 1 - \left|\frac{(\gamma m-1)\beta_t}{m-1}\right|$, where β_t is the percentage of nodes leaving the network in epoch t.*

Proof. By Definition 4.2.10, in epoch t, the number n_k^t of nodes in any shard k is $\frac{n^t}{m}$. By join-randomness, newly joined nodes will be uniformly allocated into shards. Thus, without the loss of generality, we assume at the end of epoch t, no node joins the network ($\alpha = 0$) and $\beta_t n^t$ nodes leave the network. Let Δn_k^t be the number of leaving nodes in shard $k \in [m]$ in epoch t, we have $\sum_{k=1}^{m} \Delta n_k^t = \beta_t n^t$. Upon the next system state st_{t+1}, each node executes $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, and its resulting shard complies with the probability distribution in Definition 4.2.4. After executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, there are some nodes in shard k moving to other shards, and there are some nodes from other shards moving to shard k as well.

By the definition of operability, there are $\gamma(n_k^t - \Delta n_k^t)$ nodes in shard k that do not move to other shards. There are

$$(1-\beta_t)n^t - (n_k^t - \Delta n_k^t) \tag{4.4}$$

nodes that do not belong to shard k. By Definition 4.2.4, there are

$$\frac{1-\gamma}{m-1}[(1-\beta_t)n^t - (n_k^t - \Delta n_k^t)] \tag{4.5}$$

nodes moving to shard k. Thus, the number n_k^{t+1} of nodes in shard k in epoch $t+1$ is

$$\begin{aligned} n_k^{t+1} &= \gamma(n_k^t - \Delta n_k^t) + \frac{1-\gamma}{m-1}[(1-\beta_t)n^t - (n_k^t - \Delta n_k^t)] \qquad (4.6)\\ &= \frac{\gamma m - 1}{m-1}(n_k^t - \Delta n_k^t) + \frac{(1-\gamma)(1-\beta_t)}{m-1}n^t \qquad (4.7)\end{aligned}$$

By Definition 4.2.10, to find μ, we should find the largest $\frac{|n_i^{t+1}-n_j^{t+1}|}{n^t}$, which can be calculated as

$$\begin{aligned} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t} &= \frac{|\frac{\gamma m-1}{m-1}(n_i^t - \Delta n_i^t) - \frac{\gamma m-1}{m-1}(n_j^t - \Delta n_j^t)|}{n^t} \qquad (4.8)\\ &= \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t - \Delta n_j^t)|}{n^t} \qquad (4.9)\end{aligned}$$

Thus, when $(\Delta n_i^t - \Delta n_j^t)$ is maximal, $\frac{|n_i^{t+1}-n_j^{t+1}|}{n^t}$ is maximal, and μ can be calculated. As there are $\beta_t n^t$ nodes leaving the network in total, the maximal value of $(\Delta n_i^t - \Delta n_j^t)$ is $\beta_t n^t$. Therefore, μ can be calculated as

$$\begin{aligned} \mu &= 1 - \max_{\forall i,j\in[m]} \frac{|n_i^{t+1} - n_j^{t+1}|}{n^t} \qquad (4.10)\\ &= 1 - \max_{\forall i,j\in[m]} \frac{|\frac{\gamma m-1}{m-1}(\Delta n_i^t - \Delta n_j^t)|}{n^t} \qquad (4.11)\\ &= 1 - \frac{|\frac{\gamma m-1}{m-1}\beta_t n^t|}{n^t} = 1 - \left|\frac{(\gamma m - 1)\beta_t}{m-1}\right| \qquad (4.12)\end{aligned}$$

□

Figure 4.3 visualises their relationship in Lemma 4.4.1. The line never reaches the point $(1, 1)$, indicating that $\Pi_{\mathsf{ShardAlloc}}$ can never achieve optimal values for them simultaneously. With operability increasing, the self-balance increases to 1 when $\gamma \leq \frac{1}{m}$, then decreases when $\gamma \geq \frac{1}{m}$. When

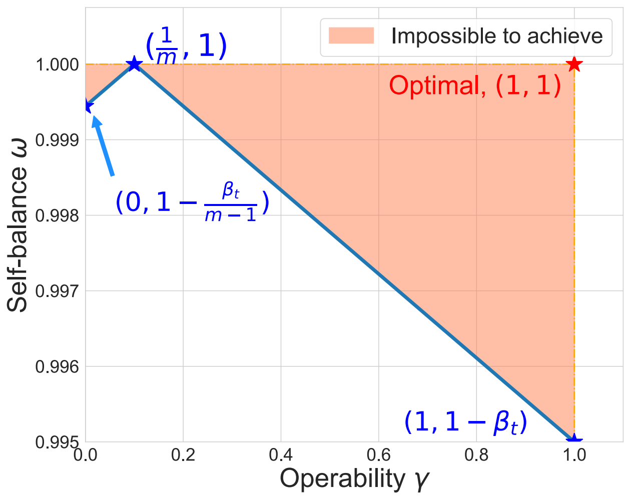


Figure 4.3: Relationship between self-balance μ and operability γ. We pick $m = 10$ and $\beta_t = 0.005$ as an example. No shard allocation protocol can go above the blue line to reach the orange area.

$\gamma = 0$, self-balance becomes $1 - \frac{\beta_t}{m-1}$. This is because when $\gamma = 0$, all nodes are mandatory to change their shards. As shard k has fewer nodes, during $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ it loses fewer nodes but receives more nodes from other shards. When $\gamma = \frac{1}{m}$, self-balance becomes 1, i.e., optimal.

Therefore, it is impossible to achieve optimal values for self-balance and operability simultaneously. Theorem 4.4.2 formally states the impossibility.

Theorem 4.4.2. *Let β_t be the percentage of nodes leaving the network in epoch t. It is impossible for a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ with m shards to achieve optimal self-balance and operability simultaneously for any $\beta_t > 0$ and $m > 1$.*

Proof. We prove this by contradiction. Assuming self-balance $\mu = 1$ and operability $\gamma = 1$. According to Lemma 4.4.1, $\mu = 1$ only when either $\beta_t = 0$ or $\gamma m = 1$. As $\gamma = 1$ and $m > 1$, $\gamma m > 1$. Thus, $\Pi_{\mathsf{ShardAlloc}}$ can achieve $\mu = 1$ and $\gamma = 1$ simultaneously only when $\beta_t = 0$. However, $\beta_t > 0$, which leads to a contradiction. □

4.4.2 Parameterising the trade-off

As shown in Figure 4.3, $(1, 1-\beta_t)$ and $(\frac{1}{m}, 1)$ are two extreme cases in the trade-off between self-balance and operability, and shard allocation protocols lying at these two points are impractical. In addition, none of our evaluated protocols allows parameterising this trade-off. We prove that, to parameterise this trade-off, sharding protocols should be *memory-dependent*, where the shard allocation result depend not only on the current system state, but also on the previous ones. In signal processing literatures, this property is also known as *non-memorylessness*, where the output signal does not only depend on the current input, but also some previous inputs [88]. Formally, memory-dependency is defined as follows.

Definition 4.4.1 (Memory-dependency)**.** A shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ is memory-dependent iff for any public parameter pp, secret key sk_i, and shard k, the output of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp,\ sk_i,\ st_t,\ k,\ \pi_{i,st_t,k},\ st_{t+1})$ depends on system states earlier than st_t.

By Definition 4.2.4, both self-balance and operability are related to the probability γ of nodes staying at the same shard. To parameterise self-balance and operability, a shard allocation protocol should incorporate shard allocation results of previous epochs. When $\gamma \in (\frac{1}{m}, 1)$, the probability distribution of allocation-randomness is non-uniform, and the membership proof of each epoch t depends on that in the previous epoch $t-1$. As the membership proof of epoch $t-1$ also depends on that of epoch $t-2$, recursively, each membership proof depends on all historical membership proofs. Thus, memory-dependency is necessary for parameterising the trade-off between self-balance and operability. Theorem 4.4.3 formally states such necessity.

Theorem 4.4.3. *If a correct shard allocation protocol $\Pi_{\mathsf{ShardAlloc}}$ is μ-self-balanced and γ-operable where $\mu \in (0, 1-\beta_t)$ and $\gamma \in (\frac{1}{m}, 1)$, then $\Pi_{\mathsf{ShardAlloc}}$ is memory-dependent.*

Proof. We prove this by contradiction. Assuming $\Pi_{\mathsf{ShardAlloc}}$ is non-memory-dependent, i.e., the output of

$$\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1})$$

only depends on st_t and st_{t+1}. This means there exists no $\delta \geq 1$ such that $\pi_{i,st_t,k}$ involves any information of $st_{t-\delta}$.

When $\gamma \in (\frac{1}{m}, 1)$, the distribution of the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is non-uniform, given the update-randomness property. In this case, executing $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ requires the knowledge of k – index of the shard that i locates at state st_t. Thus, $\pi_{i,st_{t+1},k'}$ – one of the output of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ – should enable verifiers to verify node i is at shard k in epoch t.

Verifying node i is at shard k in epoch t is achieved by verifying $\pi_{i,st_t,k}$. Thus, $\pi_{i,st_{t+1},k'}$ depends on st_t and $\pi_{i,st_t,k}$. Similarly, $\pi_{i,st_t,k}$ depends on st_{t-1} and $\pi_{i,st_{t-1},k}$, and $\pi_{i,st_{t-1},k}$ depends on st_{t-2} and $\pi_{i,st_{t-2},k}$. Recursively, $\pi_{i,st_t,k}$ depends on all historical system states. Thus, if the assumption holds, then this contradicts update-randomness. □

Remark 1. When $\gamma = \frac{1}{m}$ or 1, $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ does not rely on any prior system state. When $\gamma = \frac{1}{m}$, the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is uniformly distributed, so $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ can just assign nodes randomly according to the incoming system state. When $\gamma = 1$, the resulting shard of $\Pi_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is certain. All of our evaluated shard allocation protocols choose $\gamma = \frac{1}{m}$ or 1, except for RapidChain using Commensal Cuckoo and Chainspace allowing nodes to choose shards upon requests.

4.5 WORMHOLE: Memory-dependent shard allocation

Based on the gained insights, we propose WORMHOLE, a correct and efficient shard allocation protocol. WORMHOLE relies on a randomness beacon (RB) to generate the system states, and a verifiable random function (VRF) to guide the nodes in computing their shards. By being memory-dependent, WORMHOLE supports parameterisation of self-balance and operability. We formally analyse WORMHOLE's correctness, and its communication and computational complexity.

4.5.1 Primitives: RB and VRF

Randomness beacon. Similar to existing sharded blockchains such as Elastico, Omniledger, and Zilliqa, WORMHOLE allocates nodes based on some randomness. RB [100] is a service that periodically generates random outputs. RB is instantiated by either an external party or by a group of nodes via a decentralised randomness beacon (DRB) protocol. RB satisfies the following properties [101]:

- *RB-Availability*: No node can prevent the protocol from making progress.
- *RB-Unpredictability*: No node can know the value of the random output before it is produced.
- *RB-Unbiasibility*: No node can influence the value of the random output to its advantage.
- *RB-Public-Verifiability*: Everyone can verify the correctness of the random output.

RB schemes are both readily available and widely used. Public external RBs are maintained by countries such as the US [100], Chile [102], and Brazil [103], as well as reputable institutions such as Cloudflare [104], EPFL [105], and League of Entropy [106]. DRB protocols can be constructed from Publicly Verifiable Secret Sharing (PVSS) [36], [101], [107], Verifiable Delay Functions [108], [109], Nakamoto consensus [110], and real-world entropy [111], [112]. Several sharded blockchains, including Elastico, Omniledger, and RapidChain, employ DRB to produce the system states already; Ethereum 2.0 uses DRB for its consensus; emerging projects such as Filecoin [113] rely on an external RB for its consensus.

Verifiable random function. A VRF [114]–[116] is a public-key version of a hash function, which computes an output and a proof from an input string and a secret key. Anyone with the associated public key and the proof can verify 1) whether the output is from the input, and 2) whether the output is generated by the owner of the secret key. Some VRFs support *batch verification* [117], [118], i.e., verifying multiple VRF outputs at the same time, which is faster than verifying VRF outputs one-by-one. Formally, a VRF is a tuple of four algorithms:

- $\mathsf{VRFKeyGen}(\lambda) \rightarrow (sk, pk)$: On input a security parameter λ, outputs the secret/public key pair (sk, pk).
- $\mathsf{VRFEval}(sk, m) \rightarrow (h, \pi)$: On input sk and an arbitrary-length string m, outputs a fixed-length random output h and proof π.
- $\mathsf{VRFVerify}(pk, m, h, \pi) \rightarrow \{0, 1\}$: On input pk, m, h, π, outputs the verification result 0 or 1.
- (Optional) $\mathsf{VRFBatchVerify}(pk, \vec{\boldsymbol{m}}, \vec{\boldsymbol{h}}, \vec{\boldsymbol{\pi}}) \rightarrow \{0, 1\}$: On input pk, a series of strings $\vec{\boldsymbol{m}} = (m_1, \ldots, m_n)$, outputs $\vec{\boldsymbol{h}} = (h_1, \ldots, h_n)$, and proofs $\vec{\boldsymbol{\pi}} = (\pi_1, \ldots, \pi_n)$, outputs the verification result 0 or 1.

VRF should satisfy the following three properties [119].

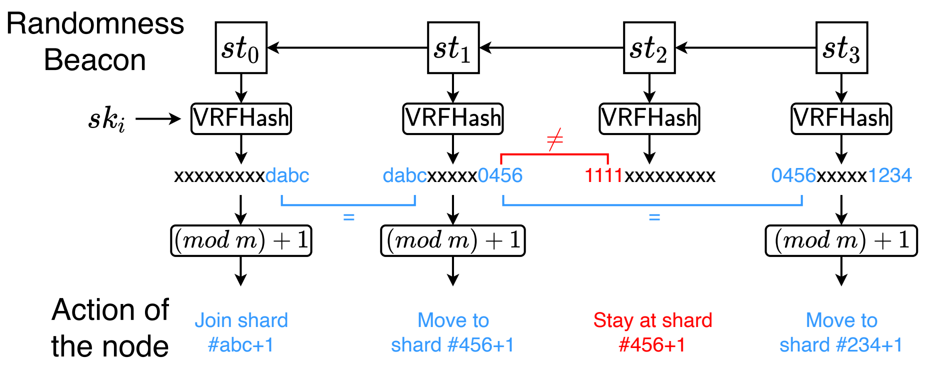


Figure 4.4: Intuition of WORMHOLE $\Pi^{\text{WH}}_{\text{ShardAlloc}}$. All numbers are in hexadecimal. We use $op = 4$ and $m = 16^3$ as an example, and assume epoch 0 is the last non-memory-dependent epoch.

- *VRF-Uniqueness*: It is computationally hard to find $(pk, m, h, h', \pi, \pi')$ such that $h \neq h'$ and $\mathsf{VRFVerify}(pk, m, h, \pi) = \mathsf{VRFVerify}(pk, m, h', \pi') = 1$.
- *VRF-Collision-Resistance*: It is computationally hard to find (m, m') such that $h = h'$ where $(h, \cdot) \leftarrow \mathsf{VRFEval}(sk, m)$ and $(h', \cdot) \leftarrow \mathsf{VRFEval}(sk, m')$.
- *VRF-Pseudorandomness*: It is computationally hard to distinguish the random output of $\mathsf{VRFEval}(\cdot)$ from a random string without the knowledge of the corresponding public key and the proof.

4.5.2 Key challenge and strawman designs

The key challenge in designing a memory-dependent shard allocation protocol is the *recursive dependency* problem: A shard membership proof in epoch t needs to prove its shard membership in epoch $t-1$ (i.e., "the memory"); and the shard membership proof in epoch $t-1$ needs to prove that in epoch $t-2$, and so on. Therefore, an extra mechanism is necessary to bound the number of history proofs.

A strawman design is to prescribe a fixed number of history proofs, so that all shard allocations but the earliest one is verifiable. However, this approach allows the adversary to enumerate all the shards as the earliest shard, and only releases one that leads them to the target shard, similar to the well-known *grinding attack* [89], [120] against proof-of-stake protocols.

Another strawman design is to periodically discard history proofs, so that nodes only need to provide history proofs up to the last non-memory-dependent epoch. Let each unit with w epochs be an *era*, which begins when $t \mod w = 0$ and ends when $t \mod w = w - 1$, where t is the epoch number. At each era's beginning, a node discards all history proofs, and computes the shard membership in a non-memory-dependent way, i.e., only based on its secret key and the current system state. This bounds the number of history proofs, but all nodes are likely to be allocated to new shards at each era's beginning, lowering the operability significantly for one epoch.

Algorithm 2: Full construction of WORMHOLE $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$.

```
Algorithm calcShard ($m, op, h_x, h_{x+1}, \ldots, h_y$):
    idx $\leftarrow x$
    for $j \in [x+1, y]$ do
        if $\mathsf{MSB}(op, h_j) = \mathsf{LSB}(op, h_{idx})$ then
            idx $\leftarrow j$ // Can be cached
    $shard\_id \leftarrow (h_{\text{idx}} \mod m) + 1$
    return $shard\_id$

Algorithm calcNMDEpoch ($w, sk_i, st_t$):
    // NMD = non-memory-dependent
    $t_{\text{era}} \leftarrow t - (t \mod w)$
    $t^-_{\text{era}} \leftarrow t_{\text{era}} - w$
    $g^-_{i,t}, \pi^-_{i,t} \leftarrow \mathsf{VRFEval}(sk_i, st^-_{\text{era}})$
    $t^-_{\text{nmd}} \leftarrow t^-_{\text{era}} + (g^-_{i,t} \mod w)$
    $g_{i,t}, \pi_{i,t} \leftarrow \mathsf{VRFEval}(sk_i, st_{\text{era}})$
    $t_{\text{nmd}} \leftarrow t_{\text{era}} + (g_{i,t} \mod w)$
    last $\leftarrow t^-_{\text{nmd}} < t < t_{\text{nmd}}\ ?\ t^-_{\text{nmd}} : t_{\text{nmd}}$
    return $(last, (g^-_{i,t}, \pi^-_{i,t}, g_{i,t}, \pi_{i,t}))$

Algorithm Setup ($\lambda$):
    $m, op, w \leftarrow \lambda$
    return $(m, op, w)$
```

```
Algorithm Join ($pp, sk_i, st_t$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}) \leftarrow \mathsf{calcNMDEpoch}(w, sk_i, st_t)$
    for $j \in [last, t]$ do
        $h_j, \pi_j \leftarrow \mathsf{VRFEval}(sk_i, st_j)$
    $k \leftarrow \mathsf{calcShard}(m, op, h_{\text{last}}, \ldots, h_t)$
    $\pi_{i,st_t,k} \leftarrow (\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t)$
    Store $\pi_{i,st_t,k}$ in memory
    return $k, \pi_{i,st_t,k}$

Algorithm Update ($pp, sk_i, st_t, k, \pi_{i,st_t,k}, st_{t+1}$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
    $(\text{last}^+, \pi^+_{\text{range}}) \leftarrow \mathsf{calcNMDEpoch}(w, sk_i, st_{t+1})$
    Remove $(h_j, \pi_j)$ from memory for
      $j \in [\text{last}, \text{last}^+)$
    $h_{t+1}, \pi_{t+1} \leftarrow \mathsf{VRFEval}(sk_i, st_{t+1})$
    $k' \leftarrow \mathsf{calcShard}(m, op, h_{\text{last}^+}, \ldots, h_{t+1})$
    $\pi_{i,st_{t+1},k'} \leftarrow$
      $(\text{last}^+, \pi^+_{\text{range}}, h_{\text{last}^+}, \ldots, h_{t+1}, \pi_{\text{last}^+}, \ldots, \pi_{t+1})$
    Store $\pi_{i,st_{t+1},k'}$ in memory
    return $k', \pi_{i,st_{t+1},k'}$
```

```
Algorithm Verify ($pp, pk_i, st_t, k, \pi_{i,st_t,k}$):
    $m, op, w \leftarrow pp$
    $(\text{last}, \pi_{\text{range}}, h_{\text{last}}, \ldots, h_t, \pi_{\text{last}}, \ldots, \pi_t) \leftarrow \pi_{i,st_t,k}$
    $(g^-_{i,t}, \pi^-_{i,t}, g_{i,t}, \pi_{i,t}) \leftarrow \pi_{\text{range}}$
    $t_{\text{era}} \leftarrow t - (t \mod w)$
    $t^-_{\text{era}} \leftarrow t_{\text{era}} - w$
    $t^-_{\text{nmd}} \leftarrow t^-_{\text{era}} + (g^-_{i,t} \mod w)$
    $t_{\text{nmd}} \leftarrow t_{\text{era}} + (g_{i,t} \mod w)$
    // Verify memory range
    if $\boxed{\begin{array}{ll} t^-_{nmd} < t < t_{nmd} \land last \neq t^-_{nmd} & \lor \\ t_{nmd} \leq t \land last \neq t_{nmd} & \lor \\ \mathsf{VRFVerify}(pk_i, st^-_{era}, g^-_{i,t}, \pi^-_{i,t}) = 0 & \lor \\ \mathsf{VRFVerify}(pk_i, st_{era}, g_{i,t}, \pi_{i,t}) = 0 & \end{array}}$ then
        return 0
    $\vec{st}, \vec{h}, \vec{\pi} \leftarrow$
      $(st_{\text{last}}, \ldots, st_t), (h_{\text{last}}, \ldots, h_t), (\pi_{\text{last}}, \ldots, \pi_t)$
    if $\mathsf{VRFBatchVerify}(pk_i, \vec{st}, \vec{h}, \vec{\pi}) = 0$ then
        return 0 // Can be cached
    if $k \neq \mathsf{calcShard}(m, op, h_{last}, \ldots, h_t)$ then
        return 0
    return 1
```

4.5.3 The WORMHOLE design

WORMHOLE addresses the above challenge by (1) prescribing a non-memory-dependent shard allocation per node per era and (2) randomising this non-memory-dependent epoch for each node, so that the size of a membership proof is bounded and nodes discard history proofs in different epochs. Algorithm 2 provides the full construction of WORMHOLE.

Each node i determines the non-memory-dependent epoch and the allocated shard in this epoch by using $\mathsf{calcNMDEpoch}(\cdot)$. When an era starts at epoch t (when $t \mod w = 0$), node i calculates $\mathsf{VRFEval}(sk_i, st_t) \rightarrow (g_{i,t}, \pi_{i,t})$, where st_t is RB's output, i.e., the system state, in epoch t. Then at epoch $t + (g_{i,t} \mod w)$, the node will remove all the memory and move to shard $k = (g_{i,t} \mod m) + 1$. Note that both the reallocation epoch and the allocated shard are non-memory-dependent, and this happens exactly once per era.

The other $w - 1$ epochs are memory-dependent, and each node i determines the allocated shard by using $\mathsf{calcShard}(\cdot)$. At epoch t, node i computes $\mathsf{VRFEval}(sk_i, st_t) \rightarrow (h_{i,t}, \pi_{i,t})$. Let op be the parameter for parameterising operability (and self-balance). Let $\mathsf{LSB}(x, m)$ and $\mathsf{MSB}(x, m)$ be the least and most significant x bits of m, respectively. Node i stays in the same shard, i.e., $k_{i,t} = k_{i,t-1}$ if $\mathsf{LSB}(op, h_{i,t-1}) \neq \mathsf{MSB}(op, h_{i,t})$, otherwise moves to shard $k_{i,t} = (h_{i,t} \mod m) + 1$. This injects the memory-dependency to the shard memberships of two consecutive epochs. Increasing op improves operability but reduces self-balance, and vice versa. Figure 4.4 illustrates this idea.

To join the system, a node i executes $\mathsf{Join}(\cdot)$: It calculates VRF outputs and proofs since the last non-memory-dependent epoch, and executes $\mathsf{calcShard}(\cdot)$ to calculate its allocated shard k. The shard membership proof $\pi_{i,st_t,k}$ includes a sequence of VRF outputs $(h_{\text{last}}, \ldots, h_t)$ and their VRF proofs $(\pi_{\text{last}}, \ldots, \pi_t)$, where $last$ is the last non-memory-dependent epoch calculated from $\mathsf{calcNMDEpoch}(\cdot)$.

Upon epoch $t+1$, node i executes $\mathsf{Update}(\cdot)$ as follows. It first calculates the VRF output of st_{t+1}. If epoch $t+1$ is memory-dependent, then $\mathsf{calcShard}(\cdot)$ only needs to check if $\mathsf{MSB}(op, h_{t+1}) = \mathsf{LSB}(op, h_{\text{idx}})$ and compute idx and shard_id accordingly, where h_{idx} is cached from epoch t. If epoch $t+1$ is non-memory-dependent, then the previous proofs are discarded and the shard ID is $(h_{t+1} \mod m) + 1$.

To verify proof $\pi_{i,st_t,k}$, $\mathsf{Verify}(\cdot)$ uses $\mathsf{calcNMDEpoch}(\cdot)$ to verify the last non-memory-dependent epoch, uses $\mathsf{VRFBatchVerify}(\cdot)$ to verify VRF outputs, and uses $\mathsf{calcShard}(\cdot)$ over these VRF outputs to verify its output against k. Previous verification results can be cached and reused: Upon an updated membership proof $\pi_{i,st_{t+1},k'}$, the verifier can reuse most of the results in verifying $\pi_{i,st_t,k}$, including verification results of previous VRF outputs and $\mathsf{calcShard}(\cdot)$.

Construction without allocation-privacy. As mentioned in §4.2.3, allocation-privacy is not always a desired property. To remove allocation-privacy in $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$, one can replace $\mathsf{VRFEval}(sk_i, st_t)$ with $H(pk_i || st_t)$, where sk_i and pk_i are key pairs of node i, st_t is the system state, and $H(\cdot)$ is a cryptographic hash function.

4.5.4 Theoretical analysis

Correctness. We first summarise the security analysis. WORMHOLE satisfies liveness as a node can compute $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$ locally. WORMHOLE satisfies unbiasibility, as $\mathsf{VRFEval}(\cdot)$ and $\mathsf{calcShard}(\cdot)$ are deterministic functions, and system states are unbiasible, guaranteed by RB. WORMHOLE satisfies join-randomness, as VRF produces uniformly distributed outputs. When the epoch is a memory-dependent epoch, the probability that two random outputs share the same op-bit substring is $\frac{1}{2^{op}}$. Within the probability $\frac{1}{2^{op}}$, the probability that two random outputs result in the same shard is $\frac{1}{m}$. This leads to $\gamma = 1 - \frac{1}{2^{op}} \cdot \frac{m-1}{m} = 1 - \frac{m-1}{m \cdot 2^{op}}$. When the epoch is a non-memory-dependent epoch, the node will be shuffled, leading to $\gamma = \frac{1}{m}$. Thus, WORMHOLE satisfies allocation-randomness. WORMHOLE satisfies allocation-privacy, as one cannot compute $\mathsf{Join}(\cdot)$ or $\mathsf{Update}(\cdot)$ for a node without knowing its secret key. The probability of guessing shard allocation follows the proof of allocation-randomness.

Then, we provide the formal proof below.

Lemma 4.5.1. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies liveness.*

Proof. By RB-Availability, the RB is always producing random outputs, and therefore new system states regularly. Given a new system state, any honest node can execute $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ (or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ for newly joined nodes). As both $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ and $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ can be computed locally without interacting with other nodes, the execution of them will eventually terminate. $\square$

Lemma 4.5.2. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies unbiasibility.*

Proof. We prove this by contradiction. Assuming that $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy unbiasibility: Given a system state, an adversary can manipulate the probability distribution of the output shard of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $.\mathsf{Update}(\cdot)$ with non-negligible probability. This consists of three attack vectors: 1) the adversary can manipulate the system state; 2) when $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ are probabilistic, the adversary can keep generating memberships until outputting a membership of its preferred shard; and 3) the adversary can forge proofs of memberships of arbitrary shards.

By RB-Unbiasibility, the randomness produced by RB is unbiasible, so the system state of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ is unbiasible. By VRF-Collision-resistance, given a secret key, the VRF output of the system state is unique except for negligible probability. This eliminates the last two attack vectors and ensures that the VRF output of the unbiasible system state is unbiasible. The output shard of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ or $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$ is a modulus of the

VRF output, which is also unbiasible. This eliminates the first attack vector. Thus, if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not resist against the first attack vector, then this contradicts RB-Unbiasibility; and if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not resist against the second and/or the last attack vectors, then this contradicts VRF-Collision-resistance. $\square$

Lemma 4.5.3. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies join-randomness.*

Proof. We prove this by contradiction. Assuming that $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy join-randomness, i.e., the probabilistic of a node joining a shard $k \in [m]$ is $\frac{1}{m} + \epsilon$ for some k and non-negligible ϵ. Running $\mathsf{Join}(\cdot)$ requires the execution of $\mathsf{VRFEval}(\cdot)$ over a series of system states. By VRF-Pseudorandomness, VRF outputs of system states are pseudorandom. As a modulo of a VRF output, the output shard of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Join}(\cdot)$ is also pseudorandom. Thus, if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy join-randomness, then this contradicts VRF-Pseudorandomness. $\square$

Lemma 4.5.4. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies update-randomness.*

Proof. We prove this by contradiction. Assuming that $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy update-randomness, i.e., with non-negligible probability, there is no γ such that the probability of a node joining a shard k complies with the distribution in Definition 4.2.4. When epoch t is a non-memory-dependent epoch, the node will be shuffled. By VRF-Pseudorandomness, the probability of moving to each shard is same. Thus, there is a $\gamma = \frac{1}{m}$ that makes the output shard of $\mathsf{Update}(\cdot)$ to comply with the distribution in Definition 4.2.4.

When t is a memory-dependent epoch, the last VRF output remains unchanged. In $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}.\mathsf{Update}(\cdot)$, given the last VRF output, the probability that the op MSBs of the new VRF output equal to op LSBs of the last VRF output is $\frac{1}{2^{op}}$. By VRF-Pseudorandomness, the probability of moving to each other shard is same. Thus, there is a $\gamma = 1 - \frac{1}{2^{op}} \cdot \frac{m-1}{m} = 1 - \frac{m-1}{m \cdot 2^{op}}$ that makes the output shard of $\mathsf{Update}(\cdot)$ to comply with the distribution in Definition 4.2.4.

Table 4.2: Evaluation of shard allocation protocols that replace DRG with a randomness beacon. Meanings of colours are same as Table 4.1. ★ means the metric is improved by replacing DRG with a randomness beacon.

		System model			Correctness					Performance metrics			
	State update	Network model	Trusted components	Fault tolerance	Public verifiability	Liveness	Allocation-rand.	Unbiasibility	Privacy^o	Join comm. compl.	Update comm. compl.	Self-balance	Operability
Elastico	New block	Async.★	Rand. Beacon*	Ψ★	✓	✓	✓	✗	✓	$O(n)$★	$O(n)$★	1	$\frac{1}{m}$
Omniledger	New block	Part. sync.	Rand. Beacon*	Ψ★	✓	✓★	✓	✓	✗	$O(n)$	$O(n)$★	$1-\frac{(2m-3)\beta_t}{3m-3}$	$\frac{2}{3}$
RapidChain	Nodes joining	Async.★	Rand. Beacon*	Ψ★	✗	✓	✓	✓	✗	$O(n)$★	$O(n)$★	$1-\beta_t$	$max(1-\kappa\alpha_t n, 0)$
Zilliqa	New block	Async.★	Rand. Beacon*	Ψ	✓	✓	✓	✗	✓	$O(n)$	$O(n)$	1	$\frac{1}{m}$
WORMHOLE (§4.5)	New rand.	Async.	Rand. Beacon*	Ψ	✓	✓	✓	✓	✓	$O(n)$	$O(n)$	$1-\beta_t+\frac{\beta_t}{2^{op}}$	$1-\frac{m-1}{m\cdot 2^{op}}$

^{o} Optional. $*$ Shard allocation protocols can rely on an external randomness beacon, or allow nodes to run a decentralised randomness beacon protocol. Ψ is the fault tolerance capacity of the sharded blockchain's consensus protocol.

Thus, if $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ does not satisfy update-randomness, then this contradicts VRF-Pseudorandomness. □

Lemma 4.5.5. $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ *satisfies allocation-privacy.*

Proof. This follows proofs of Lemma 4.5.3 and 4.5.4. □

Performance metrics. The communication complexity of $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$ of $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ are $O(n)$ where n is the number of nodes, as each node needs to receive a constant number of system states for executing $\mathsf{Join}(\cdot)$ and $\mathsf{Update}(\cdot)$. A $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$ proof contains $[3, 2w+2)$ VRF outputs/proofs, where w is the era length. $\mathsf{Join}(\cdot)$ invokes $\mathsf{VRFEval}(\cdot)$ for $[1, 2w)$ times, leading to computational complexity $O(w)$. $\mathsf{Update}(\cdot)$ invokes $\mathsf{VRFEval}(\cdot)$ for once, leading to computational complexity $O(1)$. $\mathsf{Verify}(\cdot)$ invokes $\mathsf{VRFVerify}(\cdot)$ for once if verification results are cached, otherwise $\mathsf{VRFBatchVerify}(\cdot)$ over $[1, 2w)$ VRF outputs/proofs, leading to computational complexity $O(1)$ or $O(w)$, respectively. By update-randomness, $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$'s operability $\gamma = 1 - \frac{m-1}{m} \cdot \frac{1}{2^{op}} = 1 - \frac{m-1}{m \cdot 2^{op}}$. By Definition 4.4.1, $\Pi^{\mathsf{WH}}_{\mathsf{ShardAlloc}}$'s self-balance $\mu = 1 - \left| \frac{(\gamma m - 1)\beta_t}{m-1} \right| = 1 - \beta_t + \frac{\beta_t}{2^{op}}$.

4.5.5 Comparison with existing protocols

Table 4.1 summarises the comparison result. It shows that WORMHOLE is the only shard allocation protocol that is fully correct and achieves satisfactory performance, without relying on strong assumptions. To make a fair comparison, we also evaluate shard allocation protocols while assuming RB, and the evaluation results are summarised in Table 4.2. Chainspace, Monoxide and Ethereum 2.0 are omitted as their shard allocation protocols do not rely on randomness.

According to Table 4.2, these proposals are improved in terms of the system model and communication complexity. All of them achieve $O(n)$ communication complexity, where the concrete overhead depends on the instantiation and implementation, including cryptographic primitives and message formats. However, they still suffer from some problems they originally have, and WORMHOLE still outperforms them. For example, among the correctness properties, only Omniledger's liveness issue is fixed. In addition, Omniledger should still assume partial synchorny, as liveness is guaranteed only under synchronous networks. To compute shard memberships, nodes have to broadcast their identifies and agree on a permutation of them, which require synchrony. Moreover, all of them still suffer from weak operability except for Omniledger.

4.6 Integration of WORMHOLE

In this section, we analyse how to integrate WORMHOLE into different sharded blockchains, and the corresponding impact on the system model and overhead.

4.6.1 Design choices related to WORMHOLE

The overhead introduced by WORMHOLE can be affected by two design choices of the sharded blockchain, namely the existence of identity registry and the choice of consensus protocol.

Existence of identity registry. Some sharded blockchains employ an identity registry that tracks identities of nodes in the system. For example, Elastico, RapidChain and Zilliqa require a special shard to be the identity registry; Omniledger instantiates the Sybil-resistant mechanism by using a trusted identity authority; and Chainspace requires nodes to maintain a special smart contract managing identities.

The existence of identity registry decides where a shard membership is verified and stored. If the sharded blockchain employs an identity registry, then the identity registry can maintain and verify all shard memberships, and a node can query other nodes' shard memberships over the identity registry. Without an identity registry, a node then has to receive and verify other nodes' shard memberships when executing other subprotocols (e.g., consensus).

Choice of consensus protocol. Existing research [121], [122] suggests to classify consensus protocols into two types, namely BFT-style consensus and Nakamoto-style consensus. In BFT-style consensus, given the latest blockchain, nodes propose blocks, vote to agree on a unique block, and append the agreed block to the blockchain. In Nakamoto-style consensus, given the latest blockchain, nodes compete to solve a cryptographic puzzle. If a node solves a puzzle, then it can append a new block associated to the puzzle solution to the blockchain. Nodes follow a chain selection rule to decide the main chain among all forks, and eventually the main chains of different nodes converge to the same one.

The choice of consensus protocol decides when a shard membership is queried or verified. With BFT-style consensus, a node has to additionally verify a quorum of nodes' shard memberships for each block. With Nakamoto-style consensus, a node has to additionally verify the block proposer's shard membership for each block.

Classification of sharded blockchains. Our evaluated sharded blockchains only belong to two "identity registry + consensus" combinations: Elastico, Omniledger, Chainspace, Zilliqa and Ethereum 2.0 employ an identity registry and BFT-style consensus; and Monoxide employs Nakamoto-style consensus without an identity registry.

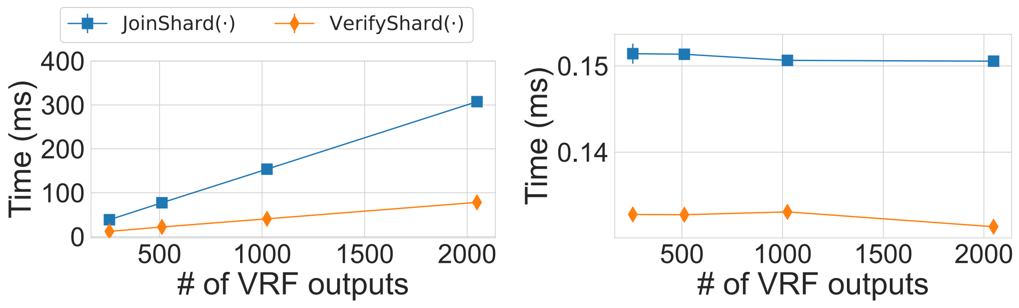
4.6.2 Integration analysis

We then analyse how to integrate WORMHOLE into the two cases.

With identity registry, BFT-style consensus. In this case, every node executes WORMHOLE to obtain a shard membership with proof and submits them to the identity registry for verification. For each new epoch, a node needs to compute a VRF output with proof and send them to the identity registry. The identity registry needs to send each node the set of all peers' identities and the shard size. Every node then executes BFT-style consensus with peers to agree on blocks. For each vote, a node looks up the the voter node's identity within the set. A block needs to obtain a quorum of votes to be valid. The identity set can be replaced with a cryptographic accumulator [123], where the size and the lookup complexity can be sublinear w.r.t. the set size. The identity registry also manages nodes' identities and handles Sybil attacks.

With this approach, the sharded blockchain inherits the network model and fault tolerance capacity from its underlying consensus protocol, and incurs some extra overhead as follows. The identity registry needs to additionally receive, store and verify shard memberships and proofs for all nodes. For each new epoch, each node needs to additionally submit a VRF output and proof to the identity registry and receive the set of identities and an integer, while the identity registry needs to verify a VRF output and update the shard membership. For each block, each node needs to look up a quorum of nodes' shard memberships within the set.

No identity registry, Nakamoto-style consensus. In this case, every node executes WORMHOLE to obtain a shard membership with proof, and keeps solving puzzles to propose blocks over the main chain decided by the chain



(a) Newly joined nodes. (b) Existing nodes.

Figure 4.5: Computation overhead of WORMHOLE.

selection rule. Each block additionally attaches the miner's shard membership and proof. Upon receiving a block, the node additionally verifies the miner's shard membership. Similar to Elastico, Chainspace, Zilliqa and Ethereum 2.0, a node has to solve a cryptographic puzzle in order to obtain an identity in the system. To support permissionless settings, the puzzle's difficulty is controlled by a difficulty adjustment mechanism.

The Nakamoto-style consensus will require WORMHOLE to assume a synchronous network and the fault tolerance capacity depending on the concrete Sybil-resistance mechanism, as analysed by Dembo et al. [89]. Nodes need to possess the dedicated resource w.r.t. the Sybil-resisitance mechanism in Nakamoto-style consensus. In addition, for every block, a node needs to additionally receive, store and verify a shard membership and proof.

4.7 Evaluation of WORMHOLE

In this section, we implement WORMHOLE and evaluate its overhead and performance metrics in the wild. The evaluation results show that WORMHOLE introduces little overhead and achieves performance metrics consistent with the theoretical values.

4.7.1 Overhead analysis

Implementation and experimental setup. We implement WORMHOLE in Rust. We use `rug` [124] for large integer arithmetic and `bitvec` [125] for bit-level operations. We use `w3f/schnorrkel` [126], which implements the standardised VRF [116] over the Curve25519 elliptic curve with Ristretto compressed points [127] and the Schnorr-style aggregatable discrete log equivalence proofs (DLEQs) [118] for batch verification. The size of keys, VRF outputs and proofs are 32, 32 and 96 Bytes, respectively. System states are simulated by `rand` [128]. We write the benchmarks using `cargo-bench` [129] and `criterion` [130]. We specify the O3-level optimisation for compilation, and sample 20 executions for each unique group of parameters. All experiments were conducted on a MacBook Pro with a 2.2 GHz 6-Core Intel i7 processor and a 16 GB RAM.

Benchmarks results. We benchmark $\mathsf{Join}(\cdot)$, $\mathsf{Update}(\cdot)$ and $\mathsf{Verify}(\cdot)$ for WORMHOLE. Recall that with era length w, a node reaches a non-memory-dependent epoch for every w epochs on average. We choose w ranging from 256 to 2048 epochs. In Bitcoin's setting where a block is generated for every ten minutes, 256 and 2048 epochs take about 2 and 14 days, respectively.

Figure 4.5 shows the results. For newly joined nodes, the execution time of $\mathsf{Join}(\cdot)$ and $\mathsf{Verify}(\cdot)$ increases linearly with the number of random outputs. With 256 random outputs, $\mathsf{Join}(\cdot)$ and $\mathsf{Verify}(\cdot)$ take 39 and 12 ms, respectively. With 2048 random outputs, $\mathsf{Join}(\cdot)$ and $\mathsf{Verify}(\cdot)$ take 300 and 90 ms, respectively. For existing nodes, $\mathsf{Update}(\cdot)$ and $\mathsf{Verify}(\cdot)$ take about 0.15 and 0.13 ms, respectively. A shard membership takes at most 4 Bytes, which can support 2^{32} shards. As VRF outputs and proofs are 32 and 96 Bytes, a membership proof size S_π is $(32 + 96) * 2w = 256w$ Bytes. The size S_π is then 64 and 512 KB with $w = 256$ and 2048, respectively; and updating a membership proof takes 128 Bytes.

Overhead of integration. We analyse the concrete overhead of integrating WORMHOLE into two types of sharded blockchains in §4.6.2 separately. When employing an identity authority and BFT-style consensus, the identity registry needs to receive, store and verify shard memberships and proofs. This incurs one-time overhead of $S_\pi * n$ on storage and communication, and n non-cached $\mathsf{Verify}(\cdot)$ invocations. For each epoch, each node sends a VRF output and proof to the identity registry, the identity registry verifies it, and sends back the set of identities and the shard size. Thus, each node sends 128 Bytes and receives $32 * \frac{n}{m}$ Bytes, and the identity registry invokes a cached $\mathsf{Verify}(\cdot)$ once for each node. If replacing the set with a constant-size accumulator of s Bytes, then the per-node communication overhead can be reduced to $s+128$ Bytes. For each block, each node has to look up a quorum of nodes' identities, which introduces computation overhead of $\frac{n}{m}$ lookup operations.

When employing Nakamoto-style consensus without identity authority, for each block, a node needs to additionally receive, store and verify a shard membership and proof, incurring the communication and storage overhead of S_π and the computation overhead of a non-cached $\mathsf{Verify}(\cdot)$ invocation.

4.7.2 Simulation

We then simulate WORMHOLE in a network with 128 shards and 32768 nodes, confirming the theoretical results on WORMHOLE's self-balance and operability guarantees in §4.5.4.

Evaluation criteira. The simulation aims at observing the load balance and operability in the real-world setting and comparing them with the theoretical analysis in §4.5.4.

Following existing distributed systems research [131], the observed load balance is quantified as the *coefficient of variation (CV)*, namely the ratio between the standard deviation $std(\cdot)$ and the mean value $mean(\cdot)$ of the node distribution across shards. Specifically, the observed load balance in epoch t is $\frac{std(\mathcal{N}^t)}{mean(\mathcal{N}^t)}$, where $\mathcal{N}^t = \{n_k^t\}_{k\in[m]}$ is the number n_k^t of nodes in every shard k in epoch t. When CV is zero, then the system achieves optimal load balance, where every shard contains the same number of nodes. When CV is smaller than 1, then it means the distribution is low-variance and the system achieves satisfactory load balance.

The observed operability is quantified as the ratio between the number of moved nodes and the number of existing nodes. Specifically, the observed operability in epoch t is $1 - \frac{n^t_{\text{moved}}}{n^t}$, where n^t and n^t_{moved} are the total number of nodes and the number of moved nodes in epoch t, respectively.

Simulation setup. We simulate WORMHOLE with $m = 128$ shards, $n = 128 * 256 = 32768$ nodes, $w = 2048$, and operability degree $\gamma = 0.95$ over 500 epochs with variant churn rate distribution. As there is no data available on the shard memberships of sharded blockchains, we align the simulated churn rate distribution to Bitcoin, where both the join rate α and leave rate β in 2021 are about 0.1 per day according to recent measurement studies [132], [133]. Depicted in Figure 4.6(a), we simulate the following scenarios. (1) Epoch 1-50: $\alpha \in [0, 0.2]$, $\beta \in [0.09, 0.11]$, epoch 101-150: $\alpha \in [0.09, 0.11]$, $\beta \in [0, 0.2]$, and epoch 201-250: $\alpha, \beta \in [0, 0.2]$. This scenario evaluates WORMHOLE's resilience against volatile join and leave churn rates. (2) Epoch 301-350:$\alpha \in [0.04, 0.06]$, $\beta \in [0.09, 0.11]$, which evaluates WORMHOLE's resilience against the case of $\alpha < \beta$, which affects self-balance and operability as analysed in Lemma 4.4.1. (3) Epoch 401-450: $\alpha \in [0.09, 0.11]$, $\beta \in [0.04, 0.06]$, which evaluates WORMHOLE's resilience against the case of $\alpha > \beta$. Other epochs are configured with $\alpha, \beta \in [0.09, 0.11]$ to allow the network to recover and avoid influence between the above epoch executions.

Simulation results (Figure 4.6). Figure 4.6(b) outlines the distribution of nodes. Figure 4.6(c) shows the observed load balance, in both best-case and

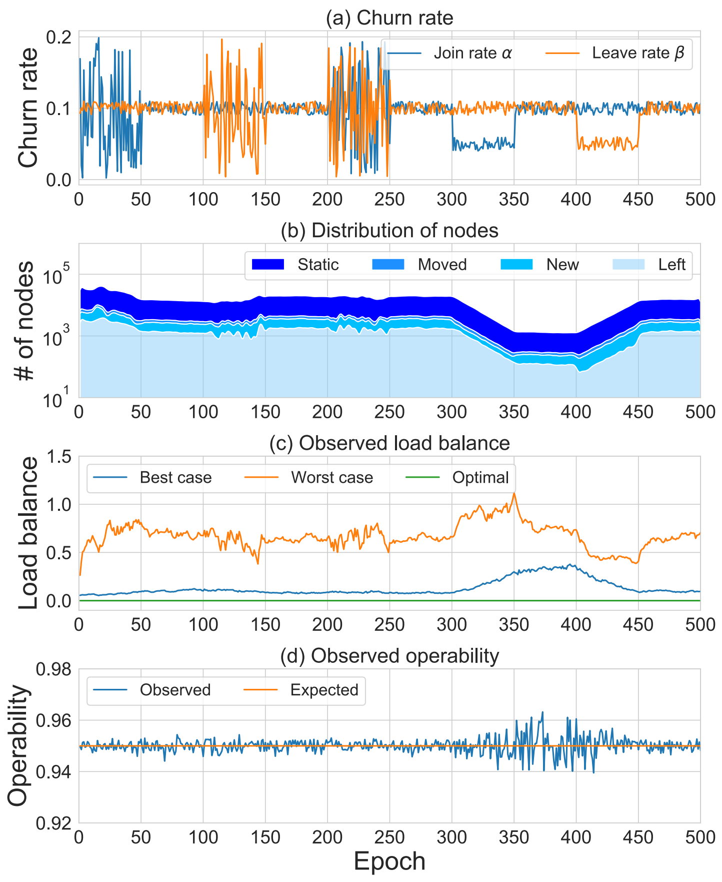


Figure 4.6: Simulation results of WORMHOLE over 500 epochs (x axis) in different churn rates. **(a)** Simulated churn rate (α, β) over epochs. **(b)** Distribution of nodes over epochs. A node is static if it stays in the same shard compared to the last epoch; is moved if it is allocated to another shard compared to the last epoch; is new if it newly joins the system in this epoch; and is left if it leaves the system in this epoch. **(c)** Observed load balance in the best-case and worst-case execution. In the best case, a random set of nodes leave the system, while in the worst case nodes in the same shard leave the system. **(d)** Observed operability compared with the expected one.

worst-case execution. In the best-case execution, a random set of nodes leave the network, and each shard is likely to lose a similar number of nodes. In the worst-case execution, nodes in the same shard leave the network, making the shards less balanced. We observe that in epoch 1-300 where the average join rate $\bar{\alpha}$ equals to the average leave rate $\bar{\beta}$, the observed load balance is about 0.1 and 0.6 in the best-case and worst-case execution, respectively. In epoch 301-350 where $\bar{\alpha} < \bar{\beta}$, the observed load balance increases to 0.5 and 1.1 in the best-case and worst-case execution, respectively. The observed load balance is less than 1 in most cases, meaning that WORM-

HOLE achieves satisfactory load balance guarantee under high leave rate. In addition, in epoch 351-400 where $\bar{\alpha} = \bar{\beta}$ again, the observed load balance in the worst-case execution reduces from 1.1 to 0.8 monotonically within about 25 epochs. This shows that WORMHOLE can recover from temporary load imbalance in a short time period. Moreover, in epoch 401-450 where $\bar{\alpha} > \bar{\beta}$, the observed load balance reduces further by 0.1 in both the best-case and worst-case execution. This is because newly joined nodes are uniformly distributed among shards, amortising the load imbalance.

Figure 4.6(d) shows the observed operability. We observe that while the expected operability is 0.95, the observed operability is 0.95 ± 0.01, meaning that WORMHOLE can achieve the parameterised operability with little bias. In epoch 351-400 where $\bar{\alpha}$ recovers to be equal to $\bar{\alpha}$, the maximum bias remains stable rather than recovering to that in epoch 1-300. This is because the number of nodes has been reduced, making the statistical results more volatile. In epoch 401-450 where $\bar{\alpha} > \bar{\beta}$, the observed operability recovers to that in epoch 1-300. This is also because newly joined nodes are uniformly distributed among shards.

4.8 Related work

We briefly review existing research on sharding distributed systems and compare our contributions with two studies systematising blockchain sharding protocols.

Sharding for CFT distributed systems. Sharding has been widely deployed in crash fault tolerant (CFT) systems to raise their throughput. Allocating nodes to shards in a CFT system is straightforward, as there is no Byzantine adversaries in the system, and the total number of nodes is fixed and known to everyone [70], [134], [135]. The main challenge is to balance the computation, communication, and storage workload among shards. Despite a large number of load-balancing algorithms [71]–[74], [136], none of them is applicable in the permissionless setting as they do not tolerate Byzantine faults.

Distributed Hash Tables. Many peer-to-peer (P2P) storage services [82], [83] employ Distributed Hash Tables (DHT) [137] to assign file metadata, i.e., a list of keys, to their responsible nodes. In a DHT, nodes share the same ID space with the keys; a file's metadata is stored at the nodes whose IDs are closest to the keys. Although designed to function in a permissionless environment, DHTs are vulnerable to several attacks [138]–[140], therefore are not suitable for blockchains, which demands strong consistency on financial data.

Distributed Slicing. Distributed Slicing [141] aims at grouping nodes with heterogeneous computing and storage capacities in a P2P network to optimise resource utilisation. In line with CFT systems, these algorithms [84]–[87] require nodes to honestly report their computing and storage capacities, therefore are not suitable in a Byzantine environment.

Evaluation of sharded blockchains. Wang et al. [10] propose an evaluation framework based on Elastico's architecture; Avarikioti et al. [80] formalise sharded blockchains by extending the model of Garay et al. [18]. Both of them aim at evaluating the entire sharded designs, and put most efforts on DRG or cross-shard communication, neglecting the security and performance challenges of shard allocation.

Chapter 5

RandChain: A scalable and fair Decentralised Randomness Beacon

5.1 Introduction

Our evaluation in Chapter 3 shows that existing permissionless sharded blockchains usually embed a Decentralised Randomness Beacon (DRB) protocol that allows nodes to jointly generate random outputs periodically. As permissionless sharded blockchains require strong security guarantee due to its high financial stake, DRBs in them have to be 1) *scalable*: Even with a large number of participants, the DRB produces random outputs with an expected rate, and 2) *fair*: Each participant controls comparable power on deciding random outputs. Without scalability, the DRB can be maintained only by a small set of participants. Without fairness, the DRB can be dominated by a small subset of powerful participants out of the entire set. When the DRB is dominated by a small set of participants, they can collude and manipulate the randomness in order to take advantage in these permissionless sharded blockchains supported by the DRB. However, designing a DRB that is both scalable and fair remains an open challenge.

Existing DRBs do not scale. Most DRB protocols are built from periodically executing a Distributed Randomness Generation (DRG) protocol, where participants contribute their local entropy and aggregate them into a single random output. Commonly used DRG protocols are based on threshold cryptosystems [99], [142], [143], Verifiable Random Functions (VRFs) [144]–[146], and/or Publicly Verifiable Secret Sharing (PVSS) [36], [101], [107], [147]–[149].

While DRG-based DRBs are fair given their "one-man-one-vote" design, they are not scalable, as they suffer from at least $O(n^2)$ communication complexity. DRG-based DRBs usually involve all-to-all broadcast primitives, leading to at least $O(n^2)$ communication complexity. To overcome the communication complexity bound, DRG-based DRBs have to employ a central point that relays messages. The central point is either a dealer [99], [107], [143], [146], [148] or a leader elected by a leader election protocol [36], [142], [144], [145], [147]. A dealer is either implemented as a trusted party or in a distributed manner which introduces extra communication overhead [150]. If the elected leader is corrupted, then it can bias random outputs by withholding messages and can compromise the liveness by sending messages to and advancing rounds for only a subset of participants [151], [152]. To tolerate corrupted leaders, the DRB has to employ an extra round synchronisation protocol [152], which allows participants to re-synchronise and replace the corrupted leader with a new leader to start a new round. However, round synchronisation protocols introduce extra communication complexity [151], [152] and/or increase latency [153].

The scalability crux: Participants are collaborative. We attribute these limitations to the design that participants are *collaborative*: Participants contribute their local inputs and aggregate them into a single output. The collaborative process ensures that no participant can fully control random outputs, making them hard to bias or predict. However, in order to collaborate, participants should continuously broadcast messages to and synchronise with each other. The former incurs at least $O(n^2)$ communication complexity, and the latter requires round synchronisation. All extra designs incorporated with DRG – e.g., using dealers [99], [107], [143], [148], leader election [36], [142], [144]–[147], sharding [36], [142], cryptographic sortition [145], Byzantine consensus [101], [145], and erasure coding [107], [148] – aim at reducing the impact of the above two limitations. However, since all of them are in the collaborative design, they inherently suffer from the two limitations and cannot address them completely.

Competitive DRBs: A new design space. To address the inherent limitations in the collaborative design, we consider a new design space for DRBs called *competitive DRBs*. Unlike existing DRBs where participants are collaborative, participants in competitive DRBs compete to solve cryptographic puzzles, whose solutions are unpredictable. The participant who first solves the puzzle becomes the leader, and broadcasts the puzzle solution to other participants. Upon a new puzzle solution, participants execute Nakamoto consensus [1] to agree on and append it to the sequence of puzzle solutions, ensuring consistency and liveness. A random output is extracted from each puzzle solution by using a Verifiable Delay Function (VDF) [154] which takes longer time than the puzzle solution becoming irreversible in the sequence. The time delay prevents the adversary from withholding its puzzle solution and biasing the random output to its own advantage.

RANDCHAIN**: The first scalable and fair DRB.** We propose RANDCHAIN, the first competitive DRB. RANDCHAIN works in permissioned settings identical to all existing DRBs, and is the first to achieve both scalability and fairness: It allows an unbounded number of participants to participate and restricts their voting power to be comparable. To achieve scalability, RANDCHAIN employs Nakamoto consensus [1] with linear communication complexity. To achieve fairness, RANDCHAIN realises *non-parallelisable mining* [155], where more processors do not give any advantage in solving a puzzle. As no existing primitive can provide *non-parallelisable mining*, we introduce *Sequential Proof-of-Work (SeqPoW)*, a cryptographic puzzle that takes a random and unpredictable number of sequential steps to solve. SeqPoW is also of independent interest in other protocols such as leader election and Proof-of-Stake (PoS)-based consensus.

Contributions. Our contributions are summarised as follows.

- We identity and formalise a new design space for DRBs, namely *competitive DRBs*, which break the scalability limit in existing DRB designs.

- As existing primitives lack the properties desired by the competitive DRBs (given the analysis in §5.3), we introduce and formalise the concept of SeqPoW that satisfies these properties. We provide two constructions based on VDFs [156], [157] and Sloth [108], and analyse their security and efficiency. We also discuss applications of SeqPoW in leader election and Proof-of-Stake (PoS)-based consensus (§5.4).

- We provide RANDCHAIN as a concrete instantiation of competitive DRBs, and provide an analysis on its security and performance (§5.5).

- We provide an implementation of SeqPoW and RANDCHAIN and evaluate their performance (§5.6). The implementation adds/changes about 4500 Rust lines of code (LoCs) on top of `parity-bitcoin` [158]. The evaluation results show that RANDCHAIN is indeed scalable and fair: with 1024 nodes, RANDCHAIN can produce a random output every 1.3 seconds (2.3x faster than RandHerd [36], 6.6x faster than HydRand [101] with 128 nodes); utilise constant bandwidth of about 200 KB/s per node (comparable with RandHerd with 1024 nodes and HydRand with 128 nodes); and provide nodes with comparable chance of producing random outputs.

- We establish a unified evaluation framework of DRBs, and compare RANDCHAIN with existing DRBs under this framework (§5.7). Our comparison shows that RANDCHAIN is the only DRB that is secure, scalable and fair, without relying on any trusted third party.

5.2 Model of DRBs

In this section, we define the model for DRBs, including the system model, correctness properties and performance metrics.

5.2.1 System model

System setting. We consider the system setting common in most DRBs [36], [99], [101], [107], [142]–[149]. In particular, a DRB contains a set of n participants $\mathcal{P} = \{p_1, \ldots, p_n\}$. Each participant $p_k \in \mathcal{P}$ has a pair of secret key sk_k and public key pk_k, and is uniquely identified by pk_k. Each participant is only directly connected to a subset of peers in the system. Participants jointly maintain a unique sequence of random outputs. Participants continuously execute the DRB protocol to agree on new random outputs and append them to the sequence.

Network model. Network model concerns the timing guarantee of messages delivery between participants. We consider a synchronous network where messages are delivered within a known finite time bound Δ.

Adversary model. The adversary controls αn processors, and can corrupt at most αn participants in the system, where $\alpha < \frac{1}{2}$. The adversary is adaptive in the sense that it can corrupt any set of $\leq \alpha n$ participants at any time. The adversary can coordinate corrupted participants without delay; and can arbitrarily delay, drop, forge and modify messages from its corrupted participants.

5.2.2 Correctness properties

Consistency and liveness. Similar to consensus, DRBs should satisfy *consistency* and *liveness*. Consistency ensures that participants agree on a unique sequence of random outputs, and liveness ensures that participants produce new random outputs at an admissible rate. We adapt the *common-prefix* and *chain-growth* definitions from Nakamoto consensus protocols [52], [53], [159], [160] rather than the *agreement* and *termination* definitions from BFT-style consensus protocols [34], as we consider a streamlined execution rather than a single-shot execution.

For consistency, we adapt the common-prefix definition in Nakamoto-style consensus where correct participants can only have different views on a certain number of last blocks. In DRBs, the consistency ensures that correct participants can only have different views on a certain number of last random outputs. Some randomness-based applications require RB to have *finality* [58], i.e., at any time, correct participants do not have conflicted views on the random output, which is equivalent to 0-consistency or *agreement* in Byzantine consensus [21].

Definition 5.2.1 (Υ-Consistency)**.** For any two correct participants at any time, their sequences can differ only in the last $\Upsilon \in \mathbb{N}$ random outputs.

For liveness, we adapt the chain-growth definition in Nakamoto-style consensus where correct participants produce blocks at a certain rate. In DRBs, the liveness ensures that correct participants produce random outputs at a certain rate. If the speed does not reach the lowest speed, then the DRB cannot satisfy the requirement of real-world applications. Papers formalising a single-shot execution of DRBs refer liveness as *termination* [99], [145], [149] or *Guaranteed Output Delivery (G.O.D.)* [107], [143], [148], [161] where, for every round, a new random output will be produced.

Definition 5.2.2 ((t, τ)-Liveness)**.** For any time period of length t, every correct participant learns at least $t \cdot \tau$ new random outputs, where $t, \tau \in \mathbb{R}^+$.

Uniform distribution. Uniform distribution ensures that every random output in the DRB is statistically close to a uniformly random string.

Definition 5.2.3 (Uniform distribution)**.** Every random output is indistinguishable from a random string of the same length, except for negligible probability.

Unpredictability. Unpredictability ensures that the adversary cannot predict random outputs that have not been produced yet. Otherwise, if the adversary can predict future random outputs, then it can take advantage in randomness-based applications.

Definition 5.2.4 (Unpredictability)**.** Any adversary can only obtain negligible advantage on the following game. Assuming participants in the DRB agree on an ℓ-long sequence of random outputs. Before the $(\ell+1)$-th random output $R_{\ell+1}$ is produced, the adversary makes a guess $R'_{\ell+1}$ on $R_{\ell+1}$. The adversary's advantage is quantified as $\Pr\left[R'_{\ell+1} = R_{\ell+1}\right]$.

Unbiasibility. Unbiasibility ensures that the adversary cannot influence the produced random output to another value to its own advantage [36], [101], [143], [161]. Otherwise, if the adversary can bias random outputs, then it can take advantage in randomness-based applications. Unbiasibility can be achieved by the *output-independent-abort* property [162]: The adversary has to decide to proceed or abort the protocol before learning the protocol's outcome. In the context of an Υ-consistent DRB, output-independent-abort ensures that, participants learn a random output only after it becomes Υ-deep in a correct participant's view.

Definition 5.2.5 (Unbiasibility)**.** Assuming a DRB satisfies Υ-consistency, and participants in the DRB agree on an ℓ-long sequence of random outputs. The adversary learns the $(\ell+1)$-th random output $R_{\ell+1}$ only after $(\ell+\Upsilon+1)$ consecutive random outputs are recorded in the sequence of at least one correct participant, except for negligible probability.

5.2.3 Performance metrics

Communication complexity. Communication complexity is the total amount of communication required to complete a protocol [96]. In the context of DRBs, the communication complexity is quantified as the amount of communication (in bits) all participants take to generate a random output. For example, for a DRB that includes n participants and achieves $O(n)$ (aka linear) communication complexity, each participant handles a constant amount of communication for generating a random output, leading to the total amount of communication proportional to n. A protocol may have different communication complexity in the best-case and worst-case executions.

Latency. Latency is the time required to complete a protocol. In the context of DRBs, the latency is quantified as the time participants take to generate a random output. Similarly, a protocol may have different latencies in the best-case and worst-case executions. If the protocol's latency only depends on the actual network delay δ but not the delay upper bound Δ, then the protocol is *responsive* [163].

5.3 Design goals and strawman designs

In this section, we describe our two design goals, namely *scalability* and *fairness*, and analyse two strawman designs towards them. The analysis reveals the need for a cryptographic puzzle with two properties, namely *sequentiality* and *hardness*. As no existing puzzle achieves these two properties simultaneously, we are motivated to propose a new primitive named *Sequential Proof-of-Work (SeqPoW, §5.4)* that satisfies both properties, allowing us to construct RANDCHAIN (§5.5).

5.3.1 Design goals: scalability and fairness

Our goal is to design a DRB that can serve security-critical protocols and applications with high financial stake, such as public blockchains and voting protocols. To ensure that the DRB can be trusted by such protocols and applications, we demand two additional requirements on the DRB atop the model in §5.2, namely *scalability* and *fairness*.

Scalability. Scalability specifies that the DRB can produce random outputs regularly even in the presence of a large set of n participants. Having a large set of participants reduces the trust needed on each participant, making the DRB more resilient to malicious parties. Otherwise, if the DRB is maintained by a small set of participants, then they can collude to bias and/or predict random outputs and thus take advantage in the randomness-based applications.

To produce random outputs regularly when n is large, the DRB has to minimise the communication complexity and latency. For communication complexity, $O(n)$ is considered scalable as each participant handles a constant amount of communication independent with n, while $O(n^2)$ is not as each node handles overwhelming communication overhead when n is large. For latency, demand it to be as small as possible.

Fairness. Fairness specifies that each participant controls comparable voting power on deciding random outputs, regardless of their financial stake or hardware resource. The voting power of a node is quantified as the amount of its contributed entropy in collaborative DRBs, and as its chance of producing the next block in competitive DRBs. Without fairness, few powerful participants among all participants will control the randomness generation process of the DRB. This is not desirable as the powerful participants can collude to compromise the DRB, similar to the scalability case.

Unlike DRG-based DRBs that satisfy fairness immediately given the "one-man-one-vote" nature, participants in competitive DRBs may have different voting power, leading to weak fairness. We define fairness as the maximum voting power difference among participants in the DRB. In the context of competitive DRBs, fairness is the maximum difference of nodes' chances of producing the next block.

Definition 5.3.1 (μ-Fairness)**.** Assuming all messages are delivered instantly and participants in a DRB agree on an ℓ-long sequence of random outputs. Let $X(p_k)$ be the event that participant p_k produces the $(\ell+1)$-th random output earlier than other participants. For any two participants p_i and p_j,

$$\mu = \min_{\forall i,j \in [n]} \frac{\Pr[X(p_i)]}{\Pr[X(p_j)]}$$

When $\mu = 1$, the DRB achieves ideal fairness and the network is fully decentralised, and vice versa when $\mu \to 0$. As a design goal, we demand μ to be as close to 1 as possible.

5.3.2 Strawman designs

We analyse two strawman designs towards the two goals. The analysis reveals the need for a cryptographic puzzle satisfying two properties, namely *sequentiality* and *hardness*. No existing puzzle satisfies both of them simultaneously.

Strawman#1: Nakamoto-style DRBs. The scalability goal requires the DRB to achieve $O(n)$ communication complexity. We have shown in §5.1 that no existing DRB achieves it without a trusted third party, motivating us to propose the competitive DRB approach. A natural choice is building upon the Nakamoto-style consensus, where each participant solves a PoW puzzle to become the leader, and a random output is extracted from the PoW solution deterministically.

Such design satisfies *scalability* but not *fairness*, as participants with more mining hardware have more chance of mining blocks than others. To achieve fairness, the DRB has to prevent participants from investing more mining resource to take advantage in mining. A possible solution is the *non-parallelisable mining* [155], where a participant can only use a single processor for mining and cannot speed up mining by using multiple parallel processors. To realise non-parallelisable mining, the puzzle has to be *sequential*: it cannot be solved faster by using multiple parallel processors.

Strawman#2: Applying time-sensitive cryptography. Sequentiality has been formalised and achieved in time-sensitive cryptographic primitives. For example, Verifiable Delay Functions (VDFs) [154] enforce a parameterisable time delay on generating outputs and allow to verify outputs fast. Recent proposals [164], [165] apply VDFs to construct Nakamoto-style consensus: Each participant derives a random output y from the latest system state, maps y to a random time parameter t in a designated interval, and solves a VDF with time parameter t. The first participant solving the VDF derives the next random output from its VDF output.

However, Nakamoto-style consensus with existing time-sensitive primitives achieves weaker fairness and consistency guarantee. All existing time-sensitive primitives have a fixed time delay. Nakamoto-style consensus with such puzzles is *locally predictable* [166]: Given the input x, each participant can learn the time parameter t immediately, and thus can predict when it will propose the next random output. The adversary can apply such prediction to amplify its advantage in selfish mining [47] and double-spending [1], weakening the system's fairness and consistency guarantee, respectively [166].

To make the mining process unpredictable, the puzzle has to take a random and unpredictable number of attempts to solve. PoW satisfies such requirement by providing the *hardness* property [167]: Upon each attempt on solving the puzzle, the solver has probability $\frac{1}{T}$ to solve the puzzle, where T is a hardness parameter. However, none of existing primitives satisfies both *sequentiality* and *hardness*.

5.4 Sequential Proof-of-Work

In this section, we introduce *Sequential Proof-of-Work (SeqPoW)*, a PoW variant that satisfies both *sequentiality* and *hardness*. We formalise SeqPoW, provide two constructions, and analyse their security and efficiency.

5.4.1 Preliminaries on VDFs

Verifiable Delay Function (VDF) [154], [156], [157] allows a prover to evaluate an input, and produce a unique output deterministically with a succinct proof attesting the output's correctness. The evaluation process takes non-negligible and parameterisable time to execute, even with parallelism.

Definition 5.4.1 (Verifiable Delay Function)**.** A Verifiable Delay Function VDF is a tuple of four algorithms

$$\mathsf{VDF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Verify})$$

$\mathsf{Setup}(\lambda) \to pp$**:** On input security parameter λ, outputs public parameter pp. Public parameter pp specifies an input domain $\mathcal{X}$ and an output domain $\mathcal{Y}$. We assume $\mathcal{X}$ is efficiently sampleable.

$\mathsf{Eval}(pp, x, t) \to y$**:** On input public parameter pp, input $x \in \mathcal{X}$, and time parameter $t \in \mathbb{N}^+$, produces output $y \in \mathcal{Y}$.

$\mathsf{Prove}(pp, x, y, t) \to \pi$**:** On input public parameter pp, input x, output y, and time parameter t, outputs proof π.

$\mathsf{Verify}(pp, x, y, \pi, t) \rightarrow \{0, 1\}$: On input pp, x, y, π and t, outputs 1 if y is a correct evaluation, otherwise 0.

VDF satisfies the following properties

- *Completeness*: For all λ, x and t,

$$\Pr\left[\begin{array}{c|c} \mathsf{Verify}(pp, x, y, & pp \leftarrow \mathsf{Setup}(\lambda) \\ & y \leftarrow \mathsf{Eval}(pp, x, t) \\ \pi, t) = 1 & \pi \leftarrow \mathsf{Prove}(pp, x, y, t) \end{array}\right] = 1 \tag{5.1}$$

- *Soundness*: For all λ and adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c|c} \mathsf{Verify}(pp, x, y, \pi, t) = 1 & pp \leftarrow \mathsf{Setup}(\lambda) \\ \wedge \mathsf{Eval}(pp, x, t) \neq y & (x, y, \pi, t) \leftarrow \mathcal{A}(pp) \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{5.2}$$

- *σ-Sequentiality*: For any λ, x, t, $\mathcal{A}_0$ which runs in time $O(\mathsf{poly}(\lambda, t))$ and $\mathcal{A}_1$ which controls any polynomial amount of processors and runs in less than time $\sigma(t)$,

$$\Pr\left[\begin{array}{c|c} & pp \leftarrow \mathsf{Setup}(\lambda) \\ \mathsf{Eval}(x, y, t) = y & \mathcal{A}_1 \leftarrow \mathcal{A}_0(\lambda, t, pp) \\ & y \leftarrow \mathcal{A}_1(x) \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{5.3}$$

VDFs are usually constructed from an iteratively sequential function (ISF) and a succinct proof attesting the ISF's execution results [156], [157]. ISF $f(t, x) = g^t(x)$ is a function that composes a sequential function $g(x)$ for t times. The fastest way of computing ISF $f(t, x)$ is to iterate $g(x)$ for t times, as $g(\cdot)$ is sequential. Squaring and squaring root over cyclic groups of unknown order (e.g., RSA group or class group [168]) are two sequential functions with proven sequentiality [108], [169], [170]. Their repeated versions – repeated squaring [156], [157] and repeated squaring root [108] over cyclic groups – are two widely used ISFs.

ISF $f(\cdot)$ usually provides the *self-composability* property: For any x and (t_1, t_2), let $y \leftarrow f(x, t_1)$, we have $f(x, t_1 + t_2) = f(y, t_2)$. VDFs usually inherit the *self-composability* from ISFs. Such VDFs are known as *self-composable VDFs* [171].

Definition 5.4.2 (Self-Composability)**.** A VDF $(\mathsf{Setup}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Verify})$ satisfies self-composability if for all λ, x, (t_1, t_2),

$$\Pr\left[\begin{array}{c|c} \mathsf{Eval}(pp, x, t_1 + t_2) & pp \leftarrow \mathsf{Setup}(\lambda) \\ = \mathsf{Eval}(pp, y, t_2) & y \leftarrow \mathsf{Eval}(pp, x, t_1) \end{array}\right] = 1 \tag{5.4}$$

Lemma 5.4.1. *If a VDF* $(\mathsf{Setup}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Verify})$ *satisfies self-composability, then for all* λ, x, (t_1, t_2),

$$\Pr\left[\begin{array}{c|c} & pp \leftarrow \mathsf{Setup}(\lambda) \\ \mathsf{Verify}(pp, x, y', & y \leftarrow \mathsf{Eval}(pp, x, t_1) \\ \pi, t_1 + t_2) = 1 & y' \leftarrow \mathsf{Eval}(pp, y, t_2) \\ & \pi \leftarrow \mathsf{Prove}(pp, x, y', t_1 + t_2) \end{array}\right] = 1 \tag{5.5}$$

5.4.2 Basic idea of SeqPoW

SeqPoW is a cryptographic puzzle that takes a random and unpredictable number of sequential steps to solve. As shown in Figure 5.1, given an initial SeqPoW puzzle S_0, the prover keeps solving it by incrementing an ISF. Each iteration takes the last output S_{i-1} as input and produces a new output S_i. For each output S_i, the prover checks whether it satisfies a difficulty parameter T. If yes, then S_i is a valid solution, and the prover can generate a proof π_i on it. Given S_i and π_i, the verifier can check S_i's correctness without solving the puzzle again.

Comparisons with relevant primitives (Table 5.1). SeqPoW is the first primitive that satisfies both sequentiality and hardness, and therefore can be used for constructing RandChain. SeqPoW differs from VDFs and other

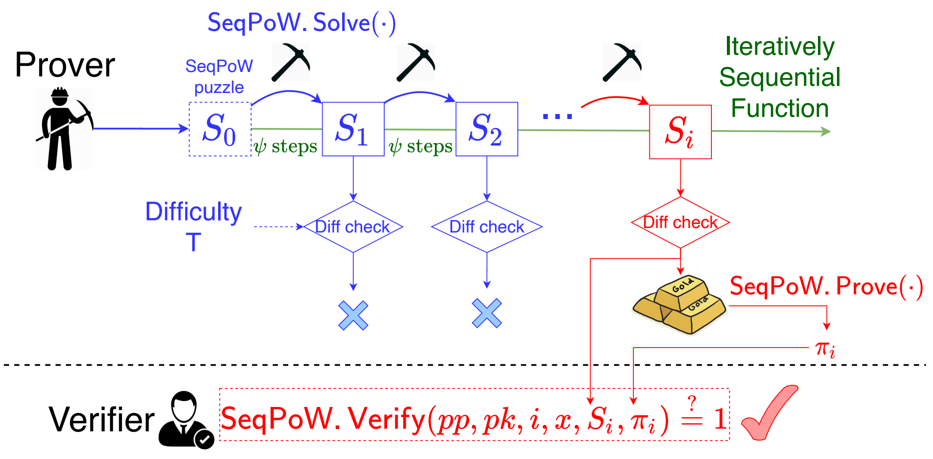


Figure 5.1: Sequential Proof-of-Work.

time-sensitive cryptographic primitives, e.g., Timelock Puzzle (TLP) [172] and Proofs of Sequential Work (PoSW) [173], [174] in that, the SeqPoW prover iterates an ISF for a *randomised* (rather than given) number of times. In addition, compared to TLP, SeqPoW provides publicly verifiable outputs. Compared to PoSW, SeqPoW allows outputs to be unique. SeqPoW differs from PoW in that SeqPoW is sequential. SeqPoW differs from *memory-hard functions* (MHFs) [175]–[177] in that, SeqPoW is bottlenecked by the processor's frequency, whereas MHF is bottlenecked by the memory bandwidth.

Two concurrent works [164], [178] propose ways to randomise the number of iterations in VDFs, without formal treatment. We are the first to formally study such primitives, including formal definitions, concrete constructions with security proofs, implementation and evaluation. We also provide SeqPoW with uniqueness that both of them cannot achieve.

Applications. Given the unpredictability and hardness properties, SeqPoW is of independent interest for other protocols. First, SeqPoW can improve the fairness of leader election protocols. Mining in PoW-based consensus can be seen as a way of electing leaders: given a set of participants, the first participant proposing a valid PoW solution becomes the leader and proposes a block. SeqPoW can be a drop-in replacement of PoW for the

Table 5.1: SeqPoW v.s. relevant primitives.

Primitive		Execution			Output	
		Sequential	# Steps	Bottleneck	Unique	Verifiable
Time-sensitive	TLP	✓	Fixed	Proc. freq.	✓	✗
	PoSW	✓	Fixed	Proc. freq.	✗	✓
	VDF	✓	Fixed	Proc. freq.	✓	✓
Resource-consuming	MHF	✓or ✗	Fixed	Mem. bandw.	✓	✓
	PoW	✗	Random	Proc. freq. + # of procs.	✗	✓
Our work	$\text{SeqPoW}_{\text{VDF}}$	✓	Random	Proc. freq.	✗	✓
	$\text{SeqPoW}_{\text{Sloth}}$	✓	Random	Proc. freq.	✓	✓

leader election purpose. In §5.5.3, we show that compared to parallelisable PoW, SeqPoW-based leader election achieves better fairness.

Second, SeqPoW can improve the fault tolerance capacity of Proof-of-Stake (PoS)-based consensus. In Proof-of-Stake (PoS)-based consensus [179], each participant's chance of mining a block is in proportion to its *stake*, e.g, the participant's balance. Most PoS-based consensus protocols [144], [147], [180]–[182] select block proposers in a *predictable* [120], [166] way, thus are vulnerable to various prediction-based attacks and tolerate less Byzantine mining power [120], [166] than PoW-based consensus, as analysed in §5.3. To make PoS-based consensus unpredictable, one can randomise the process of selecting block proposers. SeqPoW can provide such functionality: Each participant solves a SeqPoW with its identity, the last block, and the difficulty parameter inversely proportional to its stake as input, and the first participant solving its SeqPoW becomes the block proposer. A concurrent and independent work [178] provides a concrete protocol following the similar idea.

5.4.3 Definition

We present the formal syntax of Sequential Proof-of-Work (SeqPoW).

Definition 5.4.3 (Sequential Proof-of-Work (SeqPoW))**.** A Sequential Proof-of-Work SeqPoW is a tuple of algorithms

$$\mathsf{SeqPoW} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Init}, \mathsf{Solve}, \mathsf{Verify})$$

$\mathsf{Setup}(\lambda, \psi, T) \rightarrow pp$**:** On input security parameter λ, step $\psi \in \mathbb{N}^+$ and difficulty $T \in [1, \infty)$, outputs public parameter pp. Public parameter pp specifies an input domain $\mathcal{X}$, an output domain $\mathcal{Y}$, and a cryptographically secure hash function $H : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ is efficiently sampleable.

$\mathsf{Gen}(pp) \rightarrow (sk, pk)$**:** A probabilistic function, which on input public parameter pp, produces a secret key $sk \in \mathcal{X}$ and a public key $pk \in \mathcal{X}$.

$\mathsf{Init}(pp, sk, x) \rightarrow (S_0, \pi_0)$**:** On input public parameter pp, secret key sk, and input $x \in \mathcal{X}$, outputs initial solution $S_0 \in \mathcal{Y}$ and proof π_0. Some constructions may use public key pk as input rather than sk. This also applies to $\mathsf{Solve}(\cdot)$ and $\mathsf{Prove}(\cdot)$.

$\mathsf{Solve}(pp, sk, S_i) \rightarrow (S_{i+1}, b_{i+1})$**:** On input public parameter pp, secret key sk, and i-th solution $S_i \in \mathcal{Y}$, outputs $(i+1)$-th solution $S_{i+1} \in \mathcal{Y}$ and result $b_{i+1} \in \{0, 1\}$.

$\mathsf{Prove}(pp, sk, i, x, S_i) \rightarrow \pi_i$**:** On input public parameter pp, secret key sk, i, input x, and i-th solution S_i, outputs proof π_i.

$\mathsf{Verify}(pp, pk, i, x, S_i, \pi_i) \rightarrow \{0, 1\}$**:** On input pp, pk, i, x, S_i, and π_i, outputs 1 if S_i is a valid solution, otherwise 0.

We define honest tuples and valid tuples as follows.

Definition 5.4.4 (Honest tuple)**.** A tuple $(pp, sk, i, x, S_i, \pi_i)$ is (λ, ψ, T)-honest if and only if for all $pp \leftarrow \mathsf{Setup}(\lambda, \psi, T)$, the following holds:

- $i = 0$ and $(S_0, \pi_0) \leftarrow \mathsf{Init}(pp, sk, x)$, and

- $\forall i \in \mathbb{N}^+$, $(S_i, b_i) \leftarrow \mathsf{Solve}(pp, sk, S_{i-1})$ and $\pi_i \leftarrow \mathsf{Prove}(pp, sk, i, x, S_i)$, where $(pp, sk, i-1, x, S_{i-1}, \pi_{i-1})$ is (λ, ψ, T)-honest.

Definition 5.4.5 (Valid tuple). For all λ, ψ, T, and $pp \leftarrow \mathsf{Setup}(\lambda, \psi, T)$, a tuple $(pp, sk, i, x, S_i, \pi_i)$ is (λ, ψ, T)-valid if

- $(pp, sk, i, x, S_i, \pi_i)$ is (λ, ψ, T)-honest, and
- $\mathsf{Solve}(pp, sk, S_{i-1}) = (\cdot, 1)$

SeqPoW should satisfy *completeness*, *soundness*, *hardness* and *sequentiality*, plus an optional property *uniqueness*.

Definition 5.4.6 (Completeness). A SeqPoW scheme satisfies completeness if for all λ, ψ, T,

$$\Pr\left[\begin{array}{c|c} & pp \leftarrow \mathsf{Setup}(\lambda, \psi, T) \\ \mathsf{Verify}(pp, pk, i, & (sk, pk) \leftarrow \mathsf{Gen}(pp) \\ x, S_i, \pi_i) = 1 & (pp, pk, i, x, S_i, \pi_i) \\ & \text{is } (\lambda, \psi, T)\text{-valid} \end{array}\right] = 1 \tag{5.6}$$

Definition 5.4.7 (Soundness). A SeqPoW scheme satisfies soundness if for all λ, ψ, T,

$$\Pr\left[\begin{array}{c|c} & pp \leftarrow \mathsf{Setup}(\lambda, \psi, T) \\ \mathsf{Verify}(pp, pk, i, & (sk, pk) \leftarrow \mathsf{Gen}(pp) \\ x, S_i, \pi_i) = 1 & (pp, pk, i, x, S_i, \pi_i) \\ & \text{is not } (\lambda, \psi, T)\text{-valid} \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{5.7}$$

Definition 5.4.8 (Hardness). A SeqPoW scheme satisfies hardness if for all (λ, ψ, T)-honest tuple $(pp, sk, i, x, S_i, \pi_i)$,

$$\left|\Pr\left[\begin{array}{c|c} b_{i+1} = 1 & \begin{array}{c}(S_{i+1}, b_{i+1}) \leftarrow \\ \mathsf{Solve}(pp, sk, S_i, \pi_i)\end{array} \end{array}\right] - \frac{1}{T}\right| \leq \mathsf{negl}(\lambda) \tag{5.8}$$

Definition 5.4.9 (σ-Sequentiality)**.** A SeqPoW scheme satisfies σ-sequentiality if for all λ, ψ, T, i, x, $\mathcal{A}_0$ which runs in less than time $O(\mathsf{poly}(\lambda, \psi, i))$ and $\mathcal{A}_1$ which runs in less than time $\sigma(i \cdot \psi)$ with at most $\mathsf{poly}(\lambda)$ processors,

$$\Pr\left[\begin{array}{c|c} (pp, sk, i, x, S_i, \pi_i) & pp \leftarrow \mathsf{Setup}(\lambda, \psi, T) \\ \text{is } (\lambda, \psi, T)\text{-honest} & (sk, pk) \leftarrow \mathsf{Gen}(pp) \\ & \mathcal{A}_1 \leftarrow \mathcal{A}_0(pp, sk) \\ & S_i \leftarrow \mathcal{A}_1(i, x) \\ & \pi_i \leftarrow \mathsf{Prove}(pp, sk, i, x, S_i) \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{5.9}$$

SeqPoW also has an optional property *uniqueness*, by which each SeqPoW puzzle only has a single valid solution S_i. Before finding a valid solution S_i each $\mathsf{Solve}(\cdot)$ attempt follows the *hardness* definition, but after finding S_i no further $\mathsf{Solve}(\cdot)$ attempt returns a valid solution.

Definition 5.4.10 (Uniqueness (optional))**.** A SeqPoW scheme satisfies uniqueness if for any two (λ, ψ, T)-valid tuples $(pp, sk, i, x, S_i, \pi_i)$ and $(pp, sk, i, x, S_j, \pi_j)$, $i = j$ holds.

5.4.4 Constructions

Let $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ be a cryptographic hash function; G be a cyclic group with unknown order (e.g., RSA group or class group); $H_G : \{0,1\}^* \rightarrow G$ be a function mapping an arbitrary string to an element on G; g be a generator of G; sk be the secret key; and $pk = g^{sk}$ be the public key. Let expression$?x : y$ be the tenary operator that returns x if the expression is true or y otherwise.

SeqPoW from VDFs (Figure 5.2a). Let ψ be a step parameter, x be the input, and T be the difficulty parameter. The prover runs $\mathsf{Init}(\cdot)$, which generates the initial solution $S_0 = H_G(pk \| x)$. Then, the prover keeps running $\mathsf{Solve}(\cdot)$, which calculates an intermediate output $S_i = \mathsf{VDF.Eval}(pp, S_{i-1}, \psi)$

```
Setup($\lambda, \psi, T$)
  $pp_{\mathsf{VDF}} = (G, g) \leftarrow \mathsf{VDF.Setup}(\lambda)$
  $pp \leftarrow (pp_{\mathsf{VDF}}, \psi, T)$
  return $pp$

Gen($pp$)
  $(G, g, \psi, T) \leftarrow pp$
  Sample random $sk \in \mathbb{N}$
  $pk \leftarrow g^{sk} \in G$
  return $(sk, pk)$

Init($pp, pk, x$)
  $(G, g, \psi, T) \leftarrow pp$
  $S_0 \leftarrow H_G(pk \| x)$
  return $S_0$

Solve($pp, pk, S_i$)
  $(pp_{\mathsf{VDF}}, \psi, T) \leftarrow pp$
  $S_{i+1} \leftarrow \mathsf{VDF.Eval}(pp_{\mathsf{VDF}}, S_i, \psi)$
  $b_{i+1} \leftarrow H(pk \| S_{i+1}) \leq \frac{2^\kappa}{T} \, ? \, 1 : 0$
  return $(S_{i+1}, b_{i+1})$

Prove($pp, pk, i, x, S_i$)
  $(pp_{\mathsf{VDF}}, \psi, T) \leftarrow pp$
  $(G, g) \leftarrow pp_{\mathsf{VDF}}$
  $S_0 \leftarrow H_G(pk \| x)$
  $\pi_{\mathsf{VDF}} \leftarrow \mathsf{VDF.Prove}(pp_{\mathsf{VDF}}, S_0, S_i, i \cdot \psi)$
  return $\pi_{\mathsf{VDF}}$

Verify($pp, pk, i, x, S_i, \pi_i$)
  $(pp_{\mathsf{VDF}}, \psi, T) \leftarrow pp$
  $(G, g) \leftarrow pp_{\mathsf{VDF}}$
  $S_0 \leftarrow H_G(pk \| x)$
  if $\mathsf{VDF.Verify}(pp_{\mathsf{VDF}}, S_0, S_i, \pi_i, i \cdot \psi) = 0$ then
    return 0
  if $H(pk \| S_i) > \frac{2^\kappa}{T}$ then
    return 0
  return 1
```

(a) $\mathsf{SeqPoW}_{\mathsf{VDF}}$.

```
Setup($\lambda, \psi, T$)
  $pp \leftarrow (G, g, \psi, T)$
  return $pp$

Gen($pp$)
  $(G, g, \psi, T) \leftarrow pp$
  Sample random $sk \in \mathbb{N}$
  $pk \leftarrow g^{sk} \in G$
  return $(sk, pk)$

Init($pp, pk, x$)
  $(G, g, \psi, T) \leftarrow pp$
  $S_0 \leftarrow H_G(pk \| x)$
  return $S_0$

Solve($pp, pk, S_i$)
  $(G, g, \psi, T) \leftarrow pp$
  $S_{i+1} \leftarrow S_i^{\frac{1}{2^\psi}}$
  $b_{i+1} \leftarrow H(pk \| S_{i+1}) \leq \frac{2^\kappa}{T} \, ? \, 1 : 0$
  return $(S_{i+1}, b_{i+1})$

Prove($pp, pk, i, x, S_i$)
  return $\perp$

Verify($pp, pk, i, x, S_i, \pi_i$)
  $(G, g, \psi, T) \leftarrow pp$
  $y \leftarrow S_i$
  if $H(pk \| y) > \frac{2^\kappa}{T}$ then return 0
  repeat $i$ times
    $y \leftarrow y^{2^\psi}$
    if $H(pk \| y) \leq \frac{2^\kappa}{T}$ then return 0
  if $H_G(pk \| x) \neq y$ then return 0
  return 1
```

(b) $\mathsf{SeqPoW}_{\mathsf{Sloth}}$.

Figure 5.2: Construction of SeqPoW.

and checks whether $H(pk \| S_i) \leq \frac{2^\kappa}{T}$. If true, then S_i is a valid solution, and the prover runs $\mathsf{Prove}(\cdot)$, which outputs proof π_i attesting $S_i = \mathsf{VDF.Eval}^i(pp, S_0, \psi)$. Note that when VDF is *self-composable*, we have

$$S_i = \mathsf{VDF.Eval}(pp, S_{i-1}, \psi) = \mathsf{VDF.Eval}^i(pp, S_0, \psi) = \mathsf{VDF.Eval}(pp, S_0, i \cdot \psi)$$

The verifier runs $\mathsf{Verify}(\cdot)$, which checks 1) whether $S_i = \mathsf{Eval}^i(pp, S_0, \psi)$ by running $\mathsf{VDF.Verify}(pp_{\mathsf{VDF}}, pk, i \cdot \psi, x, S_i, \pi_i)$, and 2) whether S_i satisfies the difficulty T.

Unique SeqPoW from Sloth (Figure 5.2b). $\mathsf{SeqPoW}_{\mathsf{VDF}}$ does not provide *uniqueness*: The prover can keep incrementing the ISF to find as many valid solutions as possible. We construct $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ with *uniqueness* from

Sloth [108], a *slow-timed* hash function. In Sloth, the prover calculates the square root (on a cyclic group G) over the input for t times to get the output. The verifier calculates the square over the output for t times to recover the input and checks if the input is same as the one from the prover. Although the verification is linear (and thus do not meet the VDF definition [154]), verification is faster than computing: On cyclic group G, squaring is $O(\log |G|)$ times faster than square rooting. Similar to $\mathsf{SeqPoW}_{\mathsf{VDF}}$, $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ takes each of $S_i = f(i \cdot \psi, S_0)$ as an intermediate output and checks if $H(pk \| S_i) \leq \frac{2^\kappa}{T}$. To make the solution unique, $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ only treats the first solution satisfying the difficulty as valid. When verifying S_i, if the verifier finds an intermediate output S_j ($j < i$) satisfying the difficulty, then S_i is considered invalid.

5.4.5 Security and efficiency analysis

Security. We first summarise the security analysis. The completeness and soundness are immediate from Sloth and VDFs' completeness, soundness and self-composability. By *pseudorandomness* of $H_G(\cdot)$ and *sequentiality* of Sloth and VDFs, $\mathsf{Solve}(\cdot)$ outputs unpredictable solutions. As $H(\cdot)$ is modelled as a random oracle and $\mathsf{Solve}(\cdot)$ produces an unpredictable solution, the probability that the solution satisfies the difficulty is $\frac{1}{T}$, leading to *hardness*. The sequentiality and self-composability of Sloth and VDFs guarantee the sequentiality of the SeqPoW constructions.

VDFs can be instantiated with any cyclic group, including the RSA group that requires a trusted setup and the class group without such requirement. The trusted setup is usually conducted by a trusted party or a multi-party protocol [183], [184].

Then, we present the formal security proofs for the SeqPoW constructions.

Lemma 5.4.2. $\mathsf{SeqPoW}_{\mathsf{VDF}}$ *satisfies completeness.*

Proof. Assuming a (λ, ψ, T)-valid tuple $(pp, sk, i, x, S_i, \pi_i)$, by *completeness* and Lemma 5.4.1, $\mathsf{VDF.Verify}(\cdot)$ will pass. As hash functions are deterministic, difficulty check will pass. Therefore,

$$\mathsf{SeqPoW_{VDF}.Verify}(pp, pk, i, x, S_i, \pi_i) = 1$$

□

Lemma 5.4.3. $\mathsf{SeqPoW_{VDF}}$ *satisfies soundness.*

Proof. We prove this by contradiction. Assuming a tuple $(pp, sk, i, x, S_i, \pi_i)$ that is not (λ, ψ, T)-valid and

$$\mathsf{SeqPoW_{VDF}.Verify}(pp, pk, i, x, S_i, \pi_i) = 1$$

By *soundness* and Lemma 5.4.1, if (y, y^+, π^+, ψ) is generated by $\mathcal{A}$, $\mathsf{VDF.Verify}(\cdot)$ will return 0. As hash functions are deterministic, if $S_i > \frac{2^\kappa}{T}$, difficulty check will return 0. Thus, if $(pp, sk, i, x, S_i, \pi_i)$ is not (λ, ψ, T)-valid, then the adversary can break *soundness*. Thus, this assumption contradicts *soundness*. □

Lemma 5.4.4. $\mathsf{SeqPoW_{VDF}}$ *satisfies hardness.*

Proof. We prove this by contradiction. Assuming

$$\left| \Pr\left[b_{i+1} = 1 \;\middle|\; \begin{array}{c} S_{i+1}, b_{i+1} \leftarrow \\ \mathsf{Solve}(pp, sk, T, S_i) \end{array} \right] - \frac{1}{T} \right| > \mathsf{negl}(\lambda) \tag{5.10}$$

By *sequentiality*, the value of S_{i+1} is unpredictable before finishing $\mathsf{Solve}(\cdot)$. By pseudorandomness of hash functions, $H(pk \| S_{i+1})$ is uniformly distributed, and the probability that $H(pk \| S_{i+1}) \leq \frac{2^\kappa}{T}$ is $\frac{1}{T}$ with negligible probability. This contradicts the assumption. □

Lemma 5.4.5. $\mathsf{SeqPoW_{VDF}}$ *does not satisfy uniqueness.*

Proof. By *hardness*, each of S_i has the probability $\frac{1}{T}$ to be a valid solution. As i can be infinite, with $(1-\epsilon)$ probability where ϵ is negligible, there exists more than one honest tuple $(pp, sk, i, x, S_i, \pi_i)$ such that $H(pk \| S_i) \leq \frac{2^\kappa}{T}$. □

Lemma 5.4.6. *If the underlying VDF satisfies σ-sequentiality, then* $\mathsf{SeqPoW}_{\mathsf{VDF}}$ *satisfies σ-sequentiality.*

Proof. We prove this by contradiction. Assuming there exists $\mathcal{A}_1$ which runs in less than time $\sigma(i \cdot \psi)$ such that

$$\Pr\left[\begin{array}{c|c} (pp, sk, i, x, S_i, \pi_i) & pp \leftarrow \mathsf{Setup}(\lambda, \psi, T) \\ \in \mathcal{H} & (sk, pk) \xleftarrow{R} \mathsf{Gen}(pp) \\ & \mathcal{A}_1 \leftarrow \mathcal{A}_0(\lambda, pp, sk) \\ & S_i \leftarrow \mathcal{A}_1(i, x) \\ & \pi_i \leftarrow \mathsf{Prove}(pp, sk, i, x, S_i) \end{array}\right] \tag{5.11}$$

is non-negligible. By *σ-sequentiality*, $\mathcal{A}_1$ cannot solve $\mathsf{VDF.Eval}(pp_{\mathsf{VDF}}, y, \psi)$ within $\sigma(\psi)$. By Lemma 5.4.1, S_i can and only can be computed by composing $\mathsf{VDF.Eval}(pp_{\mathsf{VDF}}, y, \psi)$ for i times, which cannot be solved within $\sigma(i \cdot \psi)$. Thus, if the assumption holds, then $\mathcal{A}_1$ can break the *σ-sequentiality* property of VDF, leading to a contradiction. □

The completeness, soundness, hardness and sequentiality proofs of $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ are identical to $\mathsf{SeqPoW}_{\mathsf{VDF}}$'s. We prove $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ satisfies uniqueness below.

Lemma 5.4.7. $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ *satisfies uniqueness.*

Proof. We prove this by contradiction. Assuming there exists two (λ, ψ, T)-valid tuples $(pp, sk, i, x, S_i, \pi_i)$ and $(pp, sk, i, x, S_i, \pi_i)$ where $j < i$. According to $\mathsf{SeqPoW}_{\mathsf{Sloth}}.\mathsf{Solve}(\cdot)$, we have $H(pk \| S_i) \leq \frac{2^\kappa}{T}$ and $H(pk \| S_j) \leq \frac{2^\kappa}{T}$, and initial difficulty check in $\mathsf{SeqPoW}_{\mathsf{Sloth}}.\mathsf{Verify}(\cdot)$ will pass. However, in the for loop of $\mathsf{SeqPoW}_{\mathsf{Sloth}}.\mathsf{Verify}(\cdot)$, if S_i is valid, then verification of S_j will fail. Then, $\mathsf{SeqPoW}_{\mathsf{Sloth}}.\mathsf{Verify}(\cdot)$ returns 0, which contradicts the assumption. □

Table 5.2: Efficiency of two SeqPoW constructions.

	$\mathsf{Solve}(\cdot)$	$\mathsf{Prove}(\cdot)$	$\mathsf{Verify}(\cdot)$	Proof size (Bytes)
$\mathsf{SeqPoW}_{\mathsf{VDF}}$ + Wes19	$O(\psi)$	$O(\psi T)$	$O(\log \psi T)$	s
$\mathsf{SeqPoW}_{\mathsf{VDF}}$ + Pie19	$O(\psi)$	$O(\sqrt{\psi T} \log \psi T)$	$O(\log \psi T)$	$s \log_2 \psi T$
$\mathsf{SeqPoW}_{\mathsf{Sloth}}$	$O(\psi)$	0	$O(\psi T)$	0

Efficiency (Table 5.2). $\mathsf{SeqPoW}_{\mathsf{VDF}}$ and $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ employ repeated squaring on an RSA group and repeated square rooting on a prime-order group as ISFs, respectively. Let s be the size (in Bytes) of a group element, and ψ be the step parameter. Each $\mathsf{Solve}(\cdot)$ executes ψ steps of the ISF, and the prover attempts $\mathsf{Solve}(\cdot)$ for T times on average to find a valid solution. $\mathsf{Prove}(\cdot)$ and $\mathsf{Verify}(\cdot)$ generate and verify proofs of ψT consecutive modular squaring operations, respectively.

We analyse $\mathsf{SeqPoW}_{\mathsf{VDF}}$ with both Wesolowski's VDF (Wes19) [157] and Pietrzak's VDF (Pie19) [156] without optimisation/parallelisation techniques [156], [157], [185]. According to the existing analysis [186], the proving complexity, verification complexity and proof size of Wes19 are $O(\psi T)$, $O(\log \psi T)$ and s Bytes, respectively; and those of Pie19 are $O(\sqrt{\psi T} \log \psi T)$, $O(\log \psi T)$ and $s \log_2 \psi T$, respectively. When $\psi T = 2^{40}$ and $s = 32$ Bytes, the proof sizes of $\mathsf{SeqPoW}_{\mathsf{VDF}}$ with Wes19 [157] and with Pie19 [156] are 32 and 1280 Bytes, respectively. $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ has the verification complexity of $O(\psi T)$ and uses the solution itself to represent the proof.

5.5 RANDCHAIN: DRB from SeqPoW

In this section, we build the RANDCHAIN protocol. Figure 5.3 and 5.4 provides the intuition and full specification of RANDCHAIN, respectively. In RANDCHAIN, participants jointly maintain a sequence of random outputs as a blockchain, where each random output is derived from a block (§5.5.1). Specifically, participants agree on a unique blockchain by executing

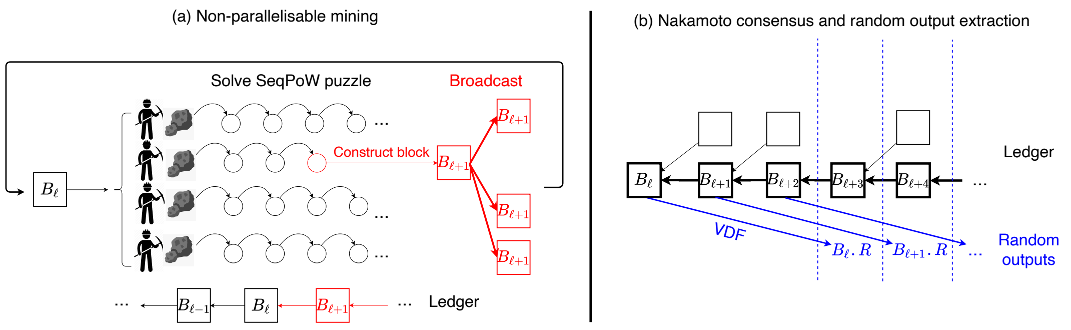


Figure 5.3: The RANDCHAIN protocol. **(a)** Upon block B_ℓ, each participant keeps solving its own SeqPoW puzzle. The participant who first solves its SeqPoW puzzle (the red one) proposes the next block $B_{\ell+1}$ (in red). $B_{\ell+1}$ piggybacks B_ℓ by including B_ℓ's ID, i.e., $B_{\ell+1}.h^- = B_\ell.h$. **(b)** Each participant maintains a local ledger formed as a DAG of blocks. It considers the longest fork of the DAG as the main chain and mines over it. For each block B_ℓ, the random output $B_\ell.R$ is extracted by a VDF that takes longer than nodes extending $(\Upsilon + 1)$ blocks (in this case $\Upsilon = 1$) so that $B_\ell.R$ is learned only after B_ℓ becomes irreversible.

```
mainChain($\mathcal{C}_k$)
1 :  $\mathcal{MC}_k \leftarrow \perp$
2 :  foreach fork $\mathcal{C}_k^*$ of $\mathcal{C}_k$ do
3 :    if $|\mathcal{C}_k^*| > |\mathcal{MC}_k|$ then $\mathcal{MC}_k \leftarrow \mathcal{C}_k^*$
4 :  return $\mathcal{MC}_k$

randomOutput($pp,\ B,\ t_{\mathsf{VDF}}$)
1 :  $(pp_{\mathsf{VDF}},\ \cdot,\ \cdot) \leftarrow pp$
2 :  $B.R \leftarrow \mathsf{VDF.Eval}(pp_{\mathsf{VDF}},\ B.pk \| B.S,\ t_{\mathsf{VDF}})$
3 :  $B.\pi_R \leftarrow \mathsf{VDF.Prove}(pp_{\mathsf{VDF}},\ B.pk \| B.S,$
4 :                      $B.R,\ t_{\mathsf{VDF}})$
5 :  return $B.R,\ B.\pi_R$

MainProcedure($pp,\ sk_k,\ pk_k$)
1 :  Synchronise ledger as $\mathcal{C}_k$
// The following two lines read/modify $C_k$ concurrently
2 :  MineRoutine($pp,\ sk_k,\ pk_k,\ \mathcal{C}_k$) in a thread
3 :  SyncRoutine($pp,\ \mathcal{C}_k$) in a thread
```

```
MineRoutine($pp,\ sk_k,\ pk_k,\ \mathcal{C}_k$)
1 :  repeat
2 :    $\mathcal{MC}_k \leftarrow \mathsf{mainChain}(\mathcal{C}_k)$
3 :    $B^- \leftarrow \mathcal{MC}_k[-1]$
4 :    $i \leftarrow 0$
5 :    $S \leftarrow \mathsf{SeqPoW.Init}(pp,\ sk_k,\ B^-.h)$
6 :    repeat
7 :      if $\mathcal{C}_k$ is updated by SyncRoutine($\cdot$) then
8 :        Repeat line 2-5
9 :      $S,\ b \leftarrow \mathsf{SeqPoW.Solve}(pp,\ sk_k,\ S)$
10 :     $i += 1$
11 :     if $b = 1$ then break
12 :   $h \leftarrow H(pk_k \| S)$
13 :   $\pi \leftarrow \mathsf{SeqPoW.Prove}(pp,\ sk,\ i,\ B^-.h,\ S)$
14 :   $B \leftarrow (B^-.h,\ h,\ i,\ S,\ pk_k,\ \pi)$
15 :   Append $B$ to $\mathcal{MC}_k$ after $B^-$
16 :   Propagate $B$
```

```
SyncRoutine($pp,\ \mathcal{C}_k$)
1 :  repeat
2 :    Wait for a new block as $B$
3 :    $(h^-,\ h,\ i,\ S,\ pk,\ \pi) \leftarrow B$
4 :    // $B$ should point to an existing block
5 :    if $\nexists B^- \in \mathcal{C}_k :\ B^-.h = h^-$ then discard $B$
6 :    // $B$ should have a valid ID $h$
7 :    if $h \neq H(pk \| S)$ then discard $B$
8 :    // $B$ should include a valid puzzle solution
9 :    if $\mathsf{SeqPoW.Verify}(pp,\ pk,\ i,\ h^-,\ S,\ \pi) = 0$ then
10 :     Discard $B$
11 :   Append $B$ to $\mathcal{C}_k$ after block $B^-$ where $B^-.h = h^-$
12 :   Propagate $B$
```

Figure 5.4: Full specification of RANDCHAIN.

the Nakamoto consensus, which ensures consistency, liveness, and scalability in synchronous networks (§5.5.2). RANDCHAIN composes Nakamoto consensus with our proposed SeqPoW puzzle to achieve *non-parallelisable mining*, guaranteeing the fairness (§5.5.3). Each random output is extracted from a block by using a Verifiable Delay Function (VDF) so that the random output is learned only after the block becomes irreversible in the blockchain, guaranteeing the uniform distribution, unpredictability and unbiasibility (§5.5.4).

5.5.1 DRB structure

Each participant p_k locally maintains a ledger $\mathcal{C}_k$ formed as a directed acyclic graph (DAG) of blocks. Following Nakamoto consensus $\mathsf{mainChain}(\cdot)$, p_k selects the longest fork in $\mathcal{C}_k$ as the main chain $\mathcal{MC}_k$. If there are multiple longest forks at the same length, p_k chooses the one it receives first. $\mathcal{MC}_k$ is formed as a blockchain, i.e., a totally ordered sequence of blocks. We denote $|\mathcal{MC}_k|$ as the length of $\mathcal{MC}_k$.

Each block B is of the format $B = (h^-, h, i, S, pk, \pi)$, where h^- is the previous block ID, h is the current block ID, i is the SeqPoW solution index, S is the SeqPoW solution, pk is the public key of this block's creator, and π is the proof that S is a valid SeqPoW solution on input h^-. Each block B is identified by its ID $B.h = H(B.pk \| B.S)$, and points to a previous block B^- by setting $B.h^- = B^-.h$. One can extract a random output $B.R$ from each block B by using a deterministic function $\mathsf{randomOutput}(\cdot)$, which we will describe later in (§5.5.4).

5.5.2 Synchronising and agreeing on blocks

Each participant p_k keeps running $\mathsf{SyncRoutine}(\cdot)$ to synchronise its local ledger $\mathcal{C}_k$ with other participants. Specifically, participant p_k keeps receiving blocks from other participants, verifying them, and adding valid blocks to its local ledger $\mathcal{C}_k$. Participant p_k keeps tracking the main chain $\mathcal{MC}_k$ following Nakamoto consensus $\mathsf{mainChain}(\cdot)$, and executes the mining routine $\mathsf{MineRoutine}(\cdot)$ on $\mathcal{MC}_k$.

Same as PoW-based Nakamoto consensus, RANDCHAIN achieves consistency and liveness in synchronous networks, and can tolerate an adversary with mining power $\alpha < \frac{1}{2}$. As Nakamoto consensus is probabilistic, RANDCHAIN does not achieve 0-consistency (aka *finality*). One can deploy existing finality layer mechanisms [41], [187], [188] into RANDCHAIN. In §5.8.3 we analyse two approaches of adding *finality* to RANDCHAIN.

RANDCHAIN inherits communication complexity and latency guarantees from Nakamoto consensus. The communication complexity is $O(n)$ as the only communication is the leader broadcasting blocks. The latency is $t_{\text{block}} + \delta$, where t_{block} is the parameterised block interval and δ is the actual network delay. Thus, RANDCHAIN achieves scalability.

5.5.3 Non-parallelisable mining

RANDCHAIN employs the SeqPoW puzzle for the mining routine, which we call $\mathsf{MineRoutine}(\cdot)$. Specifically, participant p_k keeps solving the latest SeqPoW puzzle S derived from $\mathsf{SeqPoW.Init}(pp, sk_k, B^{-}.h)$, where pp is the public parameter, sk_k is its secret key, and $B^{-}.h$ is the hash of $\mathcal{MC}_k$'s last block. To solve puzzle S, participant p_k keeps executing $\mathsf{SeqPoW.Solve}(\cdot)$ until finding a valid solution. With a valid solution, participant p_k constructs a block B, and appends B to $\mathcal{MC}_k$.

RANDCHAIN achieves non-parallelisable mining, leading to μ-fairness with $\mu > \frac{1}{5}$ in practice where every node at least preserves a commodity processor with 2∼3 GHz frequency. Each participant has a unique input deriving a unique SeqPoW puzzle, so can only use a single processor for mining. By SeqPoW's sequentiality, to accelerate solving SeqPoW puzzles, one can only increase the processor's frequency. While commodity processors usually achieve 2∼3 GHz frequency, the most advanced processor achieves the frequency of 8.723 GHz [189], which is hard to improve further due to the voltage limit [190]. Hence, the fastest processor can mine at most five times faster than normal processors, leading to $\mu > \frac{1}{5}$. The limited speedup is evidenced by the recent *VDF Alliance FPGA Contest* [191]–[193], where optimised VDF implementations are about four times faster than the baseline implementation.

The adversary can weaken μ to $\geq \frac{\mu}{2}$ by *selfish mining*, i.e., withholding and publishing blocks adaptively w.r.t. blocks from honest miners [47]. To defend against selfish mining attacks, one can deploy existing countermeasures [194]–[196], or finality protocols analysed in §5.8.3.

5.5.4 Extracting a random output from a block

Given block B, $\mathsf{randomOutput}(\cdot)$ extracts the random output $B.R$ via $\mathsf{VDF.Eval}(pp, B.pk \| B.S, t_{\mathsf{VDF}})$, and computes proof $B.\pi_R$ via $\mathsf{VDF.Prove}(pp_{\mathsf{VDF}}, B.pk \| B.S, B.R, t_{\mathsf{VDF}})$, where pp_{VDF} and t_{VDF} are VDF's public parameter and time parameter known to all participants, respectively. The time parameter t_{VDF} is chosen so that finishing $\mathsf{Eval}(\cdot)$ takes longer than participants extending $(\Upsilon + 1)$ blocks for a Υ-consistent RANDCHAIN.

The time delay in $\mathsf{randomOutput}(\cdot)$ ensures the unbiasibility of RANDCHAIN. If the random output is extracted from a block instantly, then the adversary can withhold its block if it does not like the extracted random output, compromising the unbiasibility. With the time delay of extending $(\Upsilon + 1)$ blocks, the adversary has to decide whether to broadcast or withhold its mined block before learning the random output. After learning the random output, the block either becomes irreversible (if the adversary broadcasts the block) or cannot be accepted anymore (if the adversary withholds the block).

RANDCHAIN satisfies uniform distribution: A λ-bit random string can be extracted from a block, where λ is SeqPoW and VDF's security parameter. RANDCHAIN satisfies unpredictability, as the sequentiality of SeqPoW and VDF implies their outputs are unpredictable as analysed in §5.4.1.

5.5.5 Security analysis

We prove RANDCHAIN (denoted as $\Pi_{RandChain}$ throughout the analysis) achieves all correctness properties defined in §5.2 when the network is synchronous and the adversary can corrupt $\alpha < \frac{1}{2}$ of participants. Let $\beta = 1 - \alpha$.

Consistency and liveness $\Pi_{RandChain}$ satisfies consistency and liveness when the network is synchronous and the adversary can corrupt $\alpha < \frac{1}{2}$ participants in the system. The analysis is identical to PoW-based Nakamoto consensus, where the adversary with mining rate α is competing with correct nodes with mining rate $\beta = 1 - \alpha$.

Uniform distribution We prove that each block derives a λ-bit uniformly distributed random string, where λ is the security parameter of SeqPoW and VDF.

Lemma 5.5.1. $\Pi_{RandChain}$ *satisfies uniform distribution.*

Proof. Each random output $B.R$ of $\Pi_{RandChain}$ is extracted from a block B via the VDF. By VDF's sequentiality, each VDF output contains non-negligible entropy that is unpredictable. A hash function can be applied to the VDF output to extract a λ-bit uniform random string [154]. □

Unpredictability In the prediction game, the $(\ell + 1)$-th block is either produced by correct participants or the adversary's participants. If the adversary's advantage is negligible for both cases, then $\Pi_{RandChain}$ satisfies *unpredictability*. When the $(\ell + 1)$-th block is produced by correct participants, the adversary's best strategy is guessing, leading to negligible advantage. When the $(\ell + 1)$-th block is produced by the adversary's participants, the adversary's best strategy is to produce as many blocks as possible before receiving a new block from the correct participants. First, consider $\Pi_{RandChain}$ using SeqPoW without *uniqueness*.

Lemma 5.5.2. *Assuming all messages are delivered instantly and participants agree on a blockchain of length ℓ. If the $(\ell+1)$-th block is produced by a correct participant, then the adversary's advantage on the prediction game is $\frac{1}{2^\kappa}$.*

If the next output is produced by the adversary's participants, the adversary's best strategy is to produce as many blocks as possible before receiving a new block from the correct participants. First, consider $\Pi_{RandChain}$ using SeqPoW without *uniqueness*.

Lemma 5.5.3. *Consider $\Pi_{RandChain}$ using SeqPoW without uniqueness. Assuming all messages are delivered instantly and participants agree on a blockchain of length ℓ. If the $(\ell+1)$-th block is produced by the adversary, then the adversary's advantage on the prediction game is $\frac{k}{2^\kappa}$ with probability $\alpha^k \cdot \beta$.*

Proof. The probability that the adversary and correct participants mine the next block are α and β, respectively. Note that $\alpha \leq \frac{1}{2}$ for satisfying consistency, and $\alpha + \beta = 1$.

Let V_k be the event that "the adversary mines k blocks at height $(\ell+1)$ before correct participants mine a block at height $(\ell+1)$". When SeqPoW is not unique, a participant can mine unlimited number of blocks after a single block. Thus, we have

$$\Pr[V_k] = \alpha^k \cdot \beta$$

When V_k happens, the adversary's advantage is $\frac{k}{2^\kappa}$.

Therefore, with probability $\alpha^k \cdot \beta$, the adversary mines k blocks before correct participants mine a block, leading to the advantage of $\frac{k}{2^\kappa}$. $\square$

Then, we analyse $\Pi_{RandChain}$ using SeqPoW with *uniqueness*. Without the loss of generality, we assume all participants share the same mining rate.

Lemma 5.5.4. *Consider $\Pi_{RandChain}$ using SeqPoW with uniqueness. Assuming all participants share the same mining rate, all messages are delivered instantly and participants agree on a blockchain of length ℓ. If the $(\ell+1)$-th block is produced* *by the adversary, then the adversary's advantage on the prediction game is* $\frac{k}{2^\kappa}$ *with probability* $\Pr[V'_k]$*, where*

$$\Pr[V'_k] = \prod_{i=0}^{k-1} \frac{(\alpha n - i)}{(\alpha n - i) + \beta n} \cdot \beta$$

Proof. Similar to Lemma 5.5.3, the adversary and the correct participants control mining rate α and β, respectively. When all participants share the same mining rate, the adversary and the correct participants control αn and βn participants, respectively. Let V'_k be the event that "the adversary mines k blocks at height $(\ell + 1)$ before correct participants mine a block at height $(\ell + 1)$", where $k \leq \alpha n$. By *uniqueness*, each participant can only mine a single block at height $(\ell + 1)$, and the adversary can mine at most αn blocks at height $(\ell + 1)$. Then, we have

$$\Pr[V'_0] = \beta \tag{5.12}$$

$$\Pr[V'_1] = \alpha \cdot \beta \tag{5.13}$$

$$\Pr[V'_2] = \frac{\frac{\alpha n - 1}{\alpha n}\alpha}{\frac{\alpha n - 1}{\alpha n}\alpha + \beta} \cdot \alpha \cdot \beta \tag{5.14}$$

$$\ldots \tag{5.15}$$

$$\Pr[V'_k] = \prod_{i=0}^{k-1} \frac{\frac{\alpha n - i}{\alpha n}\alpha}{\frac{\alpha n - i}{\alpha n}\alpha + \beta} \cdot \beta \tag{5.16}$$

$$= \prod_{i=0}^{k-1} \frac{(\alpha n - i)}{(\alpha n - i) + \beta n} \cdot \beta \tag{5.17}$$

When V'_k happens, the adversary's advantage is $\frac{k}{2^\kappa}$. Therefore, with probability $\Pr[V'_k] = \prod_{i=0}^{k-1} \frac{(\alpha n - i)}{(\alpha n - i) + \beta n} \cdot \beta$, the adversary mines k blocks before correct participants mine a block, leading to the advantage of $\frac{k}{2^\kappa}$ (where $k \leq \alpha n$). □

Remark 2. The adversary's advantage in $\Pi_{RandChain}$ with *unique* SeqPoW is always smaller than in $\Pi_{RandChain}$ with *non-unique* SeqPoW. That is, for every k, $\Pr[V'_k] < \Pr[V_k]$. Given k, we have

$$\frac{\Pr[V'_k]}{\Pr[V_k]} = \frac{\prod_{i=0}^{k-1} \frac{(\alpha n - i)}{(\alpha n - i) + \beta n} \cdot \beta}{\alpha^k \cdot \beta} \tag{5.18}$$

$$= \frac{\prod_{i=0}^{k-1} \frac{(\alpha n - i)}{(\alpha n - i) + \beta n}}{\alpha^k} \tag{5.19}$$

As $0 \leq i < \alpha n$, it holds that $\frac{\Pr[V'_k]}{\Pr[V_k]} < 1$ for all k.

Unbiasibility $\Pi_{RandChain}$ achieves unbiasibility by realising the *output-independent-abort* notion [162]. With a VDF with time parameter long than a new block becoming irreversible, the adversary has to decide whether to broadcast or withhold a block before learning the random output.

Lemma 5.5.5. $\Pi_{RandChain}$ *satisfies unbiasibility.*

Proof. The proof is by contradiction. Assuming participants agree on an ℓ-long blockchain, and the adversary learns the random output $B_{\ell+1}.R$ in the $(\ell + 1)$-th block $B_{\ell+1}$ when every correct participant's main chain contains less than $(\ell + \Upsilon + 1)$ blocks, where Υ is the consistency degree. Recall that extracting $B_{\ell+1}.R$ from $B_{\ell+1}$ is by evaluating a VDF with a time parameter longer than participants extending $(\Upsilon + 1)$ blocks on the blockchain. By VDF's sequentiality, to learn $B_{\ell+1}.R$, the adversary has to learn $B_{\ell+1}$ first. By SeqPoW's sequentiality, the adversary can learn $B_{\ell+1}$ only after learning its previous block B_ℓ, which is already agreed by participants. Thus, the adversary extracts $B_{\ell+1}.R$ from $B_{\ell+1}$ only after a correct participant grows its main chain from ℓ blocks to $(\ell + \Upsilon + 1)$ blocks if the adversary withholds $B_{\ell+1}$, and to $(\ell + \Upsilon + 2)$ blocks if the adversary publishes $B_{\ell+1}$, leading to a contradiction to the assumption. Therefore, $\Pi_{RandChain}$ achieves unbiasibility. □

Table 5.3: Experimental settings and results.

	Experimental setting				Experimental results	
	#nodes	#machines	Deployment	Network	Latency	Net. overhead
RandHerd [36]	1024	32	Datacenter	Simulated	3 sec	200 KB/s
HydRand [101]	128	128	Worldwide	Real	8.6 sec	180∼310 KB/s
RANDCHAIN	1024	128	Worldwide	Real	1.3 sec	200 KB/s

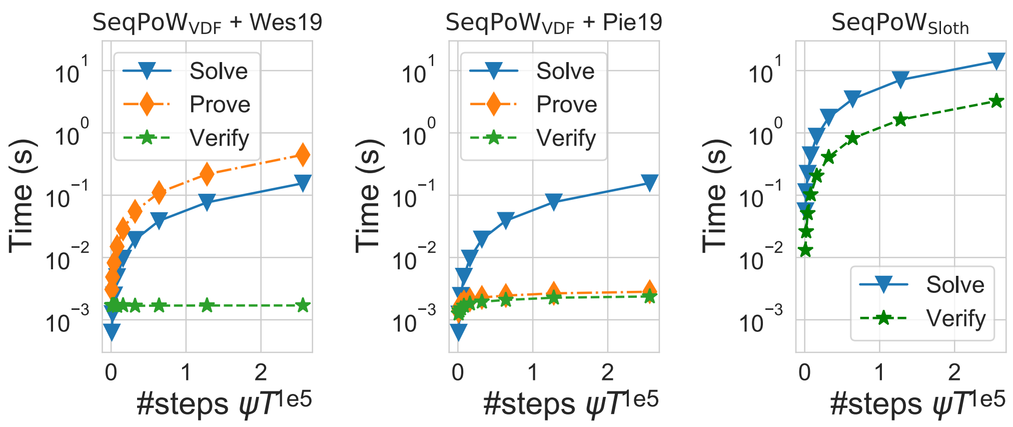


Figure 5.5: Evaluation of SeqPoW constructions.

5.6 Implementation and evaluation

We implement SeqPoW and RANDCHAIN, and evaluate their performance. The evaluation shows that all SeqPoW constructions are practical and RANDCHAIN is indeed scalable and fair. Specifically, on a cluster of 1024 nodes (each as a participant), RANDCHAIN can produce a random output every 1.3 seconds (2.3x faster than RandHerd [36] with 1024 nodes, 6.6x faster than HydRand [101] with 128 nodes); utilise constant bandwidth of about 200 KB/s per node (comparable with RandHerd with 1024 nodes and HydRand with 128 nodes); and provide nodes with comparable chance of producing random outputs. The code is available at `https://github.com/rand-chain/rust-randchain-prototype`.

5.6.1 SeqPoW: benchmarks

Implementation. We implement the SeqPoW constructions in Rust. We use `rug` [124] for big integer arithmetic, and implement the RSA group with 1024-bit keys and the group of prime order based on `rug`. We implement the two $\mathsf{SeqPoW}_{\mathsf{VDF}}$ constructions based on the RSA group, and $\mathsf{SeqPoW}_{\mathsf{Sloth}}$ based on the group of prime order. Our implementations strictly follow their original papers [108], [156], [157].

Experimental setting. For each function, we test ψT up to 256000, where ψ is the step parameter and T is the difficulty. The code for benchmarking is based on the `cargo-bench` [129] and `criterion` [130] benchmarking suites. We specify O3-level optimisation for compilation, and sample ten executions for each benchmarked function with a unique set of parameters. All experiments were conducted on a machine with a 2.2 GHz 6-Core Intel Core i7 Processor and a 16 GB 2400 MHz DDR4 RAM.

Performance (Figure 5.5). For all SeqPoW constructions, the running time of $\mathsf{Solve}(\cdot)$ increases linearly with ψT. This is as expected as $\mathsf{Solve}(\cdot)$ is dominated by the ISF. For $\mathsf{SeqPoW}_{\mathsf{VDF}}$ with Wes19, $\mathsf{Prove}(\cdot)$ takes more time than $\mathsf{Solve}(\cdot)$, making it less suitable for instantiating RANDCHAIN. For $\mathsf{SeqPoW}_{\mathsf{VDF}}$ with Pie19, $\mathsf{Prove}(\cdot)$ and $\mathsf{Verify}(\cdot)$ take negligible time compared to $\mathsf{Solve}(\cdot)$. For $\mathsf{SeqPoW}_{\mathsf{Sloth}}$, $\mathsf{Solve}(\cdot)$ is about five times slower than $\mathsf{Verify}(\cdot)$. Although this is far from the theoretically optimal value, i.e., $\log_2 |G| = 1024$ in our setting [197], the verification overhead is acceptable for the use case where random outputs are not generated frequently.

5.6.2 RANDCHAIN: end-to-end evaluation

We implement RANDCHAIN and evaluate it on computer clusters regarding the following metrics:

- **Block propagation delay (BPD)** is the time taken for the majority of nodes to receive a block (§5.6.2).
- **Block size** is the size of a block. It varies w.r.t. *blocktime* (i.e., the average time interval between two blocks) as the VDF proof size increases with the time parameter. We also estimate the network overhead of propagating blocks amortised by time (§5.6.2).
- **Network overhead** is the average bandwidth utilisation, i.e., the average amount of data transferred in a time unit, of a node (§5.6.2).
- **Decentralisation** is the evenness of nodes' chance of producing blocks. It is quantified by the distribution of nodes in terms of the number of blocks they produce on the main chain (§5.6.2).

Among the metrics, the former three are the empirical results of scalability (where BPD infers latency and the rest two infer network overhead); and decentralisation is the empirical result of fairness. We also compare RandChain with state-of-the-art DRBs that have experimental results, including RandHerd [36] and HydRand [101]. Table 5.3 summarises the evaluation results and comparison with RandHerd and HydRand.

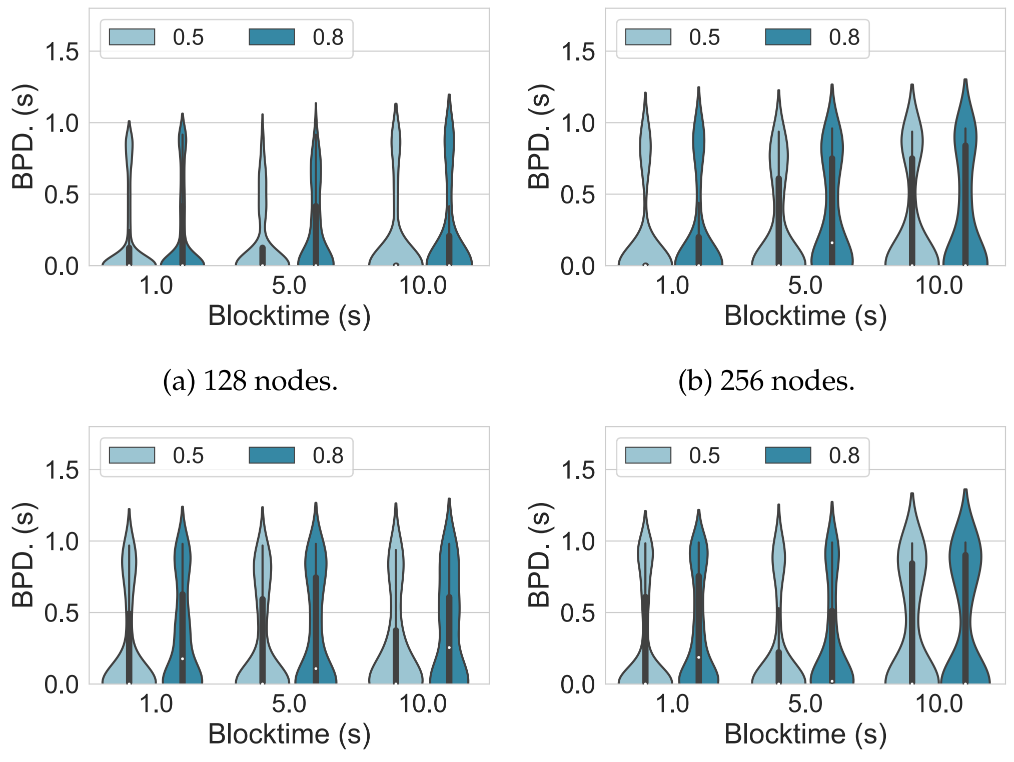
Implementation and experimental settings. We implement RandChain based on `Parity-bitcoin` [158], a Bitcoin implementation in Rust. Each node plays as a participant of RandChain. It uses `RocksDB` [198] for storage, and Bitcoin's `Wire` protocol [199] for the P2P protocol stack. We adapt the ledger structure, SeqPoW and relevant message types to RandChain's setting specified in §5.5. Given the evaluation result in §5.6.1, we use Pie19 for instantiating SeqPoW and extracting random outputs from blocks. The entire project takes approximately 23000 lines of code (LoC), where the RandChain implementation adds/changes approximately 4500 LoC over `Parity-bitcoin`. We use `dstat` [200] for monitoring system resource utilisation.

We specify O3-level optimisation for compilation, and deploy the project to clusters with $\{128, 256, 512, 1024\}$ nodes on Amazon's EC2 instances. Specifically, we deploy $\{16, 32, 64, 128\}$ `t2.micro` EC2 instances (1 GB RAM, one CPU core and 60-80 Mbit/s network bandwidth) in 13 regions around the globe[1], and each instance runs 8 RANDCHAIN nodes. Each node maintains up to 8 outbound connections and 125 inbound connections, which is same as Bitcoin's setting [199]. When a node starts, it randomly connects to 8 peers, accepts connections from other peers, and starts gossiping messages with them. As mining is not allowed in cloud computing platforms, we simulate $\mathsf{SeqPoW.Solve}(\cdot)$ rather than actually executing it. For our SeqPoW implementation, the `t2.micro` EC2 instance can do squaring operations in $\mathsf{SeqPoW.Solve}(\cdot)$ for 233868 times per second on average. We test blocktime of $\{1, 5, 10\}$ seconds by adjusting the SeqPoW difficulty. For each group of the experiments, we sample 30 minutes of the execution, collect logs from all nodes, parse the logs and calculate the metrics. The total size of logs is 1.74 GB.

Block propagation delay (BPD). Figure 5.6 shows the distribution of BPD for different sizes of clusters. First, with the increasing number of nodes (from 128 to 1024), the BPD never exceeds 1.3 seconds. In other words, the system can produce a random output every 1.3 seconds, which is 2.3x faster than RandHerd ($\sim$3 seconds on a 1024-node cluster) and 6.6x faster than HydRand ($\sim$8.6 seconds on a 128-node cluster). This is expected given the linear communication complexity.

Second, BPD is usually either less than 0.4 second or more than 0.6 second, but is hardly in-between values. This implies that a block can reach most nodes within 2 hops: The two peaks around the saddle of 0.4$\sim$0.6s indicate the average delays for 1-hop and 2-hop block propagation, respectively.

[1]The regions include North Virginia, North California, Oregon, Ohio, Canada, Mumbai, Seoul, Sydney, Tokyo, Singapore, Ireland, Sao Paulo, London, and Frankfurt.



(a) 128 nodes. (b) 256 nodes.

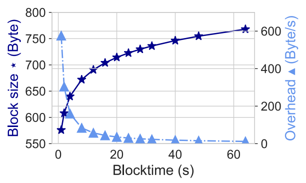
(c) 512 nodes. (d) 1024 nodes.

Figure 5.6: Distribution of block propagation delay (BPD), represented as *violin plots*. The light blue and dark blue parts indicate the distribution of BPD when blocks are propagated to 50% and 80% of nodes, respectively.

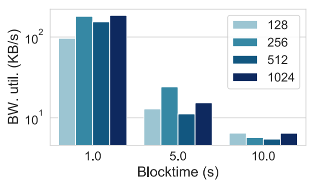
Third, the average BPD increases slowly with more nodes. This is consistent with other linear protocols [201]. In linear protocols, the average BPD is proportional to the average number of intermediate nodes of two random nodes. In Bitcoin's setting where each node connects to k random peers, the network is structured as an Erdos-Renyi random graph [202], in which two random nodes have $O(\frac{\log n}{\log k})$ intermediate nodes on average.

Last, BPD increases when blocks are produced more frequently. This is because a `t2.micro` instance only has a single processor and limited network capacity, making the overhead of verifying and propagating blocks non-negligible.

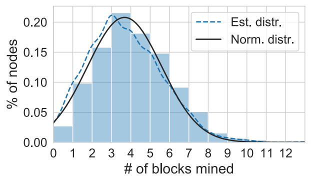
Block size. The major part of a block is the SeqPoW proof that takes $s \cdot \log_2(\psi T)$ Bytes, where ψT depends on the time taken to find a solution and the number of iterations executed in a time unit. Recall that the computer



(a) Block size and its estimated overhead.



(b) Network overhead.



(c) Decentralisation.

Figure 5.7: Block size, network overhead and decentralisation. **(a)** Block size and estimated network overhead between two nodes amortised by time v.s. blocktime. The dark blue increasing line is on the block size and the light blue decreasing line is on the overhead. **(b)** Network overhead, quantified as the bandwidth utilisation of each node with different blocktimes. **(c)** Decentralisation level, visualised as the number of blocks produced by distinct nodes. The blue and black lines are the kernel density estimation and the closest normal distribution, respectively.

can do squaring operations for 233868 times per second. Given blocktime t, the SeqPoW proof size is $s \cdot \log_2(233868 \cdot t) \approx s \cdot (18 + \log_2 t)$, and the network overhead between two nodes amortised by time is $\frac{s\cdot(18+\log_2 t)}{t}$. Figure 5.7a shows the relationship between blocktime, block size and network overhead. When blocktime is $\{1, 5, 10\}$ seconds and $s = 32$ Bytes, the block size is about $\{576, 1336, 1912\}$ Bytes, and the amortised network overhead is about $\{576, 267, 191\}$ Bytes/s. When blocktime is 60 seconds (the setting of Drand [203] and the NIST randomness beacon [100]), the block size is about 3402 Bytes, and the amortised network overhead is about 57 Bytes/s.

Network overhead. Figure 5.7b shows the bandwidth utilisation result. It shows that RANDCHAIN utilises less bandwidth compared to RandHerd and HydRand: Even with blocktime of 1 second, each node utilises ∼200KB/s bandwidth per second, which is comparable with RandHerd (∼200KB/s on a 1024-node cluster) and HydRand (180∼310KB/s on a 128-node cluster). The bandwidth utilisation remains stable with more nodes, as RANDCHAIN is linear. These two results are as expected since RANDCHAIN is linear. The inbound and outbound bandwidths are identical, as the input (i.e., the last block) and the output (i.e., the new block) are identical in terms of size, leading to identical bandwidth utilisation. With longer blocktime, the node requires less bandwidth, as nodes send and receive blocks less frequently.

Decentralisation. Figure 5.7c shows the distribution of nodes w.r.t. the number of blocks they produce on the main chain, in the experiment with 1024 nodes and the blocktime of 1 second. The kernel estimated distribution is close to the normal distribution, meaning that nodes have comparable chance of producing blocks, similar to RandHerd and HydRand that are "one-man-one-vote". The result is consistent with our experimental setting where nodes use the same processors.

5.7 Comparison with existing DRBs

In this section, we develop a unified evaluation framework for DRBs, and compare RANDCHAIN with existing DRBs. Our evaluation shows that RANDCHAIN is the only protocol that is secure, scalable and fair simultaneously, without relying on any trusted party.

5.7.1 Overview of existing DRBs

DRG-based DRBs. Participants execute the single-shot Distributed Randomness Generation (DRG) protocol periodically. DRG can be constructed from various cryptographic primitives, such as threshold cryptosystems [99], [142], [143], Verifiable Random Functions (VRFs) [144]–[146], and Publicly Verifiable Secret Sharing (PVSS) [36], [101], [107], [147]–[149]. To relax the network model assumptions, reduce the communication complexity and/or improve the fault tolerance capacity, these DRBs usually rely on a centralised dealer [99], [107], [143], [148] and/or combine techniques such as leader election [36], [142], [144]–[147], sharding [36], [142], cryptographic sortition [145], Byzantine consensus [101], [145] and erasure coding [107], [148].

Table 5.4: Comparison of RANDCHAIN with existing DRBs.

	Protocol		System model			Correctness							Performance	
	Name	Primitives	Net. model	Trust	Fault tol. cap.	Consistency	Liveness	Fairness	Uniform dist.	Unpredictability	Unbiasibility	Pub. ver.	Comm. compl.	Latency
DRG-based DRBs	Cachin et al. [99]	Thr. Sig.	Async.	Dealer‡	1/3	✓	✓	1	✓	✓	✓	✓	$O(n^3)$	$O(\delta)$
	HERB [143]	Homo. Thr. Enc.	Part. sync.	Dealer‡	1/3	✓	✓	1	✓	✓	✓	✓	$O(n)$	$O(\delta)$
	Dfinity [142]	VRF + Thr. Sig.	Sync.	-	1/3	✓	✓	1	✓	✓	✗†	✓	$O(cn)^{\diamond¶}$	$O(\Delta) \sim \infty^{¶}$
	Ouro. Praos [144]	VRF	Part. sync.	-	1/2	✓	✓	1	✓	✓	✗†	✓	$O(n)^{¶}$	$O(\Delta) \sim \infty^{¶}$
	GLOW [146]	VRF	Sync.	-	1/3	✓	✓	1	✓	✓	✗†	✓	$O(n)^{¶}$	$O(\delta) \sim \infty^{¶}$
	Algorand [145]	VRF	Sync.	-	1/3	✓	✓	1	✓	✓	✗†	✓	$O(cn)^{\diamond¶}$	$O(\Delta) \sim \infty^{¶}$
	Ouroboros [147]	PVSS	Sync.	-	1/2	✓	✓	1	✓	✓	✓	✓	$O(n^3)$	$O(\Delta)$
	SCRAPE [148]	PVSS	Part. sync.	Dealer‡	1/2	✓	✓	1	✓	✓	✓	✓	$O(n^3)$	$O(\delta)$
	RandShare [36]	PVSS	Async.	-	1/3	✓	✓	1	✓	✓	✓	✓	$O(n^3)$	$O(\delta)$
	RandHound [36]	PVSS	Sync.	-	1/3	✓	✓	1	✓	✓	✗†	✓	$O(c^2n)^{\diamond¶}$	$O(\Delta) \sim \infty^{¶}$
	RandHerd [36]	PVSS	Sync.	Dealer‡	1/3	✓	✓	1	✓	✓	✓	✓	$O(c^2 \log n)^{\diamond}$	$O(\delta)$
	HydRand [101]	PVSS	Sync.	-	1/3	✓	✓	1	✓	✓	✓	✓	$O(n^2)$	$O(\Delta)$
	Albatross [107]	PVSS	Part. sync.	Dealer‡	1/2	✓	✓	1	✓	✓	✓	✓	$O(n)$	$O(\delta)$
	Kogias et al. [149]	HAVSS	Async.	-	1/3	✓	✓	1	✓	✓	✓	✓	$O(n^4)$	$O(\delta)$
SC-based DRBs	RanDAO [40]	VDF	Part. sync.^{x}	Blockchain^x	1/2^x	✓	✓	1	✓	✓	✓	✓	$O(n)$	$t_{\text{block}} + \delta$
	Yakira et al. [204]	Escrow-DKG	Part. sync.^{x}	Blockchain^x	1/3^x	✓	✓	1	✓	✓	✓	✓	$O(n)$	$t_{\text{block}} + \delta$
ISF-based DRBs	Unicorn [108]	Sloth	Async.	Setup	(n-1)/n	✓	✓	$\rightarrow 0^{\otimes}$	✓	✓	✓	✓	$O(n)$	Any $+\delta$
	Ephraim et al. [109]	Continuous VDF	Async.	Setup	(n-1)/n	✓	✓	$\rightarrow 0^{\otimes}$	✓	✓	✓	✓	$O(n)$	Any $+\delta$
	RandRunner [161]	Trapdoor VDF	Async.	Setup	1/2	✓	✓	1	✓	✗⋆	✓	✓	$O(n) \sim O(n^2)$	Any $+\delta$
DRBs from ext. entr.	Clark et al. [112]	Rand. extractors	Async.	Data src.	(n-1)/n	✓	✓	-	✓	✓	✓	✗II	$O(n)$	Any $+\delta$
	Bonneau et al. [111]	Rand. extractors	Async.^{x}	Blockchain^x	(n-1)/n^x	✓	✓	$\rightarrow 0^{\otimes}$	✓	✓	✓	✗II	$O(n)$	$t_{\text{block}} + \delta$
	Bünz et al. [205]	Proof-of-Delay	Async.^{x}	Blockchain^x	(n-1)/n^x	✓	✓	$\rightarrow 0^{\otimes}$	✓	✓	✓	✗II	$O(n)$	$t_{\text{block}} + \delta$
This work	RANDCHAIN	SeqPoW + Nak. consensus	Sync.	-	1/2	✓	✓	$> \frac{1}{5}$	✓	✓	✓	✓	$O(n)$	$t_{\text{block}} + \delta$

‡ The analysis assumes the dealer is a trusted third party. While the dealer can be implemented in a distributed manner [150], it introduces extra communication overhead.
† The corrupted leader can withhold the random output and enforce participants to start a new round, as analysed in [101], [107].
⋄ We use c to denote the size of shards in Dfinity [142], RandHound and RandHerd [36], and the size of the committee in Algorand [145].
¶ The corrupted leader can send the random output and advance the round for a subset of participants, so that participants are executing different rounds. The DRB requires an extra round synchronisation protocol that suffers from either exponential latency [153] or worst-case communication complexity of $\geq O(n^2)$ [151], [152].
⋆ The adversary can always corrupt leaders and produce random outputs efficiently via the trapdoor.
II Entropy generated by the external source is not verifiable.
⊗ In Unicorn and Ephraim et al., the participant with the fastest processor can always propose random outputs earlier than other participants. In DRBs with PoW-based blockchains as external entropy, mining can be accelerated by using parallelism. Both cases weaken the fairness degree to near zero.
x These DRBs are usually built upon public blockchains. When considering the public blockchain as a part of the DRB, the system model will also respect that of the public blockchain. For example, the DRB may be built upon Ethereum, which requires synchronous networks and fault tolerance capacity $\alpha < \frac{1}{2}$.

Other types. In Smart contract (SC)-based DRBs [40], [204], [206], participants submit their inputs to an external smart contract, which combines them to a single random output. In DRBs from external entropy, participants periodically extract randomness from real-world entropy, e.g., real-time financial data [112] and public blockchains [111], [205], [207]. In Iteratively sequential function (ISF)-based DRBs [108], [109], [161], participants use intermediate outputs of an ISF as random outputs, and use succinct proofs for the ISF to make outputs verifiable.

5.7.2 Evaluation framework for DRBs

We extend our model in §5.2 to build an evaluation framework for DRBs. Apart from synchronous networks in §5.2.1, the framework additionally considers *partially synchrony* [17] where messages are delivered within a known finite time-bound Δ after an unknown Global Stabilisation Time (GST) and *asynchrony* where messages are delivered without a known time bound. Apart from system model aspects in §5.2.1, the framework also concerns trust assumptions that some proposals assume in order to guarantee correctness properties. Apart from the correctness properties in §5.2.2, the framework also concerns *fairness* and *public verifiability*: Whether a random output is publicly verifiable.

5.7.3 Evaluation

Table 5.4 summarises the evaluation results. Let Δ be the network latency bound in the synchrony period, δ be the actual network delay, and GST be the global stabilisation time.

System model. Most DRG-based DRBs employ synchronous leader election protocols, except for the following proposals. Cachin et al., RandShare and Kogias et al. employ randomised common coin techniques to achieve asynchrony. Ouroboros Praos allows "empty slots" (where participants produce no block) when no leader is elected before GST, and guarantees an elected leader after GST, leading to partial synchrony. HERB, SCRAPE, and Albatross employ a dealer who relays all messages and proceeds the protocol whenever receiving enough shares, which is guaranteed after GST, leading to partial synchrony. These DRBs have to trust the dealer, otherwise a corrupted dealer can selectively multicast messages to allow a subset of nodes to predict random outputs, or withhold messages to bias random outputs. While the dealer can be implemented in a distributed manner [150], it introduces extra communication overhead. SC-based DRBs rely on a permissionless blockchain to achieve partial synchrony. The blockchain is assumed to be trusted, otherwise a corrupted blockchain can censor transactions to bias random outputs, which is known as the Miner Extractable Value (MEV) issue [208]. ISF-based DRBs and DRBs from external entropy proceed as long as a single participant is honestly executing the ISF or sampling the entropy, except for RandRunner which requires a reliable broadcast with fault tolerance degree $\alpha < \frac{1}{2}$. ISFs require a trusted setup, otherwise the adversary who previously knows the seed can learn random outputs earlier than other participants. The entropy source has to be trusted, otherwise the adversary can manipulate the entropy and bias random outputs. In DRBs based on permissionless blockchains, the blockchains usually employ Nakamoto-style consensus and thus assume synchronous networks. If the blockchain-based DRBs allow nodes to run a blockchain protocol on their own, then it incurs more communication overhead.

Correctness properties. All DRG-based DRBs achieve consistency and liveness. Note that DRG-based DRBs define liveness as termination (where correct participants eventually learn the random output at the end of each round), and our evaluation of DRG-based DRBs follows such definition. All DRBs achieve the ideal fairness, i.e., $\mu = 1$, except for DRBs from PoW-based blockchains [111], [205], Unicorn [108], Ephraim et al. [109] and RANDCHAIN. DRBs from PoW-based blockchains allow accelerating mining by parallelism. For Unicorn and Ephraim et al., the participant with the fastest processor can always propose random outputs earlier than other participants. Both cases weaken the fairness degree to near zero. RANDCHAIN achieves $\mu > \frac{1}{5}$ by making the mining process unpredictable [120], [166] and non-parallelisable, as analysed in §5.5.2-5.5.3. All DRBs satisfy uniform distribution and unpredictability, except for RandRunner [161] where the adversary can keep corrupting leaders and computing random outputs efficiently via the trapdoor, breaking unpredictability. In Dfinity, Ouroboros Praos, GLOW, Algorand and RandHound, the corrupted leader can withhold the random output and enforce participants to start a new round, breaking the *unbiasibility*, as analysed in [101], [107]. DRBs from external entropy do not satisfy public verifiability, as the external entropy is not publicly verifiable.

Performance metrics. In all dealer-less DRG-based DRBs, either the leader election, view change or PVSS protocol requires the all-to-all broadcast operations, leading to at least $O(n^2)$ communication complexity. To reduce communication complexity, HERB, RandHerd and Albatross employ a dealer to relay messages; GLOW allows participants to determine a unique leader locally given the last random output; Dfinity, RandHound and RandHerd apply sharding techniques to divide participants into different shards; Algorand samples a subset of participants to execute the protocol; and SC-based DRBs rely on a smart contract that relays all messages. RandRunner is linear in the best case, but requires reliable broadcasts with $O(n^2)$ communication complexity in the worst case. The other two ISF-based DRBs and DRBs from external entropy achieve $O(n)$ communication complexity.

Asynchronous DRG-based DRBs terminate within $O(\delta)$, as asynchronous networks do not have Δ. In HERB, SCRAPE, RandHerd and Albatross, the random output is produced once the dealer receives enough shares, leading to the latency of $O(\delta)$. In Ouroboros and HydRand, the leader election terminates in $O(\Delta)$. In GLOW, when the leader is correct, the latency is $O(\delta)$. When the leader is corrupted, then it can deliver random outputs and advance the round for a subset of participants, so that participants will execute different rounds. To re-synchronise the round, nodes have to execute an extra round synchronisation protocol with either exponential latency (by using the time doubling mechanisms [153]) or at least $O(n^2)$ worst-case communication complexity (by using the broadcast-based mechanisms [151], [152]). In Dfinity, Ouroboros Praos, Algorand, and RandHound, the leader election terminates within $O(\Delta)$, and a corrupted leader can cause the round synchronisation issue similar in GLOW. SC-based DRBs and DRBs from blockchain entropy achieve the latency of the parameterised block interval t_{block} plus δ. ISF-based DRBs and DRBs from other entropy can achieve any latency plus δ, according to the frequency of sampling intermediate outputs and entropy, respectively.

5.8 Limitations and resolutions

We discuss three limitations and the corresponding resolutions for RANDCHAIN, including the energy-efficiency, churn tolerance and finality support. We consider the concrete resolutions and analysis as future work.

5.8.1 Energy efficiency

As RANDCHAIN requires all nodes to solve SeqPoW puzzles to produce a random output, RANDCHAIN seems to be less energy-efficient than existing DRG-based DRBs. In fact, whether RANDCHAIN is less energy-efficient than existing DRG-based DRBs remains arguable. In terms of communication, RANDCHAIN costs strictly less energy than DRG-based DRBs, which require at least $O(n^2)$ communication complexity. The energy cost of communication is not always less than that of computation, as shown by existing literature [209]. In terms of computation, it remains arguable whether computing a random output through threshold cryptographic primitives (which can involve computationally intensive operations such as pairing, Lagrange interpolation, and Zero Knowledge Proofs) is more energy-efficient than non-parallelisable mining, where every node executes a single SeqPoW instance. In addition, with shorter block intervals, the energy cost by computing a random output in RANDCHAIN reduces linearly, while that in collaborative DRBs remains constant. We consider the energy efficiency analysis and improvement of RANDCHAIN as future work.

5.8.2 Churn tolerance

Similar to existing DRBs, RANDCHAIN does not tolerate churn, i.e., nodes joining and leaving. However, with little modifictaions, RANDCHAIN can tolerate churn like PoW-based consensus protocols. To tolerate churn [77], PoW-based blockchains adjust difficulty parameters adaptively to the rate of new blocks. In RANDCHAIN, the difficulty adjustment mechanism can use the number i of iterations running $\mathsf{SeqPoW.Solve}(\cdot)$ to infer the historical block rate. If historical values of i are large, then this means that mining is too hard and the difficulty should be reduced, and vice versa. We consider a concrete construction and analysis on the difficulty adjustment mechanism as future work.

5.8.3 Finality

Due to the probabilistic Nakamoto consensus, RANDCHAIN does not achieve finality, and an appended block may be reverted later. A block being reverted does not lead to financial loss, as the random output is revealed only after the block becomes stable, guaranteed by the unbiasibility property. However, when a block is reverted, some randomness-based applications may abort the execution. We consider two approaches to achieve finality, namely the quorum mechanism and herding-based consensus, and consider concrete constructions and analysis as future work.

Quorum mechanism Quorum [210] is the minimum number of votes that a proposal has to obtain for being agreed by nodes. A vote is usually a digital signature with some metadata, and a quorum of votes is called a *quorum certificate*. The quorum size is $n - f$, where n and f be the number of nodes and faulty nodes in the system, respectively. Achieving agreement in synchronous networks and partially synchronous networks require $n \geq 2f + 1$ and $n \geq 3f + 1$, respectively [17], [210].

RANDCHAIN can employ the quorum mechanism as follows. A node signs a block to vote it. A node's view is represented as the latest block hash. Nodes proactively propagate their votes, i.e., signatures on blocks. A node finalises a block if collecting a quorum certificate, i.e., $\geq 2f + 1$ votes, on this block. The fault tolerance assumption changes to $n \geq 3f + 1$. RANDCHAIN still keeps Nakamoto consensus as a fallback solution. If there are multiple forks without quorum certificates, nodes mine on the longest fork. A block can be considered finalised with a sufficiently long sequence of succeeding blocks, even without a quorum certificate.

Herding-based consensus *Herding* is a social phenomenon where people make choices according to the choices of other people. Herding-based consensus allows nodes to decide proposals according to neighbour nodes' votes only, rather than a quorum of votes. Existing research [187], [211] shows that, herding-based consensus can achieve agreement with overwhelming probability in a short time period.

RANDCHAIN can employ herding-based consensus as follows. Upon a new block, nodes execute a herding-based consensus on it. If a block is the only block in a long time period, then nodes will agree on this block directly. If there are multiple blocks within a short time period, then nodes will agree on the most popular block among them with overwhelming probability. This approach has also been discussed in Bitcoin Cash community, who seeks to employ Avalanche [187] as a finality layer for Bitcoin Cash [212].

Chapter 6

Fair delivery of Decentralised Randomness Beacons

6.1 Introduction

Decentralised Randomness Beacon (DRB) is a protocol where a set of participants jointly generates a sequence of random outputs. It has been a promising approach to provide secure randomness to other protocols and applications. There have been emerging DRB proposals [101], [203], [213], [214] and deployed DRB systems [215], [216], and DRBs have been used by many high-financial-stake applications such as blockchains [113], [142], [144], [147], lotteries [217], games [218], [219], and non-fungible tokens (NFTs) [220], [221].

Applications have two common approaches to use a DRB, namely 1) by using a random output at a certain height which the DRB has not reached yet, and 2) by using a random output produced near a certain time in the future. For example, Polygon Hermez [222] and Celo [223] used the 697500-th random output [224] and the random output produced near 29/10/2021 9am UTC [225] of Drand [203] for their zkSNARK trusted setup, respectively.

Existing DRBs are designed with three main security properties in consideration, namely *consistency*, *liveness* and *unpredictability* [213], [214]. Consistency states that all correct participants (who generate random outputs) share the same view on a unique ledger, i.e., sequence of random outputs. Liveness states that all correct participants produce random outputs no slower than a certain rate. Unpredictability states that no participant can distinguish a future random output from a uniformly sampled random string of the same length.

None of these properties captures the advantage that some participants learn random outputs earlier than other participants. Such an advantage, however, is not desired in practice. In time-sensitive protocols whose execution depends on the randomness from a DRB, the advantage allows an adversary to behave adaptively according to random outputs, compromising the fairness and/or security in these protocols. In the above example of zkSNARK trusted setup, if the adversary learns the random output before the trusted setup starts, then Hermez's trusted setup will allow the last participant to decide some properties of the agreed parameter [224], and Celo's trusted setup will be insecure against an adaptive adversary [226]. Another example is the on-chain lottery which determines the winner out of all players by using random outputs from a DRB. If the adversary learns the random output before the lottery starts, then it learns whether it will win the lottery in advance, and thus can decide whether to participate in the lottery according to its outcome.

However, a systematic and formal study on this advantage is still missing. Existing DRB models only focus on certain attacks leading to certain aspects in this advantage [144], [213]. Similar security properties in other primitives cannot be adapted to capture this advantage directly, as they either concern eventual delivery without quantifying the advantage [227], or concern the advantage among correct participants excluding Byzantine participants [52], [163], [181], [228].

6.1.1 Contributions

In this chapter, we initiate the study of *delivery-fairness*, the security property capturing the advantage that some participants learn random outputs earlier than other participants in DRBs. We formalise delivery-fairness, prove its lower bound, and analyse the delivery-fairness guarantee of state-of-the-art DRBs, including Drand [203], HydRand [101], GRandPiper [213] and SPURT [214]. Through the analysis, we identify attacks on delivery-fairness and obtains several insights for improving delivery-fairness. The insights allow us to suggest lock-step variants for HydRand and SPURT with better delivery-fairness (where SPURT achieves the optimal value), without affecting system models or security properties. Table 6.1 summarises our results.

Delivery-fairness and its lower bound (§6.3). We base our study on existing DRB models [146], [213], [214]. As specified in §6.2, we consider a fixed set of n participants and an adversary who can corrupt up to f participants among them at the beginning, where f is a corruption parameter subjected to the protocol design. The network is synchronous, where messages are delivered in at least the actual network latency δ and at most a known upper bound Δ. Participants jointly execute the DRB protocol to agree on a unique ledger containing a sequence of random outputs securing three properties, namely consistency, liveness and unpredictability.

Atop the DRB model, we provide the first formal definition of delivery-fairness in §6.3. The delivery-fairness concerns two aspects of advantage, namely the *length-advantage*, i.e., how many random outputs an adversary can learn earlier than correct participants, and the *time-advantage*, i.e., how much time an adversary can learn a given random output earlier than correct participants.

Definition 6.1.1 (Delivery-fairness, informal; formalised in Definition 6.3.1). A DRB protocol satisfies (ω, ψ)-delivery-fairness if the following holds for any two participants (p_i, p_j) and any time t, except for negligible probability:

- **ω-length-advantage**: At time t, p_i's ledger pruning the last ω random outputs p_j's ledger; and

- **ψ-time-advantage**: Participant p_i's ledger at time t precedes or is equal to p_j's ledger at time $t + \psi$.

When ω and ψ are smaller, the length-advantage and time-advantage of any participant over the other participants is smaller, thus the DRB provides stronger delivery-fairness guarantee. We stress that delivery-fairness is achievable in synchronous networks, where messages are delivered in at least the actual network latency δ and at most a known upper bound Δ. Otherwise, the adversary can arbitrarily delay messages in asynchronous networks or the asynchrony period in partially synchronous networks, increasing ω and ψ to values that are impractical.

We then prove the lower bound of delivery-fairness in synchronous networks. The intuition behind the proof is that, if the time difference of learning a random output between any two participants is smaller than $\Delta - \delta$, then the group of Byzantine participants can produce valid random outputs without communicating with correct participants, contradicting to unpredictability or consistency.

Theorem 6.1.1 (Delivery-fairness lower bound of DRB, informal; formalised in Theorem 6.3.1)**.** *There does not exist a correct DRB protocol that achieves (ω, ψ)-delivery fairness with $\omega < 1$ or $\psi < \Delta - \delta$ under synchronous networks.*

Analysis of delivery-fairness of existing DRBs (§6.4-6.6). With the formalisation, we analyse the delivery-fairness of state-of-the-art DRB protocols, namely Drand [203], HydRand [101], GRandPiper [213] and SPURT [214], in §6.4-6.6. Table 6.1 summarises the results. Through the analysis, we identify attacks on delivery-fairness and obtains several insights for improving delivery-fairness. Specifically, we identify a new attack called *latency manipulation attack* that can compromise the delivery-fairness in the original versions of Drand, HydRand and SPURT, i.e., making ω and ψ too large to be practical. This attack is rooted in their non-lock-step design that participants

Table 6.1: Summary of evaluation results under synchronous networks.

	Protocol	No DKG	Fault tolerance	Comm. compl.		Latency		(ω, ψ)-Delivery-fairness¶
				Best	Worst	Best	Worst	
Existing work	Drand [203]	✗	$n \geq 2f+1$	$O(n^2)$	$O(n^2)$	δ	Δ	(∞, ∞)
	Lock-step Drand [229]	✗	$n \geq 2f+1$	$O(n^2)$	$O(n^2)$	Δ	Δ	$(1, \Delta - \delta)$
	HydRand [101]	✓	$n \geq 3f+1$	$O(n^2)$	$O(n^3)^\dagger$	3δ	3Δ	(∞, ∞)
	GRandPiper [213]	✓	$n \geq 2f+1$	$O(n^2)$	$O(n^2)$	11Δ	11Δ	$(c+1, (11c+1)\Delta - \delta)^\ddagger$
	SPURT [214]	✓	$n \geq 3f+1$	$O(n^2)$	$O(n^2)$	7δ	$(f+7)\Delta^*$	(∞, ∞)
This chapter	Lock-step HydRand	✓	$n \geq 3f+1$	$O(n^2)$	$O(n^3)^\dagger$	3Δ	3Δ	$(c+1, (3c+1)\Delta - 3\delta)^\ddagger$
	Lock-step SPURT	✓	$n \geq 3f+1$	$O(n^2)$	$O(n^2)$	7Δ	$(f+7)\Delta^*$	$(1, \Delta - \delta)$

¶ In (ω, ψ)-delivery-fairness (Definition 6.3.1), the delivery-fairness is better when ω and ψ are smaller. When $\omega = 1$ and $\psi = \Delta - \delta$, the delivery-fairness is optimal. When $\omega, \psi = \infty$, the DRB is considered to not satisfy delivery-fairness. Here δ and Δ are the actual network latency and the latency upper bound, respectively.
† In the worst case, the adversary does not reveal its committed secrets for f consecutive epochs. In the next epoch, the correct leader needs to broadcast f $O(n)$-size recovery certificates to all participants, leading to $O(n^3)$ communication complexity.
‡ Value c is a random variable meaning that "c consecutive leaders are Byzantine". In the round-robin leader election used in HydRand and GRandPiper, the best, average and worst cases of c are 0, 2, and f, respectively.
$*$ In the worst case, the adversary controls f consecutive leaders and aborts these f consecutive epochs before a correct epoch with 7Δ, leading to $(f+7)\Delta$ worst-case latency.

can make progress once receiving sufficient messages, without the need of waiting for a fixed time period. Following this observation, we suggest lock-step variants for HydRand and SPURT that resist against the latency manipulation attack and thus achieve the better delivery-fairness (where SPURT achieves the optimal value), without affecting system models or security properties. In addition, a previously known unpredictability-focused attack, which we call *private beacon attack*, can also weaken the delivery-fairness of HydRand and GRandPiper. The private beacon attack is rooted in their design that the epoch leader solely samples the entropy for the random output. To resist against the attack, the entropy should instead be sampled by a group of at least $f+1$ participants, where f is the number of Byzantine participants.

6.2 Model

In this section, we provide the model of a DRB. The model is adapted from existing collaborative DRB proposals [101], [213], [214], and is different from the model in §5.2, which focuses on competitive DRBs like RANDCHAIN.

6.2.1 System model

Participants. We consider a fixed number of n participants. Each participant $p_k \in [p_n]$ generates a pair of secret key and public key (sk_k, pk_k), and is uniquely identified by its public key in the system. We assume each participant has the knowledge of other participants' public keys.

Adversary. We consider a static adversary $\mathcal{A}$. In the beginning of the protocol, $\mathcal{A}$ can corrupt at most f participants, where f is a corruption parameter subjected to the protocol design. After that, $\mathcal{A}$ cannot change the set of corrupted participants or corrupt more participants. $\mathcal{A}$ fully controls corrupted participants, including observing the participant's internal state and controlling its messages and outputs, without any latency. $\mathcal{A}$ can read all messages between participants, but cannot modify or drop messages sent by correct participants. We also refer to a corrupted participant as *Byzantine* participant. We assume $\mathcal{A}$ is probabilistically polynomial-time (PPT), and thus cannot break standard cryptographic primitives.

Network model. We assume synchronous networks: $\mathcal{A}$ can decide to delivery any message in at least the actual network delay δ and at most a known upper bound Δ. In practice, $\delta << \Delta$.

We will conduct analysis assuming synchronous networks for all DRBs, although some of them can work in relaxed network models. The reason is that the delivery-fairness is a concrete measure and is meaningful only in synchronous networks. Otherwise, in asynchronous networks or the asynchrony period in partially synchronous networks, the adversary can arbitrarily delay messages to increase its advantage, leading to impractical delivery-fairness guarantee. Consequently, applications that require time-sensitive random outputs will be insecure or unfair. Thus, when the application scenarios demand a delivery-fair DRB, the application and DRB have to be deployed in synchronous networks.

6.2.2 Components of DRBs

The set of n participants continuously execute the DRB protocol Π to produce a sequence of random outputs. Specifically, participants jointly produce and agree on a ledger formed as a sequence of blocks. Each agreed block has to meet a verification predicate $F_V(\cdot)$. Each block deterministically derives a random output, which can be extracted by a random output extraction function $F_R(\cdot)$. Both $F_V(\cdot)$ and $F_R(\cdot)$ are accessible to anyone inside and outside the system, and their instantiation depends on the concrete protocol design.

Ledger. A ledger T is formed as a sequence of blocks. Let $T[e]$ be the e-th block in the ledger T. Let $|T|$ be the length of ledger T. Let $T[p : q]$ be the ledger from p-th block to $(q-1)$-th block of ledger T. Parameter p and q can be set empty, indicating the beginning and the end of the ledger, respectively. Let $T^{\lceil \ell} = T[: \ell]$ be the ledger from pruning the last ℓ blocks of ledger T. We denote a ledger T is a prefix of or equals to another ledger T' as $T \preceq T'$.

Epoch. DRBs are executed in epochs. In each epoch, participants are expected to produce and agree on a new block. The time period of an epoch can be fixed by the protocol design, or be variant depending on Byzantine behaviours and/or actual network delay. In leader-based DRBs, in each epoch, a leader is elected to drive the protocol execution. In some leader-based DRBs (e.g., SPURT [214]), a Byzantine epoch leader can abort the protocol, so that no block is produced in that epoch.

Verification predicate $F_V(\cdot)$. To be agreed, a block has to meet the verification predicate $F_V(\cdot)$. In $F_V(\kappa, T, B) \to \{0, 1\}$, given security parameter κ, ledger T and block B as input, outputs 1 if B is a valid successor block of T. A ledger T is valid in κ if for all $\ell \in [|T|-1]$, $F_V(\kappa, T^{\lceil \ell}, T[\ell]) = 1$. Let $\mathcal{T}_i^t$ be participant p_i's longest valid ledger at time t.

Random output extraction function $F_R(\cdot)$. Each block contains a random output, which can be extracted by the random output extraction function $F_R(\cdot)$. In $F_R(\kappa, T) \to R_e$, given security parameter κ and a ledger T of length $|T| = (e-1)$ as input, $F_R(\kappa, T)$ can derive a random output R_e. That is, every block B_e is associated with a random output R_e.

6.2.3 Security properties of DRBs

A DRB protocol Π should satisfy the following properties, namely consistency, liveness, and unpredictability.

Consistency. Consistency ensures that correct participants agree on a unique ledger, and thus a unique sequence of random outputs. The consistency definition follows the common prefix property in blockchain protocols [18].

Definition 6.2.1 (ℓ-consistency, from [18])**.** For any κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds except for probability $\mathsf{negl}(\kappa)$. For any two correct participants p_i and p_j ($i = j$ is possible) at time t,

$$(\mathcal{T}_i^t)^{\lceil \ell} \preceq \mathcal{T}_j^t \vee (\mathcal{T}_j^t)^{\lceil \ell} \preceq \mathcal{T}_i^t$$

Liveness. Liveness ensures that correct participants produce new random outputs at an admissible rate. The liveness definition follows the chain growth property in blockchain protocols [230].

Definition 6.2.2 ((t, τ)-liveness, from [230])**.** For any κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds except for probability $\mathsf{negl}(\kappa)$. For any honest participant p_i and time $t' \geq t$,

$$|\mathcal{T}_i^{t'}| - |\mathcal{T}_i^{t'-t}| \geq t \cdot \tau$$

Unpredictability. Each random output should be *unpredictable*: Given an agreed ledger, the adversary can predict the next random output before it is produced. If the adversary can predict future random outputs, then it may take advantage in randomness-based applications. The unpredictability definition follows the paradigm that without protocol transcripts from correct participants, no adversary can distinguish between a future random output of the DRB and a randomly sampled string of the same length [146], [214].

Definition 6.2.3 (Unpredictability, from [214]). A DRB protocol Π is unpredictable if for every κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds. Assuming all participants have agreed on a ledger of e consecutive random outputs $R_1, \ldots, R_e$. For any future random output $R_{e'}$ where $e' > e$ and any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, if $\mathcal{A}$ does not have the knowledge of protocol transcripts associated with $R_{e'}$ from correct participants, then

$$|\Pr[\mathcal{A}(R_{e'}) = 1] - \Pr[\mathcal{A}(r) = 1]| \leq \mathsf{negl}(\kappa)$$

, where r is a randomly sampled κ-bit string, and $\mathcal{A}(x) \to \{0, 1\}$ outputs 1 if $\mathcal{A}$ guesses x to be the random output in epoch e' and otherwise 0.

6.2.4 Performance metrics

DRBs concern two performance metrics, namely communication complexity and latency.

Communication complexity. Communication complexity is the total amount of communication required to complete a protocol [96]. In DRBs, the communication complexity is quantified as the amount of bits transferred among participants for generating a random output. A protocol may have different communication complexity in the best-case and worst-case executions.

Latency. Latency is the time required to complete a protocol. In the context of DRBs, the latency is quantified as the time participants take to generate a random output. Similarly, a protocol may have different latencies in the best-case and worst-case executions.

6.3 Delivery-fairness property

In this section, we formally define the delivery-fairness property. The delivery-fairness concerns two aspects of the advantage: *length-advantage* and *time-advantage*. Length-advantage concerns how many random outputs the adversary can learn earlier than correct participants. Time-advantage concerns how much time the adversary can learn a random output earlier than correct participants. We also prove the lower bound of the delivery-fairness, representing the optimal guarantee.

6.3.1 Defining delivery-fairness

We define delivery-fairness through two strawman definitions that are intuitive but incomplete. We begin with the *fairness* notion in multiparty computation (MPC) protocols that, if the adversary receives the output, then correct participants eventually receive the output [231]. We then generalise the fairness notion to the continuous time model, making it consistent with the DRB settings.

Attempt #1: Time advantage. We first consider relaxing the round-based fairness definition to the continuous time model by introducing a time parameter ψ. Namely, if a participant learns a random output at time t, then all other participants learn this random output no later than time $t + \psi$. However, this definition fails to capture that the adversary may learn more than one random outputs in advance than correct participants.

Attempt #2: Length and time advantage. We then consider capturing both length and time advantage. Let ω be the length parameter and ψ be the time advantage parameter. A DRB protocol satisfies (ω, ψ)-delivery-fairness if for any two participants p_i, p_j: 1) p_i's ledger is longer than p_j's ledger by no more than ω random outputs, and 2) p_j's ledger at time $t + \psi$ is no shorter than p_i's ledger at time t.

However, the definition does not specify whether the last ω random outputs of p_j at time $t + \psi$ should be identical to the last ω random outputs of p_i at time t or not. If not, then this contradicts to the consistency property.

Attempt #3: Length and time advantage with consistency. We then add the consistency guarantee to the definition in attempt #2, leading to our final definition. Specifically, delivery-fairness concerns the adversary's length advantage and time advantage, parameterised by ω and ψ, respectively. The ω-length-advantage states that the longest valid ledger pruning the last ω blocks is a prefix of the valid ledger in any participant's view at any time. The ψ-time-advantage states that the shortest valid ledger at time t should "catch up with" all participants' ledgers at time t after the time period of ψ. When ω and ψ are smaller, the DRB provides stronger delivery-fairness guarantee.

Definition 6.3.1 ((ω, ψ)-Delivery-Fairness)**.** A DRB protocol Π satisfies (ω, ψ)-delivery-fairness if for every κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds for any two participants p_i, p_j and any time t except for probability $\mathsf{negl}(\kappa)$:

- ω**-length-advantage**: $(\mathcal{T}_i^t)^{\lceil \omega} \preceq \mathcal{T}_j^t$
- ψ**-time-advantage**: $\mathcal{T}_i^t \preceq \mathcal{T}_j^{t+\psi}$

6.3.2 Lower bound of delivery-fairness

We prove that $(1, \Delta - \delta)$-delivery-fairness is the optimal delivery-fairness guarantee. Specifically, we prove the following theorem.

Theorem 6.3.1 (Delivery-fairness lower bound of DRB)**.** *There does not exist a DRB protocol that simultaneously satisfies the following in synchronous networks:*

- *consistency, liveness and unpredictability as in §6.2; and*
- (ω, ψ)*-delivery fairness with* $\omega < 1$ *or* $\psi < \Delta - \delta$

The intuition behind the proof is that, if the time difference of learning a random output between any two participants is smaller than $\Delta - \delta$, then the group of Byzantine participants can produce valid random outputs without communicating with correct participants, contradicting to unpredictability or consistency.

Proof. Assuming such a DRB protocol exists. Assuming at time t, all participants have agreed on a ledger of e consecutive random outputs $R_1, \ldots, R_e$, and start producing R_{e+1}. $\mathcal{A}$ sets the latency among correct participants to be Δ, the latency of messages from any corrupted participant to any correct participant to be Δ, and the latency of messages from any correct participant to any corrupted participant to be δ. By unpredictability, without messages from correct participants, corrupted participants cannot learn the value of R_{e+1}. Thus, the fastest possible way for corrupted participants to learn a random output is to get messages from correct participants, which is at least $t + \delta$. Given the latency set by $\mathcal{A}$, a correct participant receives messages only at $t + \Delta$.

By the assumption that the time-advantage between correct and corrupted participants is smaller than $\Delta - \delta$, correct participants learns the random output R_{e+1} before $t + \Delta$. Thus, a correct participant has to learn the random output R'_{e+1} that satisfies the verification predicate. If $R'_{e+1} = R_{e+1}$, then this means that a participant can solely produce random outputs without interacting with the other participants. Thus, f corrupted participants can also produce random outputs without interacting with the other correct participants, contradicting to the unpredictability property. If $R'_{e+1} \neq R_{e+1}$, then this means that f corrupted participants can produce valid random outputs conflicted with those from the other participants, contradicting to the consistency property. $\square$

6.4 Drand

In this section, we analyse the delivery-fairness of Drand [203], a DRB protocol adopted by Cloudflare [215]. Through the analysis, we identify an attack on the delivery-fairness, which we call *latency manipulation attack*. We show that two designs are necessary to resist against the latency manipulation attack, namely 1) lock-step execution and 2) broadcasting final random outputs. The Drand protocol specified in the documentation [232] does not satisfy delivery-fairness, while its actual implementation [229] has adopted these two designs, achieving the optimal $(1, \Delta - \delta)$-delivery fairness.

6.4.1 Primitive: BLS threshold signature

BLS threshold signature [233] is a threshold version of the BLS signature [234]. A (k, n)-BLS signature allows any k out of n participants to jointly sign a message. It requires a distributed key generation (DKG) protocol, where the set of n participants agree on security parameter κ, threshold $k \leq n$ a public key pk, and each participant p_i receives a secret key sk_i. The BLS threshold signature Π_{BLS} consists of the following functions.

- $\mathsf{Setup}(\kappa, k, n) \rightarrow (sk_1, \dots, sk_n, pk)$: A DKG protocol that takes security parameter κ, threshold k and number n of participants, outputs n secret key shares $sk_1, \dots, sk_n$ and a public key pk.

- $\mathsf{Sign}(sk_i, m) \to \sigma_i$: On input secret key sk_i and message m, outputs signature share σ_i.
- $\mathsf{Aggregate}(\vec{\boldsymbol{\sigma}}) \to \{\sigma, \bot\}$: On input k different signature shares $\vec{\boldsymbol{\sigma}}$, outputs a signature σ if every signature share $\sigma_i \in \vec{\boldsymbol{\sigma}}$ is correctly signed by sk_i otherwise $\bot$.
- $\mathsf{Verify}(pk, m, \sigma) \to \{0, 1\}$: On input public key pk, message m and signature σ, outputs 1 if σ is aggregated from k different correct signature shares otherwise 0.

Threshold BLS signature is *unique*: For any two subsets $(\vec{\boldsymbol{\sigma}}, \overrightarrow{\boldsymbol{\sigma}'})$ of k different signature shares in $\{\sigma_i\}_{i \in [n]}$ on a message m, $\Pi_{\mathsf{BLS}}.\mathsf{Aggregate}(\vec{\boldsymbol{\sigma}}) = \Pi_{\mathsf{BLS}}.\mathsf{Aggregate}(\overrightarrow{\boldsymbol{\sigma}'})$. Standard security properties of threshold signatures are less relevant to delivery-fairness analysis and thus are omitted.

6.4.2 Protocol specification

Drand requires a distributed key generation (DKG) and achieves the fault tolerance capacity of $n \geq 2f + 1$. It has two variants, namely the non-lock-step Π_{Drand} specified in the documentation [232] and the lock-step $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ in the actual implementation [229]. Compared to Π_{Drand}, $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ requires participants to wait for a time period during the phase of broadcasting signatures for each epoch e. In synchronous networks, the time period is at least Δ. In Drand's implementation $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ [229], the default time period (named `DefaultBeaconPeriod`) is 60 seconds. Figure 6.1 outlines the specification of Π_{Drand} and $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ from the perspective of participant p_i in epoch e.

1. **(Setup)** All participants jointly complete the one-time setup as follows.
   (a) All participants participate in $\Pi_{\mathsf{BLS}}.\mathsf{Setup}(\kappa, f+1, n)$, so that each participant p_i obtains the public key pk and a secret key sk_i.
   (b) All participants agree on the initial unique signature σ^0.
2. **(Broadcast signature)** Upon the signature σ^{e-1} in epoch $e-1$ from others or itself, participant p_i does the following.
   (a) p_i executes $\Pi_{\mathsf{BLS}}.\mathsf{Verify}(pk, H(e \| \sigma^{e-2}), \sigma^{e-1})$ to verify whether σ^{e-1} is valid.
   (b) p_i executes $\sigma_i^e \leftarrow \Pi_{\mathsf{BLS}}.\mathsf{Sign}(sk_i, H(e \| \sigma^{e-1}))$ to generate signature share σ_i^e.
   (c) p_i broadcasts σ_i^e.
   (d) p_i sets a timer Δ.
3. **(Generate random output)** Upon timer Δ expires and receiving at least $f+1$ signatures in epoch $e-1$, participant p_i does the following.
   (a) p_i executes $\sigma^e \leftarrow \Pi_{\mathsf{BLS}}.\mathsf{Aggregate}(\{\sigma_i^e\})$ to obtain the aggregated signature σ^e.
   (b) p_i broadcasts σ^e.
   (c) p_i calculates the random output $R_e \leftarrow H(\sigma^e)$.

Figure 6.1: Specification of Drand Π_{Drand}. Extra specification of its lock-step variant $\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ is labelled in blue.

6.4.3 Delivery-fairness analysis of Π_{Drand}: The latency manipulation attack

We identify a new attack *latency manipulation attack* that can increase the adversary's advantage of delivery-fairness in the non-lock-step Π_{Drand}. The latency manipulation attack only requires the adversary to manipulate the latency among participants (subjected to the network model), and does not require equivocating or withholding messages. Thus, the attack is unaccountable and does not affect other security properties or performance metrics.

The latency manipulation attack is presented in Figure 6.2 and depicted in Figure 6.3. The adversary $\mathcal{A}$ follows the protocol with $n-2f$ correct participants (e.g., $n-2f=1$ in Drand) while delaying all messages from and to the other f correct participants. Under the latency manipulation attack, In the beginning of the DRB protocol, the adversary $\mathcal{A}$ does the follows.

1. $\mathcal{A}$ chooses $n - 2f$ correct participants.
2. $\mathcal{A}$ sets the latency of messages among f corrupted participants and $n - 2f$ correct participants to be δ.
3. $\mathcal{A}$ sets the latency of messages from, to and among the other f correct participants to be Δ.

Figure 6.2: Latency manipulation attack on DRBs.

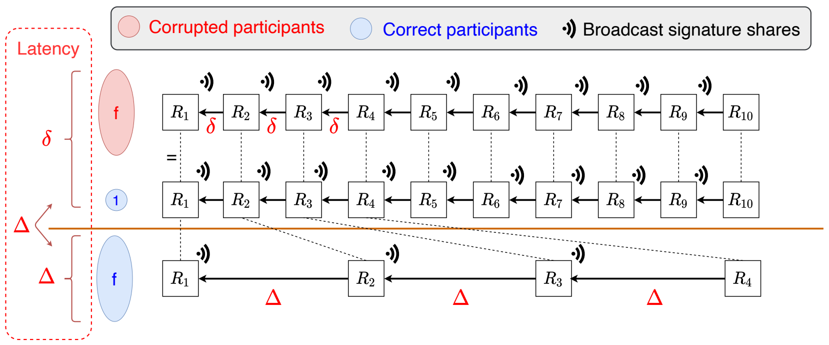


Figure 6.3: Example of the latency manipulation attack on the non-lock-step Drand Π_{Drand} with $\Delta = 3\delta$. The adversary $\mathcal{A}$ chooses $n - 2f = 1$ correct participant, sets the latency among f corrupted participants and the chosen correct participant (above the brown horizontal line) to be δ, and sets the latency from, to, and among other f correct participants (below the brown horizontal line) to be Δ. Consequently, participants above and below the brown horizontal line learn random outputs for every δ and Δ, respectively. The advantage accumulates linearly with the execution time: Participants above the brown horizontal line learn R_2, R_3, and R_4 earlier than participants below the brown horizontal line by 2δ, 4δ and 6δ, respectively. After a sufficiently long time, the delivery-fairness degree (ω, ψ) will both become ∞.

the adversary and one correct participant learn random outputs in every δ while the other f correct participants produce random outputs in every Δ. Consequently, the adversary's advantage, i.e., length-advantage degree ω and time-advantage degree ψ, accumulates linearly with the execution time After a sufficient long time period of execution, ω and ψ will become ∞, leading to impractical delivery-fairness guarantee.

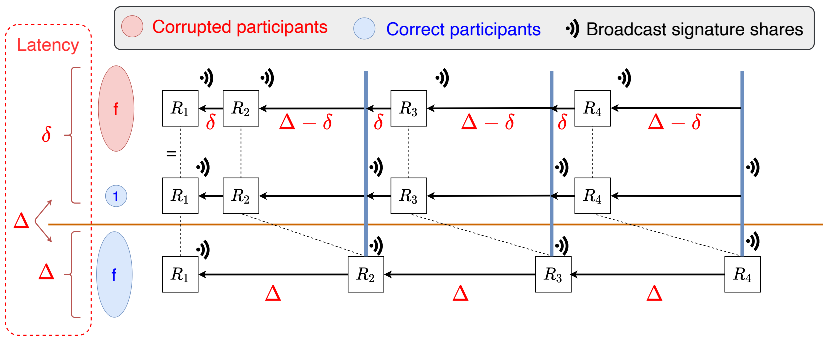


Figure 6.4: Example of the latency manipulation attack on the lock-step Drand $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ with $\Delta = 3\delta$. While corrupted participants will enter the next epoch immediately after learning a random output, correct participants will stay in every epoch for Δ, even learning the random output of this epoch in advance (i.e., δ since the beginning of the epoch). Consequently, $\mathcal{A}$ can only gain $(1, \Delta - \delta)$-delivery-fairness.

6.4.4 Delivery-fairness of $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$

We analyse the delivery-fairness guarantee of $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$, and show that $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ achieves optimal $(1, \Delta - \delta)$-delivery-fairness. The improvement compared to the non-lock-step Π_{Drand} is due to the lock-step design, where correct participants will wait for Δ before entering the next epoch and broadcasting signature shares, even learning the random output of this epoch in advance, as depicted in Figure 6.4.

Lemma 6.4.1 ($\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ epoch execution)**.** *In $\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$, at the end of every epoch e, i.e., $t = e \cdot \Delta$, for any two participants (p_i, p_j), $\mathcal{T}^t_i = \mathcal{T}^t_j$.*

Proof. The proof is by induction: Given the base case where all participants start executing the protocol, assuming an induction hypothesis holds at the end of epoch $e - 1$, we prove the induction step that the hypothesis holds at the end of epoch e. The proof implies that the induction hypothesis holds at the end of every epoch.

- **Bese case:** At time $t = 0$, for any two participants (p_i, p_j), $\mathcal{T}^0_i = \mathcal{T}^0_j$.
- **Induction hypothesis:** At time $t = (e - 1)\Delta$, for any two participants (p_i, p_j), $\mathcal{T}^t_i = \mathcal{T}^t_j$.

- **Proof goal:** At time $t = e \cdot \Delta$, for any two participants (p_i, p_j), $\mathcal{T}_i^t = \mathcal{T}_j^t$.

The induction step is as follows. After $(e-1) \cdot \Delta$, $n - f$ participants sign $(e \| \sigma^{e-1})$ and broadcast their signatures to each other. No later than $t = (e-1) \cdot \Delta + \Delta < e \cdot \Delta$, $n - f$ participants will receive $n - f$ signatures and can reconstruct σ^e, which leads to $\mathcal{T}_i^t = \mathcal{T}_j^t$ and thus closes the induction proof. □

Lemma 6.4.2 ($\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ length-advantage)**.** $\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ *achieves* 1*-length-advantage. That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $(\mathcal{T}_i^t)^{\lceil 1} \preceq \mathcal{T}_j^t$*.*

Proof. By Lemma 6.4.1, at the end of every epoch e, i.e., $t = e \cdot \Delta$, for any two participants (p_i, p_j), $\mathcal{T}_i^t = \mathcal{T}_j^t$. Therefore, for any $t \in ((e-1) \cdot \Delta, e \cdot \Delta]$, participants only differ in σ^e, leading to 1-length-advantage. □

Lemma 6.4.3 ($\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ time-advantage)**.** $\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ *achieves* $(\Delta - \delta)$*-time-advantage. That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $\mathcal{T}_i^t \preceq \mathcal{T}_j^{t+\Delta-\delta}$*.*

Proof. By Lemma 6.4.1, at the end of every epoch e, i.e., $t = e \cdot \Delta$, for any two participants (p_i, p_j), $\mathcal{T}_i^t = \mathcal{T}_j^t$. Starting from time t,the adversary will learn the next random output no earlier than $t + \delta$, while correct participants will learn the next random output no later than $t + \Delta$, leading to $(\Delta - \delta)$-time-advantage. □

Theorem 6.4.4 ($\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ delivery-fairness)**.** $\Pi_{\mathsf{Drand}}^{\mathsf{LS}}$ *achieves* (ω, ψ)*-delivery-fairness where* $\omega = 1$ *and* $\psi = \Delta - \delta$*.*

Proof. By Lemma 6.4.2-6.4.3, this theorem holds. □

6.4.5 Gained insights

Through the analysis, we obtain two insights on improving the delivery-fairness. First, the lock-step execution is necessary to bound the adversary's length-advantage to 1. Otherwise, without the lock-step execution, the adversary can always launch the latency manipulation attack and grow its ledger faster than correct participants. In addition, broadcasting the reconstructed random output (line 3b of Figure 6.1) is necessary to bound the adversary's time-advantage to Δ. Although correct participants will eventually receive enough shares to reconstruct the random output, the latency manipulation attack allows the adversary to produce random outputs faster than correct participants, leading to time-advantage of more than Δ if the random output is computed locally and is not broadcasted.

6.5 HydRand and GRandPiper

In this section, we analyse the delivery-fairness of HydRand [101] and GRandPiper [213], two DRB protocols based on the rotating leader paradigm and PVSS. We observe that a previously known unpredictability-focused attack, which we call *private beacon attack*, weakens the delivery-fairness of HydRand and GrandPiper. The attack is rooted in their design is that the entropy of a random output is solely provided by the epoch leader. To resist against the attack, the entropy should instead be provided by a group of at least $f + 1$ participants.

6.5.1 Primitives

HydRand and GRandPiper employ a number of cryptographic primitives, including leader election, Byzantine broadacst, accumulator and publicly verifiable secret sharing (PVSS). Similarly, security properties of these primitives are less relevant to delivery-fairness analysis and thus are omitted.

Leader election. Leader election protocol Π_{LE} allows participants to elect a leader for every epoch. Specifically, given the set of participants and the agreed ledger (recording historical random outputs and leaders) in epoch $e-1$ as input, Π_{LE} outputs a leader l_e for epoch e. Let $X(\Pi_{\mathsf{LE}}, c)$ be the event that c consecutive leaders are corrupted in Π_{LE}.

HydRand and RandPiper employ a round-robin leader election protocol $\Pi_{\mathsf{LE}}^{\mathsf{RR}}$, where a leader is elected from all participants excluding last f leaders and misbehaving leaders randomly with the last random output. Existing analysis [213] has proven that the probability $\Pr\left[X(\Pi_{\mathsf{LE}}^{\mathsf{RR}}, c)\right]$ that $X(\Pi_{\mathsf{LE}}^{\mathsf{RR}}, c)$ happens in $\Pi_{\mathsf{LE}}^{\mathsf{RR}}$ is $\Pr\left[X(\Pi_{\mathsf{LE}}^{\mathsf{RR}}, c)\right] = \frac{\binom{f}{c}}{(n-f)^c}$. The best, average, and worst values of c are 0, 2, and f, respectively.

Byzantine broadcast. In Byzantine broadcast Π_{BB}, a designated broadcaster in a set of n participants broadcasts a value m to other participants such that the following holds [235]–[237]:

- **Agreement:** If two correct participants commit values v and v' respectively, then $v = v'$.
- **Termination:** All correct participants eventually commit a value.
- **Validity:** If the designated broadcaster is correct, then all correct participants commit m.

By executing the leader election and allowing the leader to launch a Byzantine broadcast protocol for each epoch, one can yield a *state machine replication (SMR)* protocol. In SMR, participants commit client requests as a linearisable ledger (aka log) with two guarantees, namely *consistency* that every two correct participants commit the same value at the same ledger position and *liveness* that every client request is eventually committed by all correct participants [237].

Accumulator. Accumulator [123] allows to compress a set $\mathcal{D}$ of values into a short accumulation value z. For each value $d_i \in \mathcal{D}$, there is a short witness w_i proving that d_i is one of the values accumulated into z. An accumulator Π_{Acc} consists of the following algorithms:

- $\mathsf{Setup}(\kappa, n) \to k$: On input parameter κ and accumulation threshold n, outputs accumulation key k. Note that n is the upper bound on the total number of values that can be securely accumulated.

- $\mathsf{Eval}(k, \mathcal{D}) \to z$: On input key k and a set $\mathcal{D}$ of values, outputs accumulation value z.

- $\mathsf{CreateWit}(k, z, d_i, \mathcal{D}) \to \{w_i, \perp\}$: On input key k, accumulation value z, value d_i and the set $\mathcal{D}$ of values, outputs witness w_i if $d_i \in \mathcal{D}$ otherwise $\perp$.

- $\mathsf{Verify}(k, z, d_i, w_i) \to \{0, 1\}$: On input key k, accumulation value z for $\mathcal{D}$, value d_i and witness w_i, outputs 1 if $d_i \in \mathcal{D}$ otherwise 0.

PVSS. A (k, n)-Public Verifiable Secret Sharing (PVSS) [148], [238] allows one to distribute a secret s with n parties. Each party p_i receives a share s_i. Anyone can encrypt s_i to an encrypted share c_i by using p_i's public key pk_i. Party p_i can decrypt the encrypted share c_i back to s_i by using its secret key sk_i. A set of k different shares can be used for reconstructing the secret. A PVSS scheme Π_{PVSS} consists of the following algorithms:

- $\mathsf{Gen}(\kappa) \to (sk, pk)$: On input security parameter κ, outputs key pair (sk, pk).

- $\mathsf{Encrypt}(pk_i, s_i) \to c_i$: On input public key pk_i and share s_i, outputs encrypted share c_i.

- $\mathsf{Decrypt}(sk_i, c_i) \to s_i$: On input secret key sk_i and encrypted share c_i, outputs share s_i. It holds that $s_i = \mathsf{Decrypt}(sk_i, \mathsf{Encrypt}(pk_i, s_i))$.

- $\mathsf{Share}(\{pk_i\}_{i\in[n]}, k, s) \rightarrow (\vec{\boldsymbol{s}}, \vec{\boldsymbol{c}}, \vec{\boldsymbol{\pi}})$: On public keys $\{pk_i\}_{i\in[n]}$, threshold k, and secret s, outputs shares $\vec{\boldsymbol{s}} = \{s_i\}_{i\in n}$, encrypted shares $\vec{\boldsymbol{c}} = \{c_i\}_{i\in n} = \{\mathsf{Encrypt}(pk_i, s_i)\}_{i\in n}$, and proofs $\vec{\boldsymbol{\pi}} = \{\pi_i\}_{i\in n}$.

- $\mathsf{Verify}(c_i, \pi_i) \rightarrow \{0, 1\}$: On input encrypted share c_i and proof π_i, outputs 1 if c_i is encrypted from a certain valid share otherwise 0. Note that the proof π_i does not reveal which share is associated with c_i.

- $\mathsf{Recon}(\vec{\boldsymbol{s}}) \rightarrow s$: On input a set $\vec{\boldsymbol{s}}$ of k valid shares, outputs secret s.

6.5.2 HydRand protocol specification

HydRand does not require distributed key generation and achieve the fault tolerance capacity of $n \geq 3f + 1$. The original HydRand protocol Π_{HydRand} in the paper [101] and implementation [239] is non-lock-step. We also study its lock-step variant $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ that resists against the latency manipulation attacks and achieves better delivery-fairness. Figure 6.5 outlines the specification of Π_{HydRand} and $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ from the perspective of participant p_i in epoch e.

6.5.3 HydRand delivery-fairness analysis

The latency manipulation attack in Figure 6.2 also applies to the non-lock-step Π_{HydRand}, similar to the non-lock-step Π_{Drand}. Specifically, $\mathcal{A}$ always sets the latency among its corrupted participants and $n - 2f = f + 1$ correct participants as δ, while the latency from, to and among the other f correct participants as Δ. To exclude the impact of the latency manipulation attack, we focus on analysing $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$.

Bhat et al. [213] observes an attack on the unpredictability of HydRand and GRandPiper, and does not name it. This attack, which we call *private beacon attack*, can also weaken the delivery-fairness of HydRand (including both Π_{HydRand} and $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$) and GRandPiper. In this attack, the adversary grows its own ledger to learn random outputs earlier than correct participants. As HydRand allows the epoch leader to solely sample the entropy, the epoch leader can learn the random output instantly without communicating with others. With c consecutive Byzantine leaders, the adversary can learn c future random outputs. Same as the latency manipulation attack, the private beacon attack does not require equivocating or withholding messages, and thus remains unaccountable. The private beacon attack is presented in Figure 6.6 and depicted in Figure 6.7.

We analyse the impact of private beacon attacks on the delivery-fairness for $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$. Recall that $X(\Pi_{\mathsf{LE}}, c)$ denotes the event that "c consecutive leaders are corrupted in Π_{LE}".

Lemma 6.5.1. $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ *achieves* $(c+1)$*-length-advantage with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. *That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $(\mathcal{T}_i^t)^{\lceil c+1} \preceq \mathcal{T}_j^t$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. Assuming $\mathcal{A}$ does not corrupt the leader l_e in the current epoch e. In epoch e, $\mathcal{A}$ can launch the latency manipulation attack, so that it learns R_e earlier than correct participants. When $X(\Pi_{\mathsf{LE}}, c)$ happens with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$, the leader corrupts further c consecutive leaders $l_{e+1}, \dots, l_{e+c}$ and learns further c random outputs $R_{e+1}, \dots, R_{e+c}$. Therefore, $\mathcal{A}$ achieves $(c+1)$-length-advantage with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. □

Lemma 6.5.2. $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ *achieves* $((3c+1)\Delta - 3\delta)$*-time-advantage with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. *That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $\mathcal{T}_i^t \preceq \mathcal{T}_j^{t+(3c+1)\Delta-3\delta}$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. Assuming when epoch e starts at time t, $\mathcal{A}$ does not corrupt the leader l_e. $\mathcal{A}$ directs corrupted participants to follow the protocol while launching the latency manipulation attack, so that after three consecutive message transfers, $\mathcal{A}$ learns R_e at time $t + 3\delta$ and the other correct participants learn R_e at time $t + 3\Delta$. When $X(\Pi_{\mathsf{LE}}, c)$ happens with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$, the leader corrupts further c consecutive leaders $l_{e+1}, \ldots, l_{e+c}$ and learns further c random outputs $R_{e+1}, \ldots, R_{e+c}$. The correct participants learn R_{e+c} only at $t + 3\Delta + 3c \cdot \Delta$. Therefore, the time advantage is $(t+3\Delta+3c\cdot\Delta)-(t+3\delta) = (3c+1)\Delta - 3\delta$ with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. □

Theorem 6.5.3 ($\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ delivery-fairness)**.** $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ *achieves* (ω, ψ)*-delivery-fairness where* $\omega = c + 1$ *and* $\psi = (3c + 1)\Delta - 3\delta$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. By Lemma 6.5.1-6.5.2, this theorem holds. □

6.5.4 GRandPiper protocol specification

GRandPiper [213] follows the HydRand's approach with three major modifications. First, GRandPiper enforces participants to recover the secret value committed by the leader, without allowing the leader to reveal it by itself. Second, GRandPiper replaces the Acknowledge and Vote-confirm phase in HydRand with an explicit Byzantine broadcast protocol Π_{BB}. Note that the Byzantine broadcast and the round-robin leader election constitute a SMR protocol, as described in the RandPiper paper. Third, GRandPiper formalises the Hydrand's idea of separating the process of committing and revealing secret values as a queue-based mechanism, where each participant buffers previously committed secret values and pops one value to reconstruct for each epoch.

GRandPiper does not require distributed key generation and achieve the fault tolerance capacity of $n \geq 2f + 1$. The GRandPiper protocol $\Pi_{\mathsf{GRandPiper}}$ is constructed from Π_{LE}, Π_{BB}, and Π_{PVSS}. Π_{LE} is instantiated with $\Pi^{\mathsf{RR}}_{\mathsf{LE}}$, Π_{BB} is instantiated with a specialised protocol with $O(n^2)$ communictaion complexity and latency $t_{\mathsf{BB}} = 11\Delta$. $\Pi_{\mathsf{GRandPiper}}$ achieves $O(n^2)$ communication complexity and 12Δ latency. Figure 6.8 outlines the specification of $\Pi_{\mathsf{GRandPiper}}$ from the perspective of participant p_i in epoch e.

6.5.5 GRandPiper delivery-fairness analysis

Lemma 6.5.4 ($\Pi_{\mathsf{GRandPiper}}$ **length-advantage).** $\Pi_{\mathsf{GRandPiper}}$ *achieves* $(c+1)$*-length-advantage with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$*. That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $(\mathcal{T}_i^t)^{\lceil c+1} \preceq \mathcal{T}_j^t$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. The proof is identical to that of Lemma 6.5.1. □

Lemma 6.5.5 ($\Pi_{\mathsf{GRandPiper}}$ **time-advantage).** $\Pi_{\mathsf{GRandPiper}}$ *achieves* $((11c+1)\Delta-\delta)$*-time-advantage with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$*. That is, for every* κ*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds except for probability* $\mathsf{negl}(\kappa)$*. For any two participants* (p_i, p_j) *and any time* t*,* $\mathcal{T}_i^t \preceq \mathcal{T}_j^{(11c+1)\Delta-\delta}$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. By Lemma 6.5.4, for every epoch e, i.e., at time $t = e \cdot t_{\mathsf{BB}} + \delta$, the adversary learns the $(e+c)$-th random output with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. Meanwhile, correct participants will learn the $(e+c)$-th random output after c epochs plus a Δ in the DRB routine, i.e., at time $t = (e+c) \cdot t_{\mathsf{BB}} + \Delta$. This leads to $(c \cdot t_{\mathsf{BB}} + \Delta - \delta)$-time-advantage with probability $\Pr[X(\Pi_{\mathsf{LE}}, c)]$. As $t_{\mathsf{BB}} = 11\Delta$, $c \cdot t_{\mathsf{BB}} + \Delta - \delta = (11c+1)\Delta - \delta$. □

Theorem 6.5.6 ($\Pi_{\mathsf{GRandPiper}}$ **delivery-fairness).** $\Pi_{\mathsf{GRandPiper}}$ *achieves* (ω, ψ)*-delivery-fairness where* $\omega = c+1$ *and* $\psi = (11c+1)\Delta - \delta$ *with probability* $\Pr[X(\Pi_{\mathsf{LE}}, c)]$.

Proof. By Lemma 6.5.4-6.5.5, this theorem holds. □

6.5.6 Gained insights

In HydRand and GRandPiper, the entropy of a random output is provided by a sole leader. In this case, the adversary can always launch the private beacon attack as long as the leader is Byzantine. To mitigate the private beacon attack, the protocol should prevent the adversary from controlling the entropy for a random output. To this end, the entropy should instead be provided by a group of at least $f + 1$ participants rather than a single participant.

6.6 SPURT

In this section, we analyse the delivery-fairness of SPURT [214]. The analysis shows that by making SPURT lock-step, SPURT achieves the optimal $(1, \Delta - \delta)$-delivery-fairness.

6.6.1 Protocol specification

SPURT is constructed from a Byzantine broadcast protocol Π_{BB} and a specialised PVSS protocol $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}$. Π_{BB} is a variant of HotStuff [153] with best-case latency of 4δ and worst-case latency $t_{\mathsf{BB}} = 4\Delta$. $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}$ differs from the traditional Π_{PVSS} in three aspects. First, $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}.\mathsf{Recon}(\cdot)$ outputs $e(h_0^s, h_1)$ rather than secret s itself, where $e(\cdot)$ is a bilinear pairing function and (h_0, h_1) are public parameters. Second, an encrypted share in $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}$ consists of two elements (v_i, c_i). Last, proof π_i in $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}.\mathsf{Share}(\cdot)/\mathsf{Verify}(\cdot)$ is omitted in certain scenarios for better performance. To keep notations consistent, we denote (v_i, c_i) in $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}$ as a single element c_i, and explicitly include π_i in $\Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}.\mathsf{Verify}(\cdot)$.

SPURT [214] does not require distributed key generation and achieve the fault tolerance capacity of $n \geq 3f + 1$. The original SPURT protocol Π_{SPURT} in the paper [214] and implementation [240] is non-lock-step. We also study its lock-step variant $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ that resists against the latency manipulation attacks and achieves optimal delivery-fairness. Figure 6.9 outlines the specification of Π_{SPURT} and $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ from the perspective of participant p_i in epoch e.

6.6.2 Delivery-fairness analysis

Similar to Drand and HydRand, the non-lock-step Π_{SPURT} does not resist against the latency manipulation attack. SPURT resists against the private beacon attack, as the entropy of a random output is jointly provided by $f+1$ participants. We analyse the delivery-fairness guarantee of $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$, and show that $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ achieves the optimal $(1, \Delta - \delta)$-delivery-fairness.

Lemma 6.6.1 ($\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ epoch execution)**.** *In $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$, at the end of every epoch e, i.e., $t = e \cdot (3\Delta + t_{\mathsf{BB}})$, for any two participants (p_i, p_j), $\mathcal{T}_i^t = \mathcal{T}_j^t$.*

Proof. The proof is identical to that of Lemma 6.4.1. □

Lemma 6.6.2 ($\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ length-advantage)**.** *$\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ achieves 1-length-advantage. That is, for any κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds except for probability $\mathsf{negl}(\kappa)$. For any two participants (p_i, p_j) and any time t, $(\mathcal{T}_i^t)^{\lceil 1} \preceq \mathcal{T}_j^t$.*

Proof. The proof is identical to that of Lemma 6.5.4, except that the adversary does not learn c extra random outputs via the private beacon attack compared to correct participants. □

Lemma 6.6.3 ($\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ time-advantage)**.** *$\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ achieves $(\Delta - \delta)$-time-advantage. That is, for every κ, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds except for probability $\mathsf{negl}(\kappa)$. For any two participants (p_i, p_j) and any time t, $\mathcal{T}_i^t \preceq \mathcal{T}_j^{\Delta-\delta}$.*

Proof. The proof is identical to that of Lemma 6.5.5, except that the adversary does not learn c extra random outputs via the private beacon attack compared to correct participants. □

Theorem 6.6.4 ($\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ delivery-fairness)**.** *$\Pi^{\mathsf{LS}}_{\mathsf{Drand}}$ achieves (ω, ψ)-delivery-fairness where $\omega = 1$ and $\psi = \Delta - \delta$.*

Proof. By Lemma 6.6.2-6.6.3, this theorem holds. □

6.7 Related work

A systematic and formal study on this advantage is still missing. Existing DRB models only focus on certain attacks leading to certain aspects in this advantage [144], [213]. Similar security properties in other primitives cannot be adapted to capture this advantage directly, as they either concern eventual delivery without quantifying the advantage [227], or concern the advantage among correct participants excluding Byzantine participants [52], [163], [181], [228].

Similar properties in DRBs. Previous research either informally studies such advantage, or formally studies certain attacks leading to this advantage. Ouroboros Praos [144] states that a DRB is leaky if the leader can learn the random output in the next epoch, and embeds the property in the Universal Composability model. RandPiper [213] combines the "leaky" notion into the unpredictability property, yielding the d-unpredictability property, where the adversary can learn at most d future random outputs in advance. However, the d-unpredictability only captures *length-advantage*, i.e., how many random outputs the adversary can learn earlier than correct participants, but not *time-advantage*, i.e., how much time the adversary can learn a given random output earlier than correct participants. In addition, RandPiper only studies d-unpredictability under the private beacon attack where the adversary solely samples random outputs, neglecting other possible attacks on delivery-fairness. SPURT [214] defines the "nearly simultaneous beacon output" as a part of the unpredictability property, meaning that all correct participants learn a random output within a constant time after the adversary can learn it. The "nearly simultaneous" notion only captures the time-advantage but not the length-advantage. In addition, it quantifies the time-advantage asymptotically rather than concretely, making it less practical in the real-world networks.

Guaranteed output delivery and fairness in Multiparty Computation. Guaranteed output delivery (GOD) and fairness are properties of multi-party computation (MPC) protocols, where participants jointly compute a function of their inputs securely under a subset of corrupted participants. GOD specifies that corrupted participants cannot prevent the correct participants from receiving the function's output. Fairness specifies that corrupted participants should receive the function's output if and only if correct participants receive it. GOD and fairness are equivalent when broadcast channels are accessible [227], which is our setting. However, these two properties are usually analysed under discrete time models which only concern eventual delivery, and thus do not allow quantitative analysis.

Consistent length in Blockchain. Blockchain protocols allow participants to jointly maintain a blockchain. The consistent length property [52], [163], [181], [228] of blockchain protocols specify that if a correct participant's blockchain is of length ℓ at time t, then any correct participant's blockchain at time $t + \psi$ is of length at least ℓ. Blockchains trivially satisfy the property in synchronous networks, as a correct participant will send its chain to other correct participants within the synchronous latency Δ.

Adapting the consistent length property from blockchain protocols to DRBs suffers from two limitations. First, it only concerns the difference of blockchains between correct participants, excluding corrupted participants. In particular, the adversary may grow its blockchain faster than correct participants, while withholding its blockchain. Second, it only concerns the time advantage, i.e., how much time the adversary learns blocks earlier than correct participants, but does not concern how many blocks the adversary can learn earlier than correct participants.

1. **(Propose)** Upon a new random output R_{e-1}, participants execute as follows.
   (a) Participants execute Π_{LE} to determine the leader l_e.
   (b) Leader l_e chooses a new secret s, obtains shares, encrypted shares and share proofs $(\vec{\boldsymbol{s}}, \vec{\boldsymbol{c}}, \vec{\boldsymbol{\pi}}) \leftarrow \Pi_{\mathsf{PVSS}}.\mathsf{Share}(\{pk_i\}_{i\in[n]}, f+1, s)$, obtains the accumulation value z of $\vec{\boldsymbol{c}}$ via $z \leftarrow \Pi_{\mathsf{Acc}}.\mathsf{Eval}(k, \vec{\boldsymbol{c}})$, and obtains all witnesses $\vec{\boldsymbol{w}}$ via $w_i \leftarrow \Pi_{\mathsf{Acc}}.\mathsf{CreateWit}(k, z, c_i, \vec{\boldsymbol{c}})$ for every $i \in [n]$.
   (c) Leader l_e constructs a block B_e and broadcasts B_e. B_e includes 1) information of epoch e: $(e, \vec{\boldsymbol{c}}, \vec{\boldsymbol{\pi}}, \vec{\boldsymbol{w}}, z)$, 2) information of epoch e^-: $(e^-, s^-, H(B_{e^-}), CC(B_e^-))$, and 3) information of in-between epochs $k \in (e^-, e)$: $\{(R_k, RC(k))\}_{k\in(e^-,e)}$, where e^- is the last epoch where the leader honestly reveals its secret, $CC(B_{e^-})$ ia a collection of $\geq f+1$ signatures on B_{e^-}, R_k is the recovered secret and $RC(k)$ ia a collection of $\geq f+1$ signatures on B_k.
   (d) All participants set a timer Δ.
2. **(Acknowledge)** Upon receiving a valid block B_e {before/and} Δ expires, participant p_i executes as follows.
   (a) p_i broadcasts an ACKNOWLEDGE message containing $H(B_e)$ and $(e, R_e, s^-, B_{e^-}, H(B_{e^-}), \{R_k\}_{k\in(e^-,e)}, z)$ signed by leader l_e, and set a new timer Δ.
   (b) Otherwise, if Δ expires and no valid block B_e is received, participant p_i moves to the vote-recover phase and sets a new timer Δ.
3. **(Vote-confirm)** Upon receiving $\geq 2f+1$ ACKNOWLEDGE messages on a valid block B_e {before/and} Δ expires, participant p_i executes as follows.
   (a) p_i broadcasts a CONFIRM message containing $H(B_e)$, and sets a new timer Δ.
   (b) Upon receiving $\geq f+1$ CONFIRM messages {before/and} Δ expires (which is guaranteed), p_i commits block B_e, calculates random output $R_e \leftarrow H(R_{e-1} \| s^-)$, and starts a new epoch.
   (c) Otherwise, if Δ expires and $< 2f+1$ ACKNOWLEDGE messages on a valid block B_e are received, p_i moves to the vote-recover phase and sets a new timer Δ.
4. **(Vote-recover)** If there exists a phase where the condition does not meet after the timer expires, participant p_i moves to the vote-recover phase to recover the secret s^- committed in B_e^- jointly with others. Specifically,
   (a) p_i obtains the decrypted share s_i^- via $\Pi_{\mathsf{PVSS}}.\mathsf{Decrypt}(sk_i, c_i^-)$ and broadcasts RECOVER message $(s_i^-, c_i^-, \pi_i^-, w_i^-, R_{e-1})$.
   (b) Upon receiving $\geq f+1$ RECOVER messages {before/and} Δ expires (which is guaranteed), p_i recovers the secret $s^- \leftarrow \Pi_{\mathsf{PVSS}}.\mathsf{Recon}(\overrightarrow{\boldsymbol{s}^-})$ (where $\overrightarrow{\boldsymbol{s}^-}$ is the $\geq f+1$ shares in RECOVER messages), and calculates random output $R_e \leftarrow H(R_{e-1} \| s^-)$.

Figure 6.5: Specification of HydRand Π_{HydRand}. Extra specification of its lock-step variant $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$ is labelled in blue.

While following the DRB protocol, the adversary $\mathcal{A}$ does the follows.

1. Upon a new random output R_e, $\mathcal{A}$ calculates the next leader l_{e+1} based on Π_{LE}.
2. If the next leader l_{e+1} is a Byzantine participant, $\mathcal{A}$ follows the protocol to sample the random output R_{e+1} locally and repeats step 1.

Figure 6.6: Private beacon attack on DRBs.

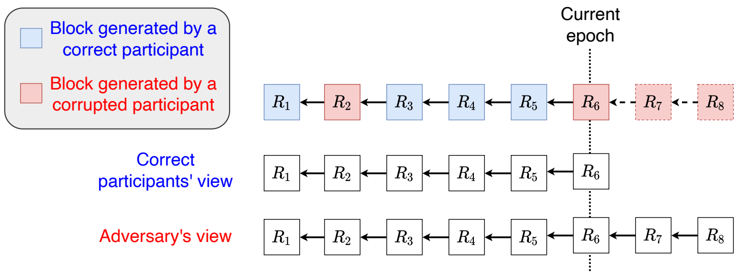

Block generated by a correct participant
Block generated by a corrupted participant
Current epoch
Correct participants' view
Adversary's view
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8

Figure 6.7: Example of a private beacon attack on the lock-step HydRand $\Pi^{\mathsf{LS}}_{\mathsf{HydRand}}$. Assuming at the current epoch $e = 6$, the leader l_6 is a corrupted participant. Leader l_6 reveals its committed secret and calculates the current random output R_6, which determines the next epoch's leader l_7, and so on. When l_6, l_7 and l_8 are all corrupted participants, the adversary $\mathcal{A}$ can learn R_7 and R_8 when epoch $e = 6$, weakening the delivery-fairness.

1. **(SMR routine)** Upon timer$_{e-1} = t_{\mathsf{BB}}$ ends, all participants execute as follows.
   (a) **Leader election.** Participants calculate the leader based on the round-robin approach same as HydRand. All participants set a new timer$_e = t_{\mathsf{BB}}$.
   (b) **Block proposal.** If elected as leader, leader l_e chooses a random value s_e, executes $(\vec{s}, \vec{c}, \vec{\pi}) \leftarrow \Pi_{\mathsf{PVSS}}.\mathsf{Share}(\{pk_i\}_{i\in[n]}, f+1, s)$, creates block $B_e = (\vec{c}, \vec{\pi})$, and triggers the Byzantine broadcast protocol Π_{BA} over B_e. Each Π_{BA} instance terminates in at most f epochs.
   (c) **Block agreement.** Upon agreeing on block B_e^- sent by leader l_{e^-} for epoch e^-, participant p_i pushes B_e^- into queue $Q(l_{e^-})$.
   (d) **Blame.** Upon timer$_e$ ends, if no block is proposed in epoch $e-t$, then remove l_{e-t} from future proposals.
2. **(DRB routine)** Upon timer$_{e-1}$ ends, participant p_i executes as follows.
   (a) p_i pops the committed block $B_{e^-} = (\overrightarrow{c^-}, \overrightarrow{\pi^-})$ from queue $Q(l_{e^-})$.
   (b) p_i decrypts its share $s_i^- \leftarrow \Pi_{\mathsf{PVSS}}.\mathsf{Decrypt}(sk_i, c_i^-)$ and broadcasts s_i^-.
   (c) Upon $f+1$ valid shares in $\overrightarrow{s^-}$, Participant p_i reconstructs $s_{e^-} \leftarrow \Pi_{\mathsf{PVSS}}.\mathsf{Recon}(\overrightarrow{s^-})$, and calculates the random output $R_{e^-} \leftarrow H(s_{e^-}, R_{e^- -1}, \ldots, R_{e^- -t})$.

Figure 6.8: Specification of GRandPiper $\Pi_{\mathsf{GRandPiper}}$.

1. **(Commitment)** Upon reconstructing R_{e-1}, every participant p_i executes as follows.
   (a) p_i executes $(\vec{\boldsymbol{s_i}}, \vec{\boldsymbol{c_i}}, \vec{\boldsymbol{\pi_i}}) \leftarrow \Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}.\mathsf{Share}(\{pk_j\}_{j\in[n]}, f, s)$.
   (b) p_i sends tuple $(\vec{\boldsymbol{c_i}}, \vec{\boldsymbol{\pi_i}})$ to leader l_e.
   (c) Set a timer Δ.
2. **(Aggregation)** Upon receiving $f+1$ tuples and Δ expires, leader l_e executes as follows.
   (a) l_e aggregates these tuples as $(\hat{\boldsymbol{c}}, \hat{\boldsymbol{\pi}}) = \{(\hat{c}_i, \hat{\pi}_i)\}_{i\in[n]} = \{(\Pi\bar{\boldsymbol{c}}_{\boldsymbol{i}}, \Pi\bar{\boldsymbol{\pi}}_{\boldsymbol{i}})\}_{i\in[n]}$, where $(\bar{\boldsymbol{c}}_{\boldsymbol{i}}, \bar{\boldsymbol{\pi}}_{\boldsymbol{i}})$ is the i-th column of $f+1$ encrypted shares and proofs, respectively.
   (b) l_e computes the digest $h \leftarrow H(I \| \hat{\boldsymbol{c}})$ where I is the set of $f+1$ indices.
3. **(Agreement)** Leader l_e executes as follows.
   (a) For each participant p_i, l_e sends $(e, h, I, \hat{\boldsymbol{c}}, \bar{\boldsymbol{c}}_{\boldsymbol{i}}, \bar{\boldsymbol{\pi}}_{\boldsymbol{i}})$ to p_i.
   (b) l_e triggers Π_{BB} over h with all participants.
   (c) Set a timer t_{BB}.
4. **(Reconstruction)** Upon deciding on h and t_{BB} expires, participant p_i executes as follows.
   (a) p_i decrypts the aggregated share $\hat{s}_i \leftarrow \Pi^{\mathsf{uniform}}_{\mathsf{PVSS}}.\mathsf{Decrypt}(sk_i, \hat{c}_i)$, and broadcasts $\hat{s}_i$. Set a timer Δ.
   (b) Upon receiving $f+1$ such decrypted shares $\hat{\boldsymbol{s}}$ and Δ expires, p_i aggregates them to $s = \sum \hat{\boldsymbol{s}}$, calculates the beacon output $R_e \leftarrow e(h_0^s, h_1)$ via pairing, and broadcasts R_e. Set a timer Δ.
   (c) Upon receiving $f+1$ R_e messages and Δ expires, p_i decides on R_e.

Figure 6.9: Specification of SPURT Π_{SPURT}. Extra specification of its lock-step variant $\Pi^{\mathsf{LS}}_{\mathsf{SPURT}}$ is labelled in blue.

Chapter 7

On the honest majority assumption of permissionless blockchains

7.1 Introduction

Proof-of-work (PoW) based consensus – first introduced by Bitcoin [1] allows distributed participants (aka. nodes) to agree on the same set of transactions. In Bitcoin, all transactions are organised as a *blockchain*, i.e., chain of blocks. Anyone can create a block of transactions, and append it into the Bitcoin blockchain as a unique successor of the last block. To create a block, one needs to solve a computationally hard Proof-of-Work (PoW) puzzle. In PoW, the puzzle solver (aka. miner) needs to find a nonce to make the hash value of the block smaller than a target value.

The blockchain may have *forks*: Miners may create different valid blocks following the same block. In Bitcoin, miners always choose the longest fork of its blockchain in order to agree on a single fork. However, a fork that is currently longer may be reverted by another fork, and all transactions in the currently longer fork will be deemed invalid. This gives the attacker an opportunity to spend a coin in a fork, then creates another longer fork to erase this transaction. This is called *double-spending attack*. To launch a double-spending attack in PoW-based consensus, an attacker should have enough mining power to create a fork growing faster than the current one. This requires the attacker to control a majority of mining power in the network. Double-spending attacks using the majority of mining power is known as *51% attacks*.

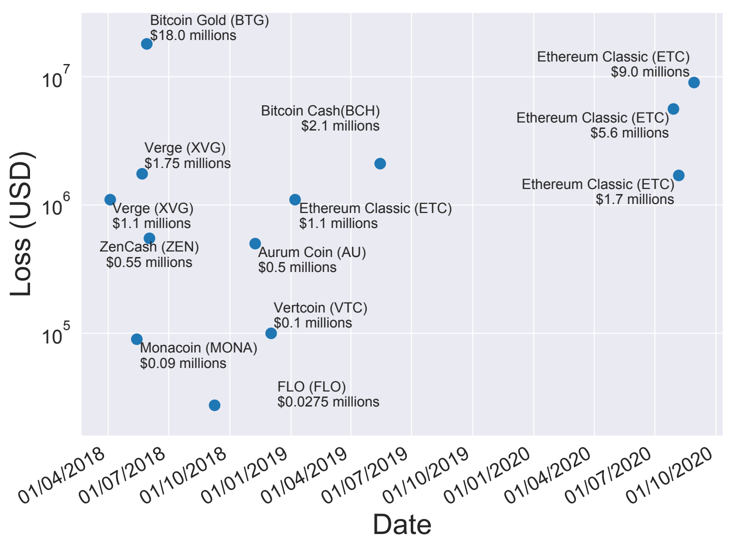


Figure 7.1: 51% Attacks in 2018-2020 [241]. We omit the 51% attack on Litecoin Cash on June 4, 2018 as the loss is unknown.

Honest majority. To avoid 51% attacks, PoW-based consensus should assume the *honest majority*: The majority of mining power in the system follows the protocol. Otherwise, the adversary with the majority of mining power can launch 51% attacks. Such security guarantee depends on the total mining power in the system: With more mining power in the system, controlling the majority of mining power will be more difficult.

Fact and Fiction. Ideally, there is only one blockchain in the world, and all miners will participate in this blockchain. This makes controlling 51% mining power extremely difficult. However, there exists numerous PoW-based blockchains [2]. As the total available mining power is shared amoung different blockchains, no blockchain enjoys the ideal security guarantee, and blockchains with more mining power is more secure than those with less mining power.

The existence of multiple blockchains gives the opportunity to 51% attacks and makes the *honest majority* breakable. As shown in Figure 7.1, there have been several 51% attacks, causing the loss of more than $41 million. Most notably, within a month from 29/07/2020 to 29/08/2020, there were three huge 51% attacks on Ethereum Classic (ETC) [242]–[244], where the largest one reverted more than 8,000 blocks and caused the loss of $9.0 million.

Incentive and rationality. For better security guarantee, a PoW-based blockchain should attract miners to contribute mining power. To attract miners, PoW-based blockchains usually employ an incentive mechanism, where a miner creating a block will be rewarded for its contribution. Such incentive mechanism makes PoW-based blockchains to assume miners are *rational* [245], i.e., making decisions for profit. The frequent huge 51% attacks on ETC indicate that, miners' rational choice may not only be mining honestly, but can also be launching 51% attacks. This leaves us a question that, does the incentive mechanism really encourages miners to mine honestly and secures the blockchain?

7.1.1 Contributions

While prior research [246]–[257] analyse PoW-based blockchains as a stand-alone system, we analyse PoW-based blockchains in the precense of externally available mining resources. We formally analysing two variants of 51% attacks using externally available mining power, and show that 51% attacks are feasible and more profitable than honestly mining for most blockchains. Our analysis leads to two results: 1) the honest majority assumption does not hold for these blockchains, and 2) instead of encouraging miners to mine honestly, the incentive mechanism encourages miners to launch 51% attacks and break "honest majority". Specifically, we make the following contributions.

Formalisation of two 51% attacks (§7.2). We consider two variants of 51% attacks that make use of externally available mining power. One is *mining power migration attack*, where the adversary migrates mining power from a stronger blockchain to attack a weaker blockchain. The other is the previously known *cloud mining attack* [14], where the adversary rents mining power from cloud mining services (e.g., Nicehash[13]) to attack a blockchain.

Whether these two attacks are feasible or profitable are unknown. Straightforward estimations are coarse-grained so may lead to biased estimations and consequently wrong conclusions. To identify PoW-based consensus' (overlooked) weaknesses and provide insights and directions towards securing them, we formalise the two 51% attacks using 51-MDP – a MDP-based model extended from Gervais et al. [253]. 51-MDP takes parameters of blockchains and the adversary as input, and outputs the cost and reward of launching a 51% attack. Of independent interest, 51-MDP can be leveraged to formally study all attacks on PoW-based blockchains while considering external environment.

Evaluaion of 51% attacks in existing blockchains (§7.3-7.4). We apply 51-MDP to evaluate two attacks on existing PoW-based blockchains. The results show that for most PoW-based blockchains, launching both 51% attacks is feasible and more profitable than hoenstly mining. For example, a miner with 12.5% mining power in Bitcoin can profit 6% ($1,894,650) more than honestly mining Bitcoin by double-spending a transaction of 300,000 BCH ($37,893,000) on BitcoinCash. The required mining power is not difficult to obtain – at the time of writing, F2Pool controls 17.7% mining power in Bitcoin [258].

Detailed case study of an 51% attack (§7.5-7.6). We apply 51-MDP to study the 51% attack on ETC happened at 07/01/2019. On 07/01/2019, an anonymous attacker launched a series of 51% attacks and double-spent more than $1.1 million on a cryptocurrency exchange Gate.io[259]. The attack is suspected to be a *cloud mining attack* using mining power from Nicehash [260]. We first analyse the pattern of double-spent transactions, and reveal the attacker's strategy for maximising and stabilising revenue. We then apply 51-MDP to reverse-engineer the attacker's revenue. The results show that, the attacker is expected to earn $84773.40, which is close to $100,000 – the attacker returned to Gate.io later[261]. This indicates the attacker was likely to launch 51% attacks in the fine-grained way described in this chapter.

Countermeasures (§7.7). We discuss potential countermeasures of these two 51% attacks derived from our observed insights. For quick remedies, we suggest ways of detecting such 51% attacks, and reacting upon detected 51% attacks. Among these approaches, we use 51-MDP to show that increasing the number of confirmation blocks is the most practical and effective countermeasure, which is also evidenced by the evaluation result of XMR (Figure 7.5) in §7.4. The recent recommended update aligns with our suggestions. For long-term remedies, we consider consensus with accumulated reputations as a future work.

7.2 Formalisation

We consider two 51% attacks that make use of externally available mining power: *Mining power migration attacks* and *cloud mining attacks*. *Mining power migration attacks* use mining power from other blockchains, while *cloud mining attacks* use mining power from cloud mining services. We formally study these two 51% attacks by proposing a Markov Decision Process (MDP)-based model called 51-MDP. 51-MDP takes our defined blockchain parameters as input, and outputs an optimal attack strategy with expected revenue of this attack.

7.2.1 System model and notations

We assume miners are rational and blockchains may share the same mining algorithm. For simplicity, our model only considers two blockchains BC_1 and BC_2 with the same mining algorithm. Table 7.1 summarises all notations used in this chapter. Let D_1 and D_2 be the difficulties, R_1 and R_2 be the mining rewards of BC_1 and BC_2, respectively. Let $d = \frac{D_1}{D_2}$ and $r = \frac{R_1}{R_2}$.

Table 7.1: Notations of parameters in 51-MDP.

Symbol	Definition
BC_1, BC_2	The stronger blockchain and the weaker blockchain
D_1, D_2	Difficulty of BC_1 and BC_2
d	Fraction of BC_1's difficulty towards BC_2's difficulty, i.e., $d = \frac{D_1}{D_2}$
$H_{h,1}, H_{a,1}$	Honest and adversary's mining power on BC_1
$H_{h,2}, H_{a,2}$	Honest and adversary's mining power on BC_2
H_a, H_h	Total honest and adversary's mining power, i.e., $H_a = H_{a,1} + H_{a,2}$, $H_h = H_{h,1} + H_{h,2}$
h_1, h_2	Fraction of the adversary's mining power towards BC_1 and BC_2's honest mining power, respectively, i.e., $h_1 = \frac{H_a}{H_{h,1}}$ and $h_2 = \frac{H_a}{H_{h,2}}$
R_1, R_2	Mining reward of a block on BC_1 and BC_2
r	Fraction of BC_1's mining reward of a block towards BC_2's, i.e., $r = \frac{R_1}{R_2}$
v_{tx}	Amount of the attacking transactions
γ	Propagation parameter of the adversary
pr	Renting price of a mining algorithm
β	Fraction of migrated mining power by the adversary
δ	Step of adjusting β
N_c	Number of blocks required to confirm a transaction

As 51% attacks (on BC_2) are usually completed within a short time period, we assumes D_1, D_2, R_1 and R_2 remain stable during the attack.

In a *mining power migration attack*, the adversary migrates its mining power on BC_1 to launch 51% attacks on BC_2. Let $H_{a,1}, H_{a,2}$ be the adversary's mining power, and $H_{h,1}, H_{h,2}$ be the honest mining power on BC_1 and BC_2, respectively. Let $H_a = H_{a,1} + H_{a,2}$, $H_h = H_{h,1} + H_{h,2}$, $H_1 = H_{h,1} + H_{a,1}$ and $H_2 = H_{h,2} + H_{a,2}$. Let $\beta = \frac{H_{a,2}}{H_a}$ be the fraction of mining power that the adversary allocates to BC_2. Let $h_1 = \frac{H_a}{H_{h,1}}$ and $h_2 = \frac{H_a}{H_{h,2}}$ be the ratio between the adversary's mining power and the honest mining power on BC_1 and BC_2, respectively.

In a *cloud mining attack*, the adversary rents mining power to launch 51% attacks on BC_2. We assume that the adversary has sufficient money for renting mining power that is compatible with the victim blockchain BC_2, and there exists unlimited rentable mining power from cloud mining services. To keep notations consistent, we denote the rentable mining power as H_a. Thus, $h_2 = \frac{H_a}{H_{h,2}}$ is the fraction of rented mining power out of rentable mining power, and $\beta = \frac{H_{a,2}}{H_a}$ be the fraction of rented mining power out of the rentable mining power. Let pr be the price of renting a unit of mining power (e.g. hash per second) for a time unit.

Let $\gamma \in [0, 1]$ be the adversary's propagation parameter: When there are two simultaneous blocks mined by the adversary and an honest miner, γ of honest miners receive the adversary's block earlier than the honest block. Let N_c be the required number of blocks for the blockchain network to confirm a transaction.

7.2.2 The 51-MDP model

Table 7.2: State transitions and reward matrices of 51-MDP.

State × Action	Resulting State	Probability	Reward				Condition
			$\mathbb{R}^-_{migration}$	$\mathbb{R}^-_{cloud}$	$\mathbb{R}_{mine}$	$\mathbb{R}_{tx}$	
$(l_h, l_a, \beta, fork)$, ADOPT	$(0, 0, \beta$, ir)	1	0	0	0	0	$l_h > l_a \geq N_c$
$(l_h, l_a, \beta, fork)$, OVERRIDE	$(0, 0, \beta$, ir)	1	0	0	$l_a R_2$	v_{tx}	$l_a > l_h \geq N_c$
$(l_h, l_a, \beta, fork)$, WAIT	$(l_h, l_a$ +1, β, $p)$	$\frac{\beta h_2}{\beta h_2+1}$	$\frac{-\beta h_2 R_1}{d(1+\beta h_2)}$	$\frac{-\beta h_2 D_2 pr}{1+\beta h_2}$	0	0	$l_h < N_c$
	$(l_h$+1, l_a, β, p)	$\frac{1}{\beta h_2+1}$	$\frac{-\beta h_2 R_1}{d(1+\beta h_2)}$	$\frac{-\beta h_2 D_2 pr}{1+\beta h_2}$	0	0	$l_h < N_c$
$(l_h, l_a, \beta, fork)$, WAIT_INC	$(l_h, l_a$+1, $\beta + \delta$, p)	$\frac{(\beta+\delta)h_2}{(\beta+\delta)h_2+1}$	$\frac{-(\beta+\delta)h_2 R_1}{d(1+(\beta+\delta)h_2)}$	$\frac{-(\beta+\delta)h_2 D_2 pr}{1+(\beta+\delta)h_2}$	0	0	$l_h < N_c$
	$(l_h$+1, l_a, $\beta + \delta$, p)	$\frac{1}{(\beta+\delta)h_2+1}$	$\frac{-(\beta+\delta)h_2 R_1}{d(1+(\beta,+\delta)h_2)}$	$\frac{-(\beta+\delta)h_2 D_2 pr}{1+(\beta+\delta)h_2}$	0	0	$l_h < N_c$
$(l_h, l_a, \beta, fork)$, WAIT_DEC	$(l_h, l_a$+1, $\beta - \delta$, p)	$\frac{(\beta-\delta)h_2}{(\beta-\delta)h_2+1}$	$\frac{-(\beta-\delta)h_2 R_1}{d(1+(\beta-\delta)h_2)}$	$\frac{-(\beta-\delta)h_2 D_2 pr}{1+(\beta-\delta)h_2}$	0	0	$l_h < N_c$
	$(l_h$+1, l_a, $\beta - \delta$, p)	$\frac{1}{(\beta-\delta)h_2+1}$	$\frac{-(\beta-\delta)h_2 R_1}{d(1+(\beta-\delta)h_2)}$	$\frac{-(\beta-\delta)h_2 D_2 pr}{1+(\beta-\delta)h_2}$	0	0	$l_h < N_c$
$(l_h, l_a, \beta, fork)$, MATCH	$(l_h, l_a$ +1, β , ir)	$\frac{\beta h_2+\gamma}{\beta h_2+1}$	$\frac{-\beta h_2 R_1}{d(1+\beta h_2)}$	$\frac{-\beta h_2 D_2 pr}{1+\beta h_2}$	$\frac{(l_a+1)R_2\beta h_2}{\beta h_2+\gamma}$	v_{tx}	$l_h = l_a \geq N_c$
	$(l_h$ +1, l_a, β , r)	$\frac{1-\gamma}{\beta h_2+1}$	$\frac{-\beta h_2 R_1}{d(1+\beta h_2)}$	$\frac{-\beta h_2 D_2 pr}{1+\beta h_2}$	0	0	$l_h = l_a \geq N_c$
$(l_h, l_a, \beta, fork)$, MATCH_INC	$(l_h, l_a + 1, \beta + \delta$, ir)	$\frac{(\beta+\delta)h_2+\gamma}{(\beta+\delta)h_2+1}$	$\frac{-(\beta+\delta)h_2 R_1}{d(1+(\beta+\delta)h_2)}$	$\frac{-(\beta+\delta)h_2 D_2 pr}{1+(\beta+\delta)h_2}$	$\frac{(l_a+1)R_2(\beta+\delta)h_2}{(\beta+\delta)h_2+\gamma}$	v_{tx}	$l_h = l_a \geq N_c$
	$(l_h + 1, l_a, \beta + \delta$, r)	$\frac{1-\gamma}{(\beta+\delta)h_2+1}$	$\frac{-(\beta+\delta)h_2 R_1}{d(1+(\beta+\delta)h_2)}$	$\frac{-(\beta+\delta)h_2 D_2 pr}{1+(\beta+\delta)h_2}$	0	0	$l_h = l_a \geq N_c$
$(l_h, l_a, \beta, fork)$, MATCH_DEC	$(l_h, l_a$ +1, $\beta - \delta$, ir)	$\frac{(\beta-\delta)h_2+\gamma}{(\beta-\delta)h_2+1}$	$\frac{-(\beta-\delta)h_2 R_1}{d(1+(\beta-\delta)h_2)}$	$\frac{-(\beta-\delta)h_2 D_2 pr}{1+(\beta-\delta)h_2}$	$\frac{(l_a+1)R_2(\beta-\delta)h_2}{(\beta-\delta)h_2+\gamma}$	v_{tx}	$l_h = l_a \geq N_c$
	$(l_h$ +1, l_a, $\beta - \delta$, r)	$\frac{1-\gamma}{(\beta-\delta)h_2+1}$	$\frac{-(\beta-\delta)h_2 R_1}{d(1+(\beta-\delta)h_2)}$	$\frac{-(\beta-\delta)h_2 D_2 pr}{1+(\beta-\delta)h_2}$	0	0	$l_h = l_a \geq N_c$

The 51-MDP model – summarised in Table 7.2 – describes the attacks as a series of actions performed by an adversary. At any time, the adversary lies in a state, and can perform an action, which transits its state to another state by a certain probability. For each state transition, the adversary may get some reward or penalty. Formally, our 51-MDP model is a four-element tuple (S, A, P, R) where S is the state space containing all possible states of an adversary; A is the action space containing all possible actions performed by an adversary; P is the stochastic transition matrix presenting the probabilities of all state transitions; and R is the reward matrix presenting the rewards of all state transitions.

State space S consists of four dimensions $(l_h, l_a, \beta, fork)$. Parameters l_h and l_a are the length of the honest and the adversary's forks on BC_2, respectively. Eventually, nodes will agree on only one of these two forks. Let $\beta \in [0, 1]$ be the ratio of mining power allocated on BC_2 out of the adversary's total mining power, and $\delta \in [0, 1]$ be the step of adjusting β. We denote the the state of the adversary's fork as $fork$, which has three possible values.

- **Relevant (**$fork = r$**)** means the adversary's fork is published but the honest blockchain is confirmed by the network. This indicates that the attack is unsuccessful at present. (Note that the adversary can keep trying and may succeed in the future.)
- **Irrelevant (**$fork = ir$**)** means the adversary's fork is published and confirmed in network. This indicates a successful attack.
- **Private (**$fork = p$**)** means the adversary's fork is private and only the adversary is mining on it. This indicates that an attack is in process.

Action space A includes actions that the adversary can perform given a state. The adversary's possible actions include:

- **ADOPT.** The adversary accepts the honest blockchain and discards its fork, which means the adversary aborts its attack.
- **OVERRIDE.** The adversary publishes its fork (which is longer than the honest one). Consequently, the honest blockchain is overridden, and the payment transaction from the adversary is successfully reverted.
- **MATCH.** The adversary publishes its fork with the same length as the honest blockchain.
- **WAIT.** The adversary keeps mining on its fork. The adversary can perform **WAIT** in two scenarios. One is when $l_h < N_c$, i.e., the merchant is still waiting for the payment confirmation. The other is when

MATCH has failed, i.e., $N_c < l_a \leq l_h$ but the adversary does not give up its fork.

When performing **MATCH** and **WAIT**, the adversary can adjust mining power allocated to BC_2. We denote two variants of **MATCH** as **MATCH_INC** and **MATCH_DEC**, where the adversary adds and reduces δh_2 mining power allocated to BC_2, i.e., $\beta \mapsto \{\beta + \delta, \beta - \delta\}$, respectively. Similarly, we denote two variants of **WAIT** as **WAIT_INC** and **WAIT_DEC**.

State Transition Matrix P is defined as a 3-dimensional matrix $S \times A \times S$: $\Pr[s, a \mapsto s']$, where S is the state space, and A is the action space. Each point (s, a, s') means that, the participant at state $s \in S$ performs the action $a \in A$ to transit its state to $s' \in S$ with probability $\Pr[s, a \mapsto s']$. An action a transits a state s to one of multiple possible states $s'_1, s'_2, \cdots, s'_n$ with probability $\Pr[s, a \mapsto s'_i]$, where $\sum_{i=1}^{n} \Pr[s, a \mapsto s'_i] = 1$.

When $a =$ **WAIT[_INC, _DEC]**, the adversary is mining its fork alone, until either the honest miners or the adversary mine a new block. The probability of $l_a \mapsto l_a + 1$ (i.e., the adversary mines the next block) and $l_h \mapsto l_h + 1$ (i.e., the honest miners mine the next block) are

$$\Pr[l_a \mapsto l_a + 1] = \frac{H_{a,2}}{H_{a,2} + H_{h,2}} = \frac{\beta H_a}{\beta H_a + H_{h,2}} = \frac{\beta h_2}{\beta h_2 + 1} \tag{7.1}$$

$$\Pr[l_h \mapsto l_h + 1] = 1 - \Pr[l_a \mapsto l_a + 1] = \frac{1}{\beta h_2 + 1} \tag{7.2}$$

When $a =$ **MATCH[_ENC, _DEC]**, the adversary tries to overtake the honest fork once $l_a \geq N_c$ and $l_a = l_h$. Besides the adversary's mining power, the eclipsed mining power of $\gamma H_{h,1}$ mines on the adversary's blockchain after **MATCH**. Therefore, the possibility of $l_a \mapsto l_a + 1$ and $l_h \mapsto l_h + 1$

becomes

$$\Pr[l_a \mapsto l_a + 1] = \frac{\beta H_a + \gamma H_{a,2}}{\beta H_a + H_{h,2}} = \frac{\beta h_2 + \gamma}{\beta h_2 + 1} \tag{7.3}$$

$$\Pr[l_h \mapsto l_h + 1] = 1 - \Pr[l_a \mapsto l_a + 1] = \frac{1-\gamma}{\beta h_2 + 1} \tag{7.4}$$

Reward Matrix R is defined as $S \times A \times S : Re(s, a \mapsto s')$, where the adversary performs action $a \in A$ which transits the system from state $s \in S$ to a new state $s' \in S$ while getting reward $Re(s, a \mapsto s')$. The reward is twofold: the reward of mining $\mathbb{R}_{mine}$ and the reward from the double-spent transactions $\mathbb{R}_{tx}$. The adversary also costs some money on the mining power, and we denote the cost as $\mathbb{R}^-$. Thus, $Re(s, a \mapsto s') = \mathbb{R}_{mine} + \mathbb{R}_{tx} - \mathbb{R}^-$.

$\mathbb{R}_{mine}$. The adversary receives the block reward $\mathbb{R}_{mine}$ on BC_2 only when its fork is published and accepted by the honest network. Therefore, only **OVERRIDE** and the winning scenarios of **MATCH[_INC, _DEC]** have a positive $\mathbb{R}_{mine}$, while $\mathbb{R}_{mine} = 0$ in other scenarios. When performing **OVERRIDE**, the adversary's blockchain of length l_a is directly accepted, so $\mathbb{R}_{mine} = l_a R_2$. When performing **MATCH[_INC, _DEC]**, the adversary needs to win the next block so that its blockchain overrides the honest one, leading to $\mathbb{R}_{mine} = (l_a + 1)R_2$.

$\mathbb{R}_{tx}$. Similar to $\mathbb{R}_{mine}$, the adversary receives the double-spent money only when its fork is published and accepted by the honest network. Therefore, $\mathbb{R}_{tx} = v_{tx}$ for **OVERRIDE** and the winning scenarios of **MATCH**-style actions, while $\mathbb{R}_{tx} = 0$ for other scenarios.

$\mathbb{R}^-$. We analyse the cost of *mining power migration attacks* $\mathbb{R}^-_{migration}$ and *cloud mining attacks* $\mathbb{R}^-_{cloud}$, separately. $\mathbb{R}^-_{migration}$ is the loss of block rewards from BC_1 due to the migrated mining power. Consequently, the cost can be computed as the mining reward of the migrated mining power on BC_1 during the time of state transition. For **ADOPT** and **OVERRIDE** actions, state transitions take negligible time. For **WAIT**-style and **MATCH**-style actions, a state transition is triggered by a new block. Therefore, $\mathbb{R}^{-}_{migration}$ under **WAIT**-style and **MATCH**-style actions can be calculated as follows:

$$\begin{aligned}
\mathbb{R}^{-}_{migration}(l_a \mapsto l_a + 1) = \mathbb{R}^{-}_{migration}(l_h \mapsto l_h + 1) \quad &(7.5)\\
= -\beta H_a \cdot R_1 \cdot \frac{D_2}{H_{h,2} + \beta H_a} \cdot \frac{1}{D_1} \quad &(7.6)\\
= \frac{-\beta h_2 R_1}{d(1 + \beta h_2)} \quad &(7.7)
\end{aligned}$$

$\mathbb{R}^{-}_{cloud}$ is from renting cloud mining power. The price pr of renting cloud mining power is quantified as "the price of renting a unit of mining power for a time unit". Similar with the *mining power migration attack*, only **WAIT**-style and **MATCH**-style actions take a non-negligible time period. Therefore, $\mathbb{R}^{-}_{cloud}$ under **WAIT**-style and **MATCH**-style actions can be calculated as follows:

$$\begin{aligned}
R_{BC_1}(l_a \mapsto l_a + 1) = R_{BC_1}(l_h \mapsto l_h + 1) \quad &(7.8)\\
= -\beta H_a \cdot pr \cdot \frac{D_2}{H_{h,2} + \beta H_a} \quad &(7.9)\\
= \frac{-\beta h_2 D_2 pr}{1 + \beta h_2} \quad &(7.10)
\end{aligned}$$

7.3 Model evaluation

In order to identify the most important aspects on the profitability of 51% attacks, we use 51-MDP to evaluate our two 51% attacks. Together with public blockchain data, attackers can identify blockchains that are most profitable to attack, and defenders can prepare for potential 51% attacks in advance.

7.3.1 Experimental methodology.

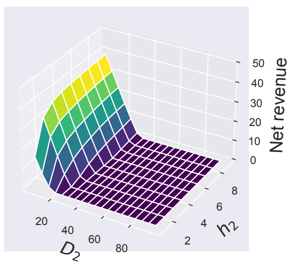
We implement 51-MDP using Python 2.7 and the *pymdptoolbox* library[262]. We give an upper bound $limit = 10$ for l_a and l_h. We choose

$\delta = 0.2$, so the value of β can be $(0.0, 0.2, 0.4, 0.6, 0.8, 1.0)$. We apply the *ValueIteration* algorithm [263] with a discount value of 0.9 and an epsilon value of 0.1. We apply this discount value to encourage the adversary to finish the attack in a short time. In practice, the longer time a 51% attack takes, the more risk it will have. For example, shifting mining power to the victim blockchain might be detected by threat intelligence services. We choose a small discount factor to quantify such risk. We omit the evaluation of *cloud mining attacks* as both attacks share the same parameters D_2, h_2, R_2, v_{tx}, γ and N_c.

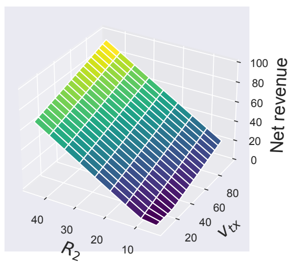
7.3.2 Evaluation

We categorise parameters in 51-MDP to five types according to their related aspects: **1)Mining status** which includes two mining difficulties (D_1 and D_2) and two ratios of adversary's mining power (h_1 and h_2); **2)Incentive** which includes mining reward (R_1 and R_2) and the adversary's transaction amount v_{tx}; **3)Adversary's network condition** which includes the propagation parameter γ of the adversary; **4)Vigilance of the merchant** which includes the number N_c of required block confirmations; and **5)Mining power price** which includes pr only. Figure 7.2 shows the evaluation results.

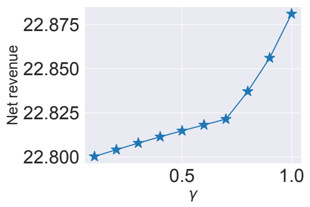
Mining status. Figure 7.2a shows the impact of mining-related parameters on the adversary's net revenue. We observe that the net revenue increases monotonically with D_2 decreasing and h_2 increasing. Mining difficulty variation reflects the fluctuation of network mining power. When D_2 decreases, network mining power decreases, then mining on BC_2 will be easier. Also, launching a 51% attack will be in a lower cost and easier to succeed, which encourages both types of our attacks on BC_2. By these observations, attackers prefer to invest more computing power to BC_2, then h_2 increases by migrating attacker's mining power from other blockchain or renting from



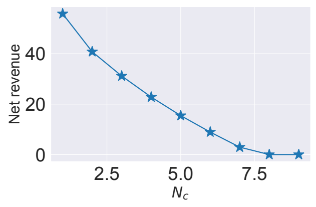
(a) Mining status h_2 and D_2.



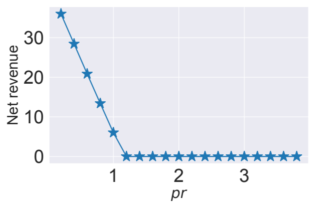
(b) Incentive v_{tx} and R_2.



(c) The adversary's network condition γ.



(d) Vigilance of the merchant N_c.



(e) Mining power price pr.

Figure 7.2: Impacts of parameters on the net revenue of 51% attacks.

cloud services. Therefore, both decreasing D_2 and increasing h_2 incentivise 51% attacks on BC_2.

Incentive. Figure 7.2b shows the impact of incentive-related parameters on the net revenue. We observe that increasing R_2 and v_{tx} leads the adversary to profit more. When R_2 increases, mining BC_2 will be more profitable, and 51% attacks on BC_2 will also be more profitable. This encourages both types of 51% attacks on BC_2. The 51% attack generates v_{tx} out of thin air, so v_{tx} is the direct revenue of the 51% attack, and increasing v_{tx} directly increases the net revenue. Therefore, both increasing R_2 and v_{tx} incentivise 51% attacks on BC_2.

Adversary's network condition. Figure 7.2c shows the impact of γ on the relative revenue. In particular, we can see that the relative reward increases slightly with γ increasing. Interestingly, when the attacker's propagation parameter $\gamma = 0.7$, the curve slope increases.

According to our model, γ counts only when the adversary launches the **MATCH** action. When $h_2 \geq 1$, the adversary can always launch the

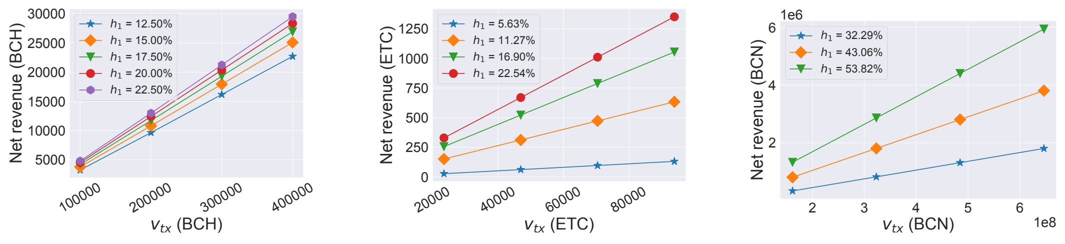
51% attack, regardless of the reward. Therefore, the **MATCH** action is an infrequent choice compared to **OVERRIDE**, so the influence of γ is negligible in our case. The slope change is suspected to be when $\beta H_a + \gamma H_{h,2} \geq (1-\gamma)H_{h,2}$. At that point, the allocated mining power from the adversary plus its eclipsed honest mining power outperforms the un-eclipsed honest power. Consequently, the adversary is confident to override the small blockchain by **MATCH** action.

Vigilance of the merchant. Figure 7.2d shows the impact of N_c on the net revenue. We observe the net revenue decreases monotonically with N_c increasing, and finally reaches 0. More block confirmations require the adversary to keep mining secretly for a longer time. This leads to a lower probability and greater cost of successful 51% attack through both types of attacks, and discourages 51% attacks on BC_2.

Mining power price. The impact of the mining power price pr is shown in Figure 7.2e. We observe that the net revenue decreases sharply with pr increasing, and finally reaches 0. When the price of renting mining power is low, the related blockchains are vulnerable to the cloud mining attack as the attack cost is also low. Increasing pr leads to the greater cost of launching 51% attack through renting cloud mining power, which will discourage this kind of 51% attacks on BC_2.

7.3.3 Analysis

We observe some insights from the results. First, the attacker's profit is mainly affected by the parameters that are out of the attacker's control. The only important parameter that the adversary can control is v_{tx}, which is bound to its budget. Thus, to maximise the profit, an attacker should choose its target carefully. Once choosing the targeted blockchains, the adversary has little control over the attack.



(a) BTC and BCH (b) ETH and ETC (c) XMR and BCN

Figure 7.3: Mining power migration attacks on three different pairs of blockchains. We use $\gamma = 0.3$ for this group of experiments.

In addition, although either the attackers or the defenders cannot fully control important parameters, monitoring them in real-time is possible. By monitoring these parameters, attackers can identify targets with most expected profit, and defenders (e.g. the cryptocurrency exchanges and the merchants) can be aware of potential attacks then perform countermeasures. For example, one of the effective countermeasures is to increase N_c, which greatly reduces the attacker's profit according to Figure 7.2d.

7.4 Evaluation of blockchains in the wild

This section evaluates the security of mainstream PoW-based blockchains against 51% attacks. We evaluate the *mining power migration attack* on 3 pairs of top-ranked blockchains with the same mining algorithm: 1) Bitcoin (BTC) and BitcoinCash (BCH) with Sha256d, 2) Ethereum (ETH) and EthereumClassic (ETC) with Ethash, and 3) Monero(XMR) and ByteCoin (BCN) with CryptoNight. Our evaluation shows that, the *mining power migration attack* is feasible and profitable on BTC/BCH and ETH/ETC, but it is not as effective on XMR/BCN.

For the *cloud mining attack*, we evaluated the security of ten leading PoW-based blockchains. Our evaluation shows that the *cloud mining attacks* are feasible and profitable on most selected blockchains.

7.4.1 Mining power migration attacks

We evaluate the profitability and feasibility of the mining power migration attack on 3 pairs of top-ranked cryptocurrencies with the same mining algorithm: BTC/BCH, ETH/ETC, and XMR/BCN. By permuting the adversary mining power H_a and the transaction value v_{tx}, our experiments reveal their relationship with the relative revenue. As shown in Figure 7.3, it is easy and profitable for a miner of BTC (or ETH) to launch a 51% attack on BCH (resp. ETC). In particular,

- With approximately 12.5% mining power of BTC ($5000E + 15h/s$), an adversary can gain 6% (15000 BCH, or $1,894,650) extra profit (than honest mining) by double-spending a transaction of 300000 BCH (equivalent to $37,893,000).
- With approximately 11.27% mining power of ETH ($16E + 12h/s$), the adversary can gain 1.33% (600 ETC, or $2,556) extra profit by double-spending a transaction of 90000 ETC (equivalent to $383,400).

The required mining power is not difficult to achieve. The top three mining pools in ETH are Sparkpool (30.9%), Ethermine (23.3%), f2pool2 (10.7%) [258]; and the top three mining pools in BTC are F2Pool (17.7%), Poolin (16.1%), BTC.com (11.9%) [264].

However, for XMR, a miner cannot profit much from the *mining power migration attack*. This is because the total available mining power in Monero is only about 2.8 times of the mining power in the BCN, although their market caps differ greatly. Meanwhile, the total available mining power in BTC is about 27.8 times of the total mining power in BCH; and the total available mining power in ETH is about 16.4 times of the total mining power in ETC.

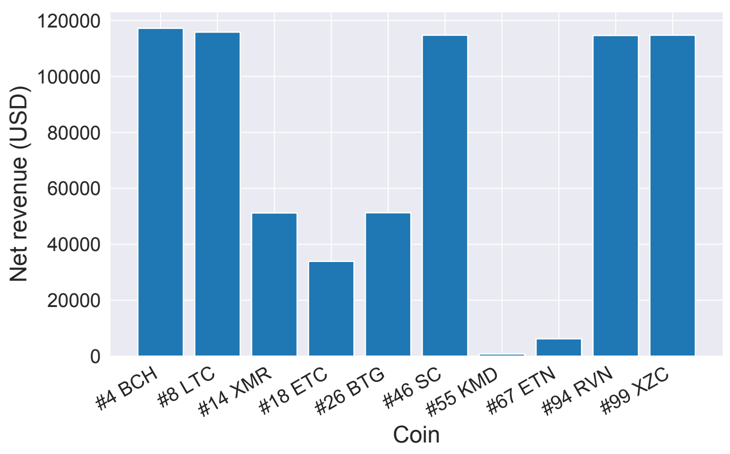


Figure 7.4: Cloud mining attacks on selected 10 PoW blockchains. We use v_{tx} = \$500,000 , $h_2 = 2$ and $\gamma = 0.3$. We use the value of N_c recommended by cryptocurrency communities.

7.4.2 Cloud mining attacks

We evaluate ten leading PoW-based blockchains against the *cloud mining attack*. There are 22 PoW-based blockchains in the top 100 blockchains by market cap [2]. DigiByte and Verge use multiple mining algorithms simultaneously, and NiceHash does not support Bytom, ByteCoin, Electroneum, WaltonChain, and Aion. In addition, NiceHash does not have enough mining power to attack BTC with SHA256D, ETH with Ethash, ZEC with Equihash, DOGE with Scrypt and DASH with X11. Thus, we focus on analysing the rest ten leading blockchains. We set $v_{tx} = \$500,000$ (i.e., the double-spending transaction amount is \$500,000), and $h_2 = 2$ (i.e., the rentable mining power is twice of the honest mining power). We choose the value of N_c according to the recommended values from cryptocurrency community, as listed in Table 7.3.

Figure 7.4 summarises our evaluation results. It shows that, unfortunately, all selected blockchains are vulnerable towards *cloud mining attacks*. For example:

Table 7.3: Data of 15 PoW blockchains and NiceHash prices.

	Rank	**Rent($/h/s)**	**Coin Price($)**	**Hashrate**	N_c
Bitcoin	1	2E-18	3585.99	4E+19	6
Ethereum	3	1.36E-13	118.53	142E+14	12
BitcoinCash	4	2E-18	126.31	1.44E+18	6
Litecoin	8	3.34E-14	30.84	2.77E+14	6
Monero	14	9.13E-11	43.64	9.29E+8	10
Dash	15	3.53E-16	71.79	2.32E+15	6
EthereumClassic	18	1.36E-13	4.26	8.62E+12	12
Zcash	20	1.38E-08	54.77	3.36E+9	6
Dogecoin	23	3.34E-14	0.002132	3.76E+14	6
BitcoinGold	26	1.38E-08	11.93	3170000	6
Siacoin	46	3.74E-17	0.002389	1.88E+15	6
Komodo	55	1.38E-08	0.640292	4.48E+7	30
Electroneum	67	9.13E-11	0.006184	4.4E+9	20
Ravencoin	94	3.36E-13	0.011905	5.9E+12	6
Zcoin	99	2.79E-12	4.83	9.69E+10	6

- the attacker needs approximately $2,000 to launch a *cloud mining attack* on ETC for an hour, and the net revenue will be $33,899 if successful; and
- the attacker needs approximately $2,600 to launch a *cloud mining attack* on BCH for an hour, and the net revenue will be $117,198 if successful.

Evaluation of KMD The only exception is Komodo (KMD): The attacker cannot profit much by launching *cloud mining attacks* on KMD. The reason is that the value of N_c recommended by the KMD community is 30 [265] – much higher than other blockchains. As shown in §7.3, increasing N_c can significantly reduce the profit of 51% attacks. Figure 7.5 shows the profitability of cloud mining attacks on KMD. Although feasible, both attacks on KMD will not give much extra profit - the attacker can only gain 1% $\sim$ 2% more revenue compared to honest mining.

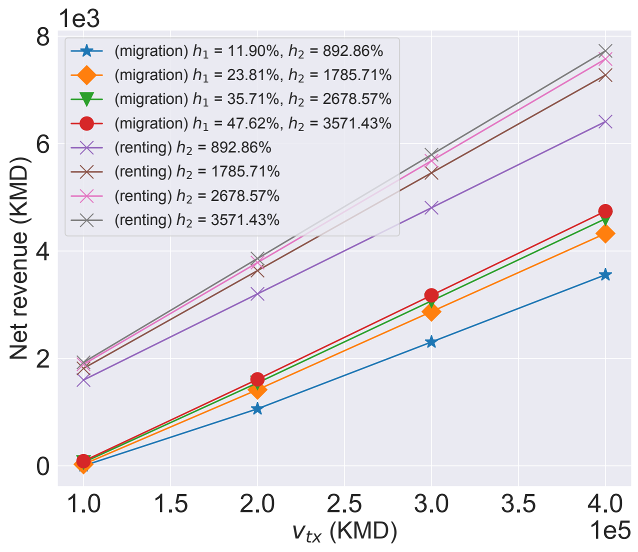


Figure 7.5: Profitability of mining power migration attacks and cloud mining attacks on Komodo (KMD). We choose $\gamma = 0.3$, and $N_c = 30$ - the values recommended by KMD community.

7.5 Case study: The 51% attack on Ethereum Classic

On 07/01/2019, a 51% attack happened to Ethereum Classic (ETC): The attacker double-spent transactions of more than $1.1 million on a cryptocurrency exchange Gate.io[259]. Though the mining power source remains unknown, the attack is highly suspected as a *cloud mining attack*. In this section, we investigate this 51% attack as a case of *cloud mining attacks*. We use 51-MDP to evaluate the attack and estimate the attacker's revenue. The evaluation result shows that the attacker launches the *cloud mining attack* in a fine-grained way, and obtains the theoretically optimal revenue from the attack. We also analyse the attacker's behaviours, and show that the attacker's strategy is the best practice of launching *cloud mining attacks*.

Table 7.4: All 12 double-spent transactions during the 51% attack on ETC[269]. Transaction IDs and addresses are shortened.

Trans. ID	**From**	**To**	**Amount (ETC)**	**Height**	**Waiting time (#block)**
0x1b47a700c0	0x3ccc8f7415	0xbbe1685921	600	7249357	-
0xbba16320ec	0x3ccc8f7415	0x2c9a81a120	4000	7254430	**5073**
0xb5e0748666	0x3ccc8f7415	0x882f944ece	5000	7254646	216
0xee31dffb66	0x3ccc8f7415	0x882f944ece	9000	7255055	409
0xfe2da37fd9	0x3ccc8f7415	0x2c9a81a120	9000	7255212	157
0xa901fcf953	0x3ccc8f7415	0x2c9a81a120	15700	7255487	275
0xb9a30cee4f	0x3ccc8f7415	0x882f944ece	15700	7255554	67
0x9ae83e6fc4	0x3ccc8f7415	0x882f944ece	24500	7255669	115
0xaab50615e3	0x3ccc8f7415	0x53dffbb307	5000	7256012	343
0xd592258715	0x07ebd5b216	0xc4bcfee708	26000	7261492	**5480**
0x9a0e8275fc	0x07ebd5b216	0xc4bcfee708	52800	7261610	118
0x4db8884278	0x07ebd5b216	0xc4bcfee708	52200	7261684	74
					Total: 219500 ETC

7.5.1 The attack details

Ethereum Classic (ETC) is a PoW-based blockchain forked from Ethereum (ETH). In 07/01/2019, a 51% attack on ETC resulted in the loss of more than 1.1 million dollars. The attack lasted for 4 hours, approximately from 0:40 am to 4:20 am UTC, 07/01/2019. During the attack, the attacker repetitively created coin withdrawal transactions on the Gate.io cryptocurrency exchange[266] and launched double-spending attacks[259]. Among these attempts, 12 transactions were successfully double-spent (listed in Table 7.4). Interestingly, the attacker later returned ETC equivalent to $100,000 back to Gate.io[261].

While the source of the mining power for this attack remains unknown, the NiceHash cloud mining platform [13] is highly suspected. One day before the attack, an anonymous person rented all available Ethash (the mining algorithm used by ETH/ETC) mining power from NiceHash[267], [268].

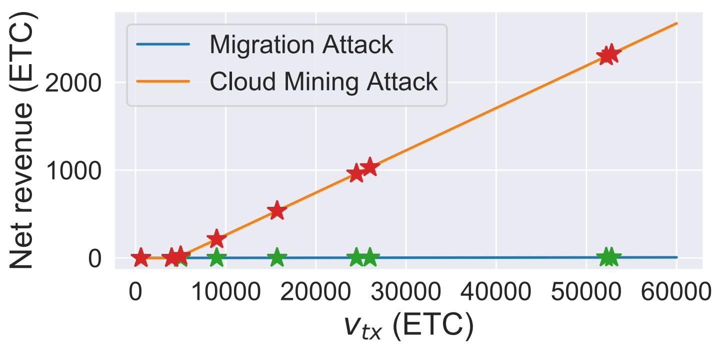


Figure 7.6: Simulated 51% attack on ETC. The blue line denotes the relative reward of *cloud mining attacks*. The orange line denotes the relative reward of *mining power migration attacks* for making comparisons. We also marked different transaction amounts in the attack using dots.

7.5.2 Evaluation

Table 7.4 summarises the attack-related data. According to Gate.io[266], during the attack's time period, $N_c = 12$ – the recommended value of the ETH community and ETC community[270]. The price of ETC and BTC was \$5.32 and \$4061.47, respectively. The mining difficulty of ETC was 131.80E+12, and the ratio h_2 was about 1.16. The block reward is 4 ETC coins, and the price of Nicehash mining power was 3.8290 BTC/TH/-day. We keep assuming $\gamma = 0.3$ as there is no data on γ and the impact of γ is relatively small. Figure 7.6 shows our evaluation result. We mark the transaction values used by the attacker. We also plot the same curve in the *mining power migration attack* to compare the profitability of two mining power sources.

The result shows that when the transaction value is over 5000 ETC, double-spending is more profitable than by honest mining. Having a transaction (or a set of transactions) of value over 5000 ETC (approximately \$26,000 at the time of attack) should not be difficult for an attacker, so the incentive of launching double-spending attacks is very strong. In addition, *cloud mining attacks* are more profitable than *mining power migration attacks*. This means that renting mining power to attack ETC is much cheaper than migrating mining power from ETH. This is because both ETH and ETC use Ethash[271] as the mining algorithm. Ethash is a memory-hard function, making it GPU-friendly while ASIC-resistant[272]. Thus, any GPU can be used for mining ETH/ETC, making mining power much cheaper than that from dedicated hardware such as ASICs.

7.5.3 Estimating the attacker's net revenue

According to Table 7.4, the attacker has stolen 219,500 ETC, which is the attacker's gloss revenue. As we don't know transactions of failed 51% attack attempts, the cost of the attack is unknown. Thus, it's hard to determine the cost of attacks, and we cannot calculate the attacker's revenue directly. Nevertheless, we can apply 51-MDP to estimate the attacker's net revenue. As we know the amount of mining power of the attacker, we can estimate the success rate of attacks. With the success rate, we can estimate the total amount of transactions for failed attacks, and therefore derive the total amount of double-spending transactions. With the total amount and blockchain data as input, 51-MDP can estimate the attacker's net revenue. By using this method, we find that our estimated net revenue is approximately $84773.40, which is close to $100,000 – the value that the attacker returned to Gate.io after the attack[261].

Modelling. We first calculate the success rate of the attack. Let N_c be the required number of blocks to confirm transactions, and h_2 be the ratio of attacker's mining power over the honest network. Then, the attacker controls $p = \frac{h_2}{h_2+1}$ of the total mining power. Mining can be modelled as a binomial distribution $B(n_a + n_h, p)$ where n_a and n_h are the numbers of blocks that the adversary and the honest miners have mined, respectively. Let $\Pr[X = n_a]$ be the probability of the attacker to mine n_a blocks while honest miners mine h_h blocks, and we have

$$\Pr[X = n_a] = \Pr[n_a; n_a + N_c, p] \tag{7.11}$$

When $n_h = N_c \wedge n_a < N_c$, the attack fails. Thus, the probability P of a successful 51% attack is calculated as

$$P = 1 - \sum_{n_a=0}^{N_c-1} \Pr[n_a; n_a + N_c, p] \tag{7.12}$$

Then, we estimate the net revenue from observed successful attacks. Let R_s and R_f be the estimated revenue of successful and failed attack attempts, respectively. We have

$$\frac{R_s}{P} = \frac{R_s}{1-P} \Longrightarrow R_f = \frac{(1-P)R_s}{P} \tag{7.13}$$

and the estimated total net revenue R is

$$R = R_s + R_f = R_s + \frac{(1-P)R_s}{P} \tag{7.14}$$

Estimation. Summing profits of all successful transactions in Figure 7.6, the attacker's gloss revenue is approximately 9000 ETC coins ($R_S = 9000$). Recall that $h_2 = 1.16$, and the attacker controls $p = \frac{h_2}{h_2+1} = 53.7\%$ of ETC mining power. Recall that $N_c = 12$ in ETC. From Equation 7.12, the success rate P of an attack can be calculated as

$$P = 1 - \sum_{n_a=0}^{N_c-1} Pr[n_a; n_a + N_c, p] \tag{7.15}$$

$$= 1 - \sum_{n_a=0}^{N_c-1} C_{n_a+N_c}^{n_a} p^{n_a} (1-p)^{N_c} \tag{7.16}$$

$$= 56.48\% \tag{7.17}$$

From Equation 7.14, we calculate the estimated net revenue R as

$$R = R_s + R_f = R_s + \frac{(1-P)R_s}{P} \tag{7.18}$$

$$= 9000 + \frac{(1-0.5648) \cdot 9000}{0.5648} \tag{7.19}$$

$$= 9000 + 6934.85 = 15934.85 \; (ETCcoins) \tag{7.20}$$

Therefore, the attacker's net revenue is expected to be $9000 + 6934.85 = 15934.85$ ETC coins. At the time of attack, 15934.85 ETC coins is equivalent to \$84773.40, which is slightly less than \$100,000 – the amount that the attacker returned to Gate.io. To achieve the optimal revenue, the attacker should launch *cloud mining attacks* using the optimal strategy, which is usually fine-grained. This indicates that, the attacker adopted a near optimal strategy for launching *cloud mining attacks*.

7.6 The attacker's strategy

According to Table 7.4, the attacker continuously increased the value of new transactions throughout the attack (except the last double spending of the first account). It is suspected that this behaviour belongs to the strategies used by the attacker to maximise and stabilise its revenue, for the following reasons.

Stabilising the revenue. First, launching multiple small double-spending attempts can stabilise the expected revenue. Double-spending attacks may fail even if the adversary controls more than 50% of the computing power. Compared to a one-off attempt, the revenue will be more stable if dividing a transaction into multiple smaller transactions.

Bypassing risk management systems. Second, this strategy may be used for bypassing risk management systems of cryptocurrency exchanges. Cryptocurrency exchanges run risk management systems to combat misbehaviours, including fraudulent payments and abnormal login attempts. A huge coin withdrawal transaction is very likely to trigger the risk management system, while multiple small transactions might be overlooked. In addition, a big transaction may lead to longer confirmation time, and a longer attack period is easier to be detected. Therefore, bypassing the risk management system is naturally a part of the attacker's strategy. According to the Gate.io report [259], the risk management system ignored transactions from the attacker, as the attack was decently prepared – they registered and real-name authenticated the account on Gate.io more than 3 months before the attack. The attacker slowly increasing the transaction value is also highly suspected as an approach for reverse-engineering the threshold of invoking the risk management system.

Using multiple wallets. In addition, we investigate the waiting time between each two attacks (quantified by using the number of blocks). The waiting time varies mostly from 67 blocks to 409 blocks. Interestingly, there are two large gaps of more than 5000 blocks before the transactions 0xbba16320ec and 0xd592258715. The first gap is after the first attack, and the second gap is before the attacker changed its account. The first gap may be because the attacker was cautious when first launching the double-spending attack. The attacker double-spent a transaction of 600 ETC coins, which is much smaller than its following transactions. After the first attack, the attacker waited for a long time to confirm the success of it, then started to increase the transaction value. The second gap may be because the attacker ran out of money in its first account 0x3ccc8f7415, so changed to another account 0x07ebd5b216. The last transaction 0xd592258715 sent by account 0x3ccc8f7415is is right before the second gap. It's value is 5000 ETC coins, which is much smaller than its previous transaction of 24500 ETC coins. After the transaction 0xd592258715, the attacker changed to its another account 0x07ebd5b216, leading to the second time gap.

7.7 Discussions on attack prevention

This section discusses short term and long term solutions to detect and prevent both the *mining power migration attack* and *cloud mining attack*. We make use of the 51% attack incident on ETC (see §7.5) as an example, and demonstrate how to make use of 51-MDP to gain insights that helps to defend against such cloud mining attacks in Section 7.7.1.

7.7.1 Quick remedies

We first discuss several quick remedies for cryptocurrency exchanges to reduce the damage of 51% attacks. It consists of detecting potential attack attempts, and reacting upon detection through conventional risk management techniques.

Detecting 51% attacks. For the two 51% attacks, the attacker needs to move a considerable amount of mining power from somewhere, such as the other blockchain or a cloud mining service.

This gives us an opportunity to detect the anomaly state where a "large" portion of mining power suddenly disappears from a source. For example, a potential victim can monitor the available compatible mining power of other blockchains or cloud mining services. If there is a sudden change on the amount of total available mining power, then this might indicate a potential 51% attack. The threshold of "large" is blockchain specific according to the risk management rules. For example, a blockchain which cares less on such attacks can set the threshold to 100% of its current total mining power. That is, once the disappearance of this amount of mining power in other sources is detected, then an alarm of a potential attack is raised. However, this will not detect an attacker who gains 90% mining power from one source, and 10% from another sources. A more cautious blockchain may set a tighter threshold, e.g. 5%, however, this may cause false positive alarms.

There are two limitations of this method. First, it may introduce false positive detections, and it is hard to identify which blockchain will be the victim upon detection. Second, it is expensive to monitor all the possible mining compatible blockchains and cloud mining services in real-time. Even though, the monitoring result may be inaccurate.

Reactions upon 51% attacks. Upon detecting the two 51% attacks, the exchange can take several reactions to prevent them from happening. First, the exchange can increase the number N_c of block confirmations. According to Figure 7.7, for the 51% attack on ETC in 2019, the attack can be avoided if increasing N_c to 18. The ETC community's action further proves the effectiveness of increasing N_c: After the last 51% attack [244], the ETC community urged to raise N_c to 10,000 [273], while it takes approximately two weeks to generate 10,000 blocks. Second, the exchange can decrease the maximum amount of cash out. Figure 7.2b and 7.7 show the impact of the transaction amount v_{tx} on the 51% attack on ETC. If the maximum amount of cash out was limited to 9,000 ETC (approximately $38340.0), then the attacker would no longer profit. Third, limiting the frequency of cash out also discourages 51% attacks. With a limited frequency of cash out, the attacker will need more time to launch attacks, and thus the attack takes more opportunity cost. Last, when the risk management system considers attacks

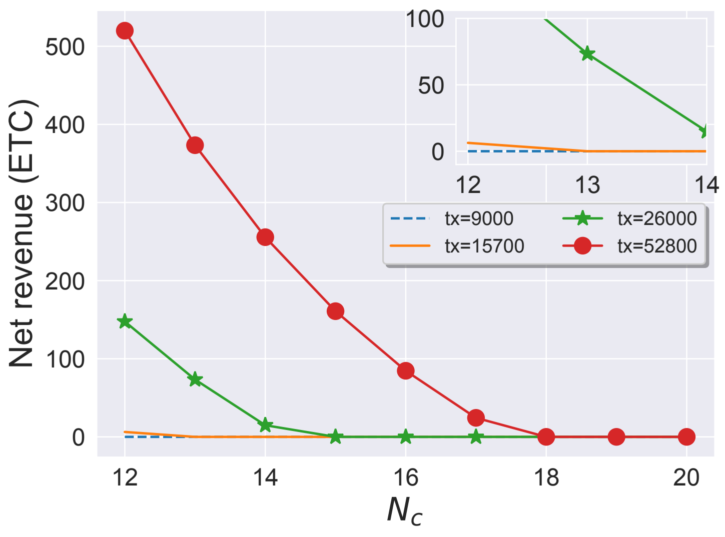


Figure 7.7: Impacts of v_{tx} and N_c on the ETC attack.

are likely to happen, then the exchange can halt all cash out temporarily. We consider quantifying the effectiveness of these countermeasures based on existing Anti-Money Laundering models as future work.

7.7.2 Long term solutions

Though easy to deploy, these quick remedies are not sufficient. First, they sacrifice the usability of blockchains. Second, all of them only minimise the effect of the potential attacks, rather than eliminating them.

Improving the PoW protocol from the protocol-level is also a promising approach to defend against our attacks. There are limited works aiming at minimizing the effects of powerful miners being malicious. For example, RepuCoin[48] aims at mitigating the 51% attacks in PoW protocols by introducing the "physics-based reputation". In RepuCoin, the weight of each miner is decided by the reputation rather than the mining power. The reputation of a miner depends on the mining power, but also takes the past contribution of miners into consideration. In this way, a 51% attacker cannot gain a high-enough reputation within a short time period, and the 51% attacks we studied become much harder to launch.

Table 7.5: Taxonomy of existing attacks and analyses.

		System setting	
		Standalone system	System + external environment
Adversary's objective	Optimise profit	Selfish mining [246]–[249]	Fickle mining [274], [275]
	Break consistency & liveness	51% attacks [250]–[257]	Bribery attacks [14], [257], [276], [277] + **This work**

7.8 Related work

Table 7.5 classifies existing attacks and analyses on PoW-based blockchains. To our knowledge, we are the first to challenge the honest majority assumption of PoW-based blockchains in the presence of externally available mining power. Most existing papers [246]–[257] analyse PoW-based blockchains while assuming the honest majority and omitting external factors. In this section, we compare our work with two closely related work, namely fickle mining[274], [275] and bribery attacks[14], [257], [276], [277].

Fickle mining[274], [275] is that, a miner adaptively allocates mining power on two blockchains with the same mining algorithm (e.g., BTC and BCH) for extra profit. Similar to *mining power migration attacks*, fickle mining also consider miners' behaviours between multiple blockchains. While fickle mining assumes the honest majority and miners mine honestly, we consider the honest majority can be broken and rational miners can launch 51% attacks.

Bonneau et al.[14] introduce the family of bribery attacks, where an adversary bribes other miners and asks them to launch 51% attacks. They discuss two bribery attacks: one is our *cloud mining attack*, and the other is by creating a mining pool with negative fee. While Bonneau et al.[14]

only informally discuss them, we formally study the *cloud mining attack* and additionally consider *mining power migration attack*. There have been new bribery attack vairants[257], [276], [277], where an adversary bribes miners to mine on a previous block and fork the blockchain. While these attacks are triggered by an external adversary, our results show that even without an external adversary, miners are incentivised to launch 51% attacks.

Chapter 8

On the optionality and fairness of Atomic Swaps

8.1 Introduction

Atomic Swap allows two parties on two blockchains to exchange their assets "atomically" without trusted third parties. The swap is "atomic" in the sense that the swap either succeeds or fails for both parties. It can be used for supporting cross-shard transactions in sharded blockchains. For example, a user Alice who has some coins in shard #1 hopes to make transactions on a smart contract in shard #2, thus needs to obtain some coins in shard #2. To this end, Alice can initiate an Atomic Swap with Bob, where Alice provides coins in shard #1 and Bob provides coins in shard #2. By doing Atomic Swap with users in different shards, Alice can transact on any shard in a sharded blockchain.

Meanwhile, Atomic Swap is a central primitive in many applications, such as Decentralised Exchanges (DEXes), Decentralised Finance (DeFi) platforms. To date, there are more than 250 DEXes [278], more than 30 DEX protocols [279], and more than 4,000 active traders [280] constituting the market volume of about 50,000 ETH [280].

Atomic Swap can be implemented by using Hashed Time-locked Contracts (HTLCs) [281]. The HTLC is a type of transaction that, the payee should provide the preimage of a hash value before a specified deadline, otherwise the payment fails - the money goes back to the payer and the payee will not get any money.

However, being atomic does not indicate the Atomic Swap is fair. In an Atomic Swap, the swap initiator can decide whether to proceed or abort the swap, and the default maximum time for him to decide is 24 hours [15]. This enables the swap initiator to speculate without any penalty. More specifically, the swap initiator can keep waiting before the timelock expires. If the price of the swap participant's asset rises, the swap initiator will proceed the swap so that he will profit. If the price of the swap participant's asset drops, the swap initiator can abort the swap, so that he won't lose money.

A user with ID "ZmnSCPxj" at the Lightning-dev mailing list [282] pointed out that, this problem is equivalent to the Optionality in Finance, which has already been studied for decades [283]. In Finance, an investment is said to have optionality if 1) settling this investment happens in the future rather than instantly; 2) settling this investment is optional rather than mandatory. For an investment with optionality, the option itself has value besides the underlying asset, which is called the *premium*. The option buyer should pay for the premium besides the underlying asset, even if he aborts the contract. In this way, he can no longer speculate without penalties.

In the Atomic Swap, the swap initiator has the optionality, as he can choose whether to proceed or abort the swap. Unfortunately, the swap initiator is not required to pay for the premium - the Atomic Swap does not take the optionality into account. Furthermore, Atomic Swap should not have optionality. Atomic Swap is designed for currency exchange, and the currency exchange has no optionality. Instead, once both parties agree on a currency exchange, it should be settled without any chance to regret.

8.1.1 Contributions

In this chapter, we investigate the unfairness of the Atomic Swap. We start from describing the Atomic Swap and the American Call Option in Finance, then we show how an Atomic Swap is equivalent to a premium-free American Call Option. After that, we then evaluate how unfair the Atomic Swap is from two different perspectives, namely quantifying the unfairness and estimating the premium. Furthermore, we propose an improvement on the Atomic Swap, which implements the premium mechanism, to make it fair. Our improvement supports blockchains with smart contracts (e.g. Ethereum) directly, and can support blockchains with scripts only (e.g. Bitcoin) by adding a single opcode. We also implement our protocol in Solidity (a smart contract programming language for Ethereum), and give detailed instructions on implementing our protocols on Bitcoin. Specifially, we make the following contributions.

We show that the Atomic Swap is equivalent to the premium-free American Call Option, and thus is unfair to the participant (§8.3). We describe the Atomic Swap and the American Call Option, then show that an Atomic Swap is equivalent to a premium-free American Call Option, which is a type of Options (in Finance). More specifically: the initiator and the participant in an Atomic Swap are the option buyer and the option seller in an American Call Option, respectively; the initiator asset and the participant asset in an Atomic Swap are the used currency and the underlying asset in an American Call Option, respectively; the participant asset's timelock in an Atomic Swap is the strike time in an American Call Option; the current price of the participant asset in an Atomic Swap is the strike price in an American Call Option; redeeming cryptocurrencies in an Atomic Swap is equivalent to exercising the contract in an American Call Option.

Thus, Atomic Swap is unfair to the participant, especially in the highly volatile cryptocurrency market. In practice, the initiator can decide whether to proceed the swap while investigating the cryptocurrency market. However, proceeding or aborting the swap does not require the initiator to pay for the premium. This leads to the scenario that, if the participant's asset price rises before the strike time, he will proceed the swap to profit, otherwise he will abort the swap to avoid losing money. In this way, the swap initiator can speculate without any risk in Atomic Swaps.

We quantify the unfairness and the premium's value in Atomic Swap (§8.4). We quantify how unfair the Atomic Swap with mainstream cryptocurrency pairs is, and compare this unfairness with those of conventional financial assets (stocks and fiat currencies). We first classify the unfairness to two parts, namely the profit when the price rises and the mitigated loss when the price drops, then quantify them based on historical exchange rate volatility. Our results show that, in the default timelock setting, the profit and the mitigated loss of our selected cryptocurrency pairs are approximately 1%, while for stocks and fiat currencies the values are approximately 0.3% and 0.15%, respectively.

We use the Cox-Ross-Rubinstein option pricing model to estimate how much the premium should be for Atomic Swaps. In Finance, the Cox-Ross-Rubinstein model [16] is the conventional option pricing model for American-style options. Our results show that, in the default timelock setting, the premium should be approximately 2% for Atomic Swaps with cryptocurrency pairs, while the premium is approximately 0.3% for American Call Options with stocks and fiat currencies. Also, the premium values rise for all assets with the strike time increasing, then start to converge when the strike time reaches 300 days.

We propose an improvement on the Atomic Swap to make it fair (§8.5). With the observation that the unfairness is from the premium, we propose an improvement on the Atomic Swap, which implements the premium mechanism, to make it fair. It supports both the currency exchange-style Atomic Swap and the American Call Option-style Atomic Swap. In the currency exchange-style Atomic Swap, the premium will go back to the swap initiator if the swap is successful. In the American Call Option-style Atomic Swap, the premium will definitely go to the swap participant if the participant participates in the swap.

We describe how to implement our protocol on existing blockchains (§8.6). We give instructions to implement our protocols on existing blockchains, including blockchains supporting smart contracts and blockchains supporting scripts only. For blockchains supporting smart contracts (e.g. Ethereum), our protocol can be directly implemented. For blockchains supporting scripts only (e.g. Bitcoin), our protocol can be implemented by adding one more opcode. We call the opcode "OP_LOOKUP_OUTPUT", which looks up the owner of a specific UTXO output. We give the reference implementation in Solidity as an example of smart contracts. We also give that in Bitcoin script (which assumes "OP_LOOKUP_OUTPUT" exists) as an example of scripts.

8.2 Background

In this section, we explain basic concepts of Atomic Swap and the Option (in Finance).

8.2.1 Atomic Swap

An Atomic Swap [15] is that two parties exchange their assets "atomically". "Atomic" means the swap is indivisible: It either succeeds or fails for both parties.

In Blockchain, the Hashed Time-locked Contract (HTLC) [281] enables the Atomic Swap without trusted third parties. HTLC was originally introduced to secure routing across multiple payment channels [284]. In a HTLC-style transaction, the payee can redeem the payment prior to a deadline only by providing the preimage of a specific hash value, otherwise the payment will expire and the money will go back to the payer. This is achieved by the hashlock - to lock the payment by a hash value, and the timelock - to give the deadline of redeeming. The timelock avoids locking money in a payment forever when the payee cannot provide the preimage.

8.2.2 Option in Finance

In Finance, an option is a contract which gives the option buyer the right to buy or sell an asset, at a specified price prior to or on a specified date [283]. Here the option buyer can choose whether to fulfill the contract. The specified price is called the *strike price*; the specified date is called the *strike time*; the party proposing the option is called the *option seller*; the other party choosing to fulfill or abort the contract is called the *option buyer*; the asset is called the *underlying asset*; and fulfilling the contract is called *exercising*.

The option has two types, namely the American-style Option and the European-style Option. They differ from the *strike time*: The European-style Option buyer can only exercise the contract on the strike time, and the American-style Option buyer can exercise the contract no later than the strike time.

Who holds the option is irrelevant with who is buying the underlying asset. More specifically, the option buyer is who can decide to exercise or abort the contract. Whether the option buyer is buying or selling the underlying asset depends on the option contract. In Finance, if the option buyer is the party buying the underlying asset, this option is a "Call Option", otherwise this option is a "Put Option" [285].

Besides the underlying asset, the option contract itself is considered to have value. The value of the contract is called the *premium*. The option buyer should pay for the *premium* to the option seller once both parties agree on the option contract.

The *premium* is priced prior to the contract agreement. As the *premium* is the only variable within the option contract, pricing the *premium* is also known as the *option pricing* problem. *Option Pricing* is rather a complex task, and is still a hot research topic in Finance and Applied Mathematics.

The Black-Scholes (BS) Model is the first widely used model for option pricing [286]. It can estimate the value of European-style Options using the historical price of the underlying asset. The Cox-Ross-Rubinstein (CRR) model [16], also known as the Binomial Option Pricing model, extends the BS model for pricing American-style Options.

8.3 Atomic Swap and American Call Option

In this section, we describe the Atomic Swap (the original version on Bitcointalk [15]) and the American Call Option, then point out that an Atomic Swap is equivalent to an American Call Option without the premium.

8.3.1 Atomic Swap

Security assumptions

First, we assume blockchains involved in the Atomic Swap are secure, and execute all transactions correctly. The Atomic Swap is based on blockchains. If the blockchains are insecure, the Atomic Swap will also be insecure.

Second, we assume the HTLC mechanism in blockchains is reliable. More specifically, 1) blockchains produce new blocks with stable speeds; 2) the hash algorithms used by HTLCs are secure; 3) blockchains execute HTLCs correctly.

Third, the time for confirming a transaction is negligible compared to timelocks in HTLCs. In practice, the swap initiator's timelock is 48 hours and the swap participant's timelock is 24 hours by default [15], while confirming a transaction is less than 1 hour for most blockchains.

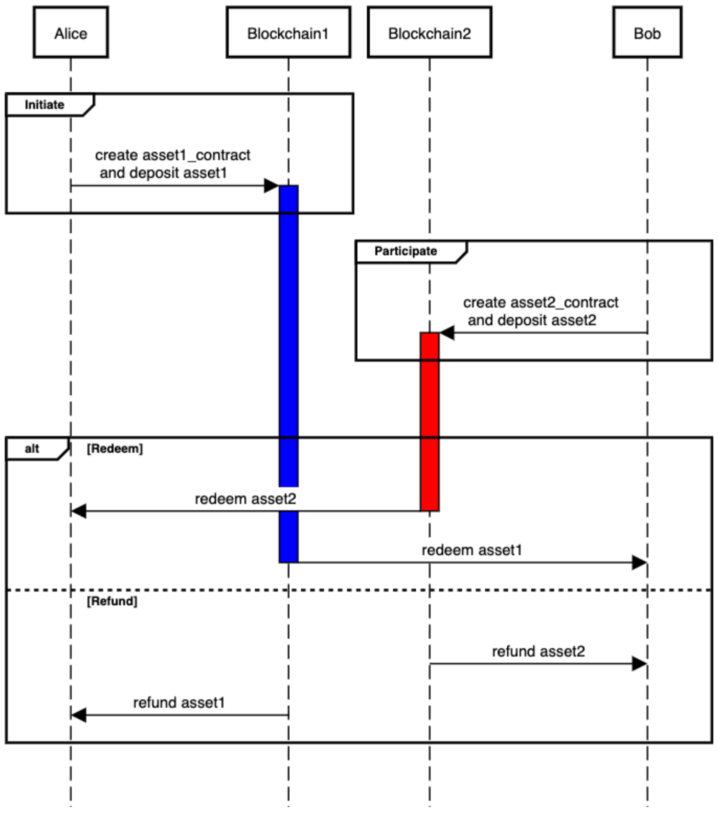


Figure 8.1: Sequence diagram of the Atomic Swap.

Process

Assuming the swap initiator Alice hopes to get x_2 $Coin_2$ from the swap participant Bob in exchange of x_1 $Coin_1$. $Coin_1$ is the cryptocurrency on the blockchain BC_1, and $Coin_2$ is the cryptocurrency on the blockchain BC_2. We denote the Atomic Swap as

$$\mathcal{AS} = (x_1, Coin_1, x_2, Coin_2)$$

Let Alice be the holder of the address $\beta_{A,1}$ on BC_1 and the address $\beta_{A,2}$ on BC_2. Let Bob be the holder of the address $\beta_{B,1}$ on BC_1 and the address $\beta_{B,2}$ on BC_2. $\beta_{A,1}$ holds $Coin_1$ with the amount no smaller than x_1, and $\beta_{B,2}$ holds $Coin_2$ with the amount no smaller than x_2.

Figure 8.1 shows the process of $\mathcal{AS}$. In detail, $\mathcal{AS}$ consists of four stages: **Initiate**, **Participate**, **Redeem**, and **Refund**.

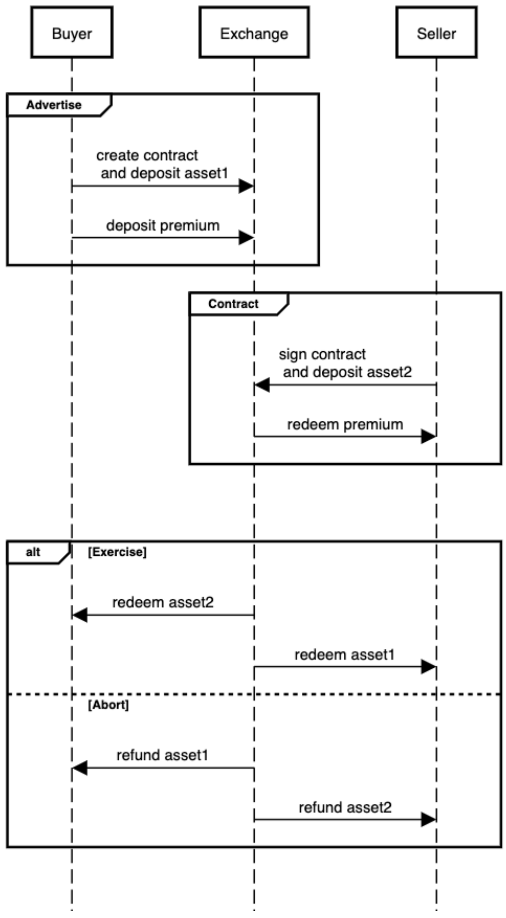


Figure 8.2: Sequence diagram of the American Call Option.

Initiate. Alice initiates $\mathcal{AS}$ at this stage. First, Alice picks a random secret s only known to herself, and computes the hash $h = \mathcal{H}(s)$ of s, where $\mathcal{H}$ is a secure hash function. Then, Alice creates an HTLC script $\mathcal{C}_1$ that "Alice pays x_1 $Coin_1$ from $\beta_{A,1}$ to $\beta_{B,1}$ if Bob can provide s which makes $\mathcal{H}(s) = h$ before or on a timelock δ_1 (which is a timestamp). After δ_1, Alice can refund the money - get x_1 $Coin_1$ back." After creating $\mathcal{C}_1$, Alice publishes $\mathcal{C}_1$ as a transaction $tx_{\mathcal{C},1}$ on BC_1. Note that h is published when publishing $tx_{\mathcal{C},1}$. Besides $\mathcal{C}_1$, Alice also creates a refund script $\mathcal{R}_1$ that "Alice pays x_1 $Coin_1$ from $\beta_{A,1}$ to her another address." This is to ensure x_1 $Coin_1$ can no longer be redeemed by others. Alice can publish $\mathcal{R}_1$ only after δ_1. If Bob does not redeem x_1 $Coin_1$ and δ_1 expires, Alice can refund x_1 $Coin_1$ by publishing $\mathcal{R}_1$ as a transaction $tx_{\mathcal{R},1}$ on BC_1.

Participate. Bob participates in $\mathcal{AS}$ after **Initiate**. With the published h in $tx_{\mathcal{C},1}$, Bob creates another HTLC script $\mathcal{C}_2$ that "Bob pays x_2 $Coin_2$ from $\beta_{B,2}$ to $\beta_{A,2}$ if Alice can provide s before or on a timelock δ_2 (which is a timestamp). After the time of δ_2, Bob can refund the money - get x_2 $Coin_2$ back." Here δ_2 should expire before δ_1. After creating $\mathcal{C}_2$, Bob publishes $\mathcal{C}_2$ as a transaction $tx_{\mathcal{C},2}$ on BC_2. Note that Alice knows s so she can redeem x_2 $Coin_2$ in $tx_{\mathcal{C},2}$ anytime before δ_2, but Bob cannot redeem x_1 $Coin_1$ in $tx_{\mathcal{C},1}$ because he does not know s. Besides $\mathcal{C}_2$, Bob also creates a refund script $\mathcal{R}_2$ that "Bob pays x_2 $Coin_2$ from $\beta_{B,2}$ to his another address." This is to ensure x_2 $Coin_2$ can no longer be redeemed by Alice. Bob can do this only after δ_2. If Alice does not redeem x_2 $Coin_2$ before δ_2 expires, Bob can refund x_2 $Coin_2$ by publishing $\mathcal{R}_2$ as a transaction $tx_{\mathcal{R},2}$ on BC_2.

Redeem/refund. At this stage, Alice can choose either to redeem x_2 $Coin_2$ or refund x_1 $Coin_1$. Note that both **Redeem** and **Refund** are atomic: if Alice chooses to redeem x_2 $Coin_2$, Bob can also redeem x_1 $Coin_1$; if Alice chooses to refund x_1 $Coin_1$, Bob can also refund x_2 $Coin_2$.

- **Redeem.** Alice redeems x_2 $Coin_2$ by publishing s, then Bob can also redeem x_1 $Coin_1$ with the published s. First, Alice provides s to $tx_{\mathcal{C},2}$ in order to redeem x_2 $Coin_2$ in $tx_{\mathcal{C},2}$. As a result, Alice redeems x_2 $Coin_2$, but exposes s to Bob. After that, Bob provides s to $tx_{\mathcal{C},1}$ in order to redeem x_1 $Coin_1$ in $tx_{\mathcal{C},1}$. In this way, Alice and Bob successfully exchanges x_1 $Coin_1$ and x_2 $Coin_2$.

- **Refund.** If Alice does not redeem x_2 $Coin_2$ after δ_2 expires, Bob can refund his x_2 $Coin_2$ by publishing $tx_{\mathcal{R},2}$. As a result, Alice cannot redeem x_2 $Coin_2$, and will not publish s. After δ_1, Alice can also refund her x_1 $Coin_1$ by publishing $tx_{\mathcal{R},1}$.

Atomicity analysis. We can see that $\mathcal{AS}$ either succeeds or fails for both Alice and Bob. In detail,

- If Alice misbehaves when triggering **Initiate**, Bob will lose nothing as he hasn't deposited x_2 $Coin_2$ yet.
- If Bob misbehaves when triggering **Participate**, Alice can choose to abort $\mathcal{AS}$ by triggering **Refund**.
- Alice can only choose to redeem x_2 $Coin_2$ by triggering **Redeem** or wait δ_2 to expire. Once Alice triggers **Redeem**, Bob can also trigger **Redeem**. Once δ_2 expires, Bob can trigger **Refund** to get his x_2 $Coin_2$ back.

However, one may take both x_1 $Coin_1$ and x_2 $Coin_2$ if the other does not trigger **Redeem** or **Refund** on time. For example, if Bob does not trigger **Redeem** after Alice triggers **Redeem** and δ_1 expires, Alice can also refund x_1 $Coin_1$ by triggering **Refund**. It is Bob to blame in this case, because he should have had enough time - at least $\delta_2 - \delta_1$ (48 - 24 = 24 hours by default) - to redeem x_1 $Coin_1$. Another example is that Alice broadcasts $tx_{\mathcal{C},1}$ after δ_2, but Bob has already triggered **Refund**. Therefore, Bob can also redeem x_1 $Coin_1$ with s in $tx_{\mathcal{C},1}$ before δ_1. Similarly, it is Alice to blame in this case, because she should have had enough time - before δ_2 (24 hours by default) - to trigger **Redeem**.

8.3.2 American Call Option

The American Call Option is a contract that "one can buy an amount of an asset with an agreed price prior to or on an agreed time in the future". Here, the agreed price is usually called the *strike price*; and the contract settlement is called *exercising*; the one who proposes the contract and buys the asset is called the *option buyer*; the one who sells the asset is called the *option seller*.

As mentioned in Section 8.2.2, the option contract itself has value, called the *premium*. In an American Call Option, the option buyer should pay for the premium when the contract is agreed by both parties, and should pay for the asset when the contract is exercised.

We denote an American Call Option contract Π as

$$\Pi = (\pi_1, \pi_2, K, A, T, C)$$

where the option buyer Alice with π_1 hopes to buy π_2 from the option seller Bob; π_1 and π_2 are Alice's currency and Bob's asset, respectively; K is the strike price with the unit π_2/π_1 - the price of π_2 measured in π_1; A is the amount of the asset π_2 that Bob wants to sell; T is the agreed strike time; C is the *premium* with the unit π_1.

The process of an American Call Option is as follows:

1. **Advertise**: Alice creates and advertises an American Call Option contract $\Pi = (\pi_1, \pi_2, K, A, T, C)$.

2. **Contract**: If Bob believes Π is profitable and Alice does not abort Π, Bob will participate in Π. When Bob participates, Alice should pay C to Bob first. Note that Alice does not pay for A π_2 at this stage. Also note that Bob cannot abort Π after participating in Π.

3. **Exercise** or **Abort**: Alice exercises Π - pays AK π_1 to Bob - no later than T, and Bob gives A π_2 to Alice. If Alice does not exercise Π no later than T, Π will abort - Alice gets π_1 back and Bob gets π_2 back. In other words, both of them get their underlying asset back, but Alice loses the premium C to Bob when **Contract**.

8.3.3 An Atomic Swap is a premium-free American Call Option

We show that an Atomic Swap is equivalent to a premium-free American Call Option. More specifically, $\mathcal{AS} = (x_1, Coin_1, x_2, Coin_2)$ is equivalent to the American Call Option contract

$$\Pi = (Coin_1, Coin_2, \frac{x_2}{x_1}, x_2, \delta_2, 0)$$

where: **Advertise** in the American Call Option is equivalent to **Initiate** in the Atomic Swap; **Contract** in the American Call Option is equivalent to **Participate** in the Atomic Swap; **Exercise** in the American Call Option is equivalent to **Redeem** in the Atomic Swap; **Abort** in the American Call Option is equivalent to **Refund** in the Atomic Swap.

In the American Call Option context, the option buyer Alice wants to buy x_2 $Coin_2$ from the option seller Bob by using x_1 $Coin_1$. $Coin_1$ is the currency Alice uses, $Coin_2$ is the asset Bob has. This is equivalent to that Alice with π_1 wants to buy π_2 from Bob. δ_2 is the timelock of the contract transaction on BC_2, which is equivalent to the strike time T in Π. In $\mathcal{AS}$ Bob can refund his asset back after δ_2 to abort $\mathcal{AS}$, while Π will be automatically aborted after the strike time T. Establishing $\mathcal{AS}$ does not require Alice to pay anything other than x_1 $Coin_1$ to Bob, which is equivalent to Π with $C = 0$.

Note that both the Atomic Swap and the American Call Option are "speculative": Both the cryptocurrency exchange rates in Atomic Swaps and asset prices in the American Call Options are fluctuating overtime. Therefore, the "premium-free" property enables Alice to speculate without any risk: If Bob's asset price rises right before the strike time, she will proceed the swap to profit, otherwise she will abort the swap to avoid the loss. Therefore, without the premium, Alice is risk-free towards the market.

8.4 Unfairness of Atomic Swaps

In this section, we evaluate the unfairness of the Atomic Swap based on our observation in Section 8.3. In particular, our evaluations are from two perspectives: quantifying the unfairness and estimating the unpaid premium. Quantifying the unfairness is based on analysing the historical exchange rate volatility. Estimating the unpaid premium is based on the Cox-Ross-Rubinstein model - the conventional option pricing model for pricing American-style options in Finance. Furthermore, we also evaluate conventional financial assets - the stocks and the currency exchanges - and compare their results with cryptocurrencies. The code for the evaluation is available on Github **github-repo**.

8.4.1 Experimental setting

We collected relevant data of mainstream cryptocurrencies for one year, starting from May 3th, 2018 to May 3th, 2019. In particular, the cryptocurrency exchange rate data was retrieved from from CoinGecko [1]; the stock index data was retrieved from Yahoo Finance [2]; the currency exchange rate data was retrieved from Investing.com [3].

8.4.2 Quantifying the unfairness

Assume that Alice initiates the swap by triggering **Initiate**$(\cdot)$ at the time t, then by default $\delta_1 = t + 48(\text{hours})$. We also assume that Bob participates in the swap by triggering **Participate**$(\cdot)$, then by default $\delta_2 = t + 24(\text{hours})$.

In this way, Alice can decide whether to proceed the swap within $\delta_1 - \delta_2 = 24(\text{hours})$. When Bob's asset price rises, Alice profits directly. When Bob's asset price drops, Alice can abort the swap to avoid losing

[1] `https://www.coingecko.com`. Data was fetched at May 4th, 2019.
[2] `https://finance.yahoo.com`. Data was fetched at May 4th, 2019.
[3] `https://www.investing.com`. Data was fetched at May 4th, 2019.

money. Based on this observation, we classify Alice's advantages to two parts, namely the profit when Bob's asset price rises and the mitigated loss when Bob's asset price drops.

We then test the unfairness by using a single Atomic Swap with the value of x USD, then we show the degree of unfairness in dollars based on the historical data. For each day, Alice may either profit α percent of x directly (when Bob's asset price rises), or mitigate β percent of x by aborting the swap (when Bob's asset price drops) on average. Assume the possibility for Bob's asset price to rise is P_α, and to drop is P_β. Then, the expected profit rate is $E_\alpha = \alpha P_\alpha$, and the expected mitigated risk rate is $E_\beta = \beta P_\beta$. Therefore, the expected unfairness is that Alice profits $E_\alpha x$ and mitigates the risk of losing $E_\beta x$. Also, as E_α and E_β are equally calculated, adding up them together ($E_\alpha + E_\beta$) can derive the total unfairness.

Experimental methodology. In our scenario, quantifying the unfairness is to calculate E_α and E_β, so we calculate E_α and E_β for each selected cryptocurrency pair. Furthermore, we also quantify the unfairness of stock indices and fiat currencies in the same setting, in order to make comparisons. We use S&P500 and Dow Jones Index (DJI) as examples of stock indices, and USD-EUR and USD-GBP as examples of fiat currencies.

Results and analysis. Figure 8.3 shows the calculated E_α, E_β, the maximum daily rises max_α and the maximum daily drops max_β for 8 mainstream cryptocurrency pairs, stock indices (S&P500 and DJI) and fiat currency exchange rates (USD-EUR and USD-GBP). For each plot, points in the red Profit Area indicate that Alice profits directly at those days, and points in the green Risk Area indicate that Alice can abort the swap to mitigate the risk at those days.

We observe that for all chosen cryptocurrency pairs, max_α and max_β are considerably big - ranging from 8% to 25%. Meanwhile, max_α and max_β of stock indices are much smaller than all cryptocurrency pairs, and max_α and

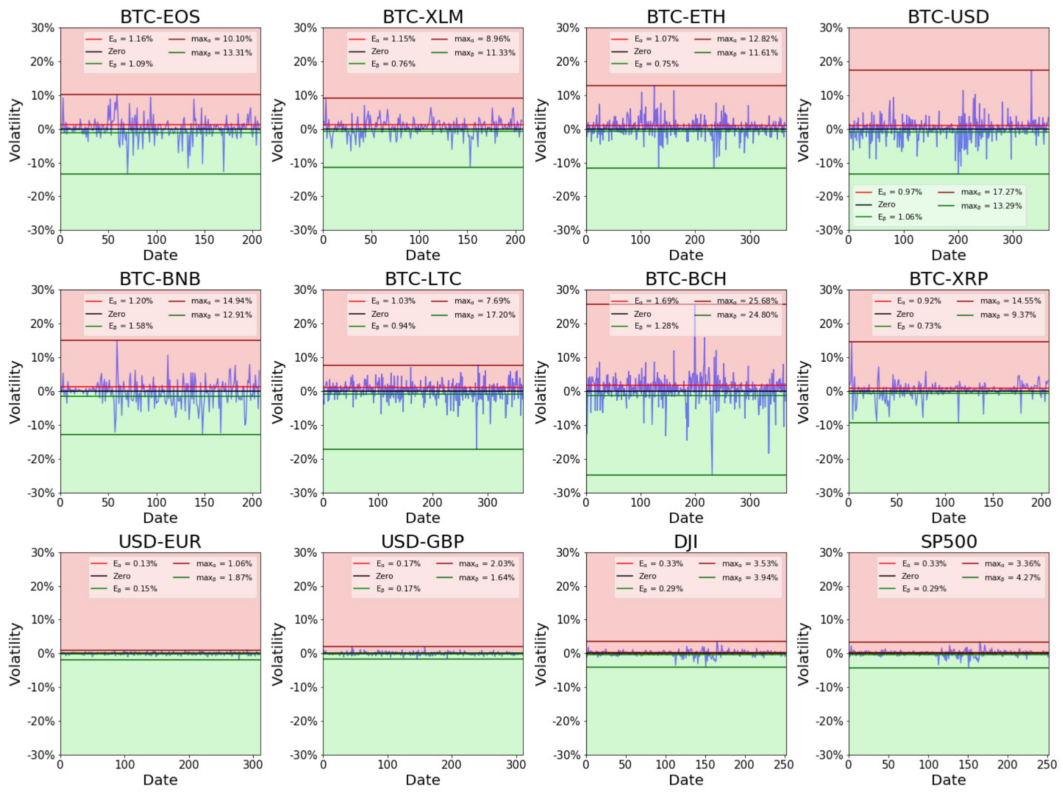


Figure 8.3: The daily percentage changes for all selected cryptocurrency pairs, stock indices and fiat currency pairs over one year (from 03/05/2018 to 03/05/2019). For each figure, E_α, E_β, max_α and max_β are the expected profit rate, the expected mitigated risk rate, the maximum daily profit and the maximum daily mitigated risk, respectively. The red area is the Profit Area where Alice profit from the rising asset price, and the green area is the Risk Area where Alice mitigates the loss from the dropping asset price.

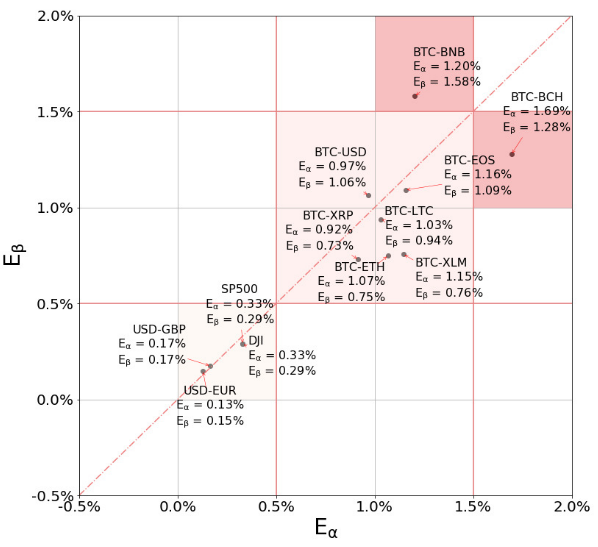


Figure 8.4: The expected profit rate E_α and the expected mitigated risk rate E_β for each cryptocurrency pair, stock index and fiat currency pair.

max_β of fiat currencies are even smaller than stock indices. This indicates that in the setting of an 24-hour Atomic Swap, the Atomic Swap with cryptocurrencies is much more unfair than with stocks, and the Atomic Swap with stocks are more unfair than with fiat currencies.

Figure 8.4 visualises E_α and E_β of all evaluated items in Figure 8.3. In particular, we classify scatters to 4 groups based on their E_α and E_β: The first group ($0 < E_\alpha < 0.005 \wedge 0 < E_\beta < 0.005$) consists of all stock indices (S&P500 and DJI) and all fiat currency pairs (USD-GBP and USD-EUR); the second group ($0.005 < E_\alpha < 0.015 \wedge 0.005 < E_\beta < 0.015$) consists of most cryptocurrency pairs; the third group ($0.010 < E_\alpha < 0.015 \wedge E_\beta > 0.015$) only contains one cryptocurrency pair BTC-BNB; the fourth group ($E_\alpha > 0.015 \wedge 0.010 < E_\beta < 0.015$) only contains the last cryptocurrency pair BTC-BCH. Moreover, we draw a line $E_\beta = E_\alpha$ to separate two areas: $E_\beta > E_\alpha$ and $E_\beta < E_\alpha$.

Obviously, the Atomic Swap with first-group items is fairer than with second-group items, and the Atomic Swap with second-group items is fair than with third-group and fourth-group items. More specifically, we can get the following results. First, the Atomic Swap with cryptocurrency pairs is more unfair than with stocks and fiat currency pairs. This result is consistent with results in Figure 8.3. Second, E_β and $E_\beta - E_\alpha$ of BTC-BNB are bigger than of others. This means the exchange rate of BTC-BNB, and drops generally over the last year. Meanwhile, E_α and $E_\alpha - E_\beta$ of BTC-BCH are bigger than of others. This means the exchange rate of BTC-BCH, and rises generally over the last year. Both observations indicate that BTC-BNB and BTC-BCH are highly volatile, so the Atomic Swap with those assets is more unfair than with other assets. Third, all dots are close to the line $E_\beta = E_\alpha$, while the dots of stock indices and fiat currency pairs almost lay on this line. A dot lying on $E_\beta = E_\alpha$ means the exchange rate rises and drops at the same level. Therefore, although more volatile than stocks and fiat currencies, exchange rates of cryptocurrency pairs rise and drop at the same level.

8.4.3 Estimating the premium

The unfairness of the Atomic Swap comes from the fact that Alice can abort the contract without punishment. In Finance, the premium mechanism guarantees the good behaviours. As the Atomic Swap is equivalent to the premium-free American Call Option, the Cox-Ross-Rubinstein (CRR) Model [16] can be used for estimating the premium of Atomic Swaps.

Therefore, we can evaluate the unfairness of the Atomic Swap by estimating the premium for American Call Options with cryptocurrencies.

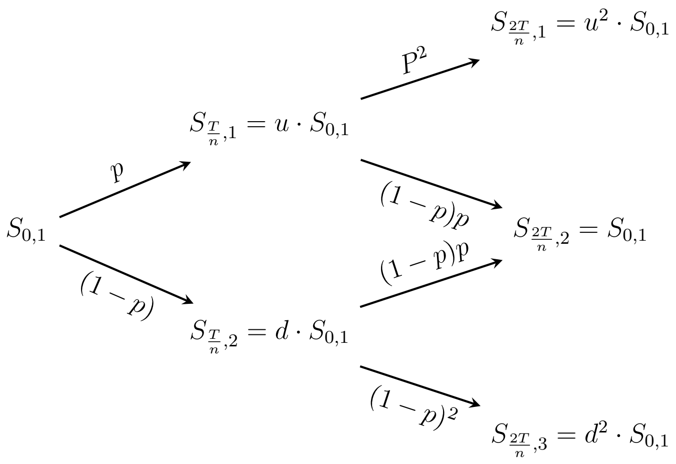
As the premium is the only variable in an option contract, estimating the premium is also called the "Option Pricing" problem. In Finance, the Black-Scholes (BS) Model [286] is utilised to price the European Call Options, while the CRR model is utilised to price the American Call Options.

Table 8.1: Summary of symbols in the Cox-Ross-Rubinstein Model.

Variable	Description	Comment
u, d	The rising and dropping rates for prices in the binomial tree $\mathcal{T}$	$u \cdot d = 1$
σ_d, σ_a	The daily and annualised percentage change rates of asset prices	
T	The strike time (measured in years)	
n	The depth of $\mathcal{T}$	we pick $n = 36$
Δt	The time period between two adjacent nodes on $\mathcal{T}$ (measured in years)	$\Delta_t = \frac{T}{n}$
$S_{t,i}$	The asset price of the i-th node on the $\frac{t}{\Delta t}$-th level of $\mathcal{T}$	
$C_{t,i}$	The premium of the i-th node on the $\frac{t}{\Delta t}$-th level of $\mathcal{T}$	
p, q	The probabilities that the asset price rises and drops	

Therefore, in order to evaluate the unfairness of the Atomic Swap, we use the CRR Model to estimate how much the premium should be in the Atomic Swap.

The Cox-Ross-Rubinstein Model Explained



$S_{\frac{2T}{n},1} = u^2 \cdot S_{0,1}$
p^2
$S_{\frac{T}{n},1} = u \cdot S_{0,1}$
p
$(1-p)p$
$S_{0,1}$
$S_{\frac{2T}{n},2} = S_{0,1}$
$(1-p)p$
$(1-p)$
$S_{\frac{T}{n},2} = d \cdot S_{0,1}$
$(1-p)^2$
$S_{\frac{2T}{n},3} = d^2 \cdot S_{0,1}$

Figure 8.5: The binomial price tree $\mathcal{T}$.

The Cox-Ross-Rubinstein (CRR) Model [16] - a.k.a. the Binomial Options Pricing Model (BOPM) - is a numerical method for pricing American-style Options. Intuitively, the CRR model enumerates all possible asset prices of the asset in the near future based on the price volatility, then reverse-engineers the premium based on the enumerated asset prices. More specifically, using the CRR model to price the American Call Option $\Pi = (\pi_1, \pi_2, K, A, T, C)$ follows the steps below:

1. Creating the binomial price tree
2. Calculating the premiums for leaf nodes
3. Iteratively reconstructing the premiums for non-leaf nodes

Creating the binomial price tree. The binomial price tree $\mathcal{T}$ of the height n (as shown in Fig. 8.5) represents the possible future prices within the time period T discretely. n can be picked arbitrarily: With larger n, the result will be more accurate, but the computing overhead will be heavier. Each node $\mathcal{T}_{t,i}$ is attached with an asset price $S_{t,i}$ and a premium price $C_{t,i}$, where $t \in \{0, \frac{T}{n}, \frac{2T}{n}, \ldots, T\}$ is the point of time, and i is the id of this node at its level. The CRR model assumes that the asset price will either move up or down by a specific factor per step in $\mathcal{T}$. The move-up factor is u, and the move-down factor is d. For example, given the initial asset price $S_{0,1}$, the asset price after one move-up $S_{\frac{T}{n},1}$ is $u \cdot S_{0,1}$, and the asset price after one move-down $S_{\frac{T}{n},2}$ is $d \cdot S_{0,1}$.

u and d are calculated using the annualised volatility σ_a of the underlying asset price. In the CRR model, the move-up and move-down are symmetric - $u \cdot d = 1$, and the rate of move-up and move-down is positive correlated with σ_a:

$$u = e^{\sigma_a \sqrt{\frac{T}{n}}} \tag{8.1}$$

$$d = e^{-\sigma_a \sqrt{\frac{T}{n}}} = \frac{1}{u} \tag{8.2}$$

Here, T is measured in years, and σ_a is defined as the standard deviation of the annual price changes in percentage. σ_a can be computed from the standard deviation σ_d of daily price changes in percentage as below:

$$\sigma_a = \sigma_d \sqrt{d} \tag{8.3}$$

$$\sigma_d = \sqrt{\frac{\sum_{i=1}^{d}(S'_i - \bar{S}')^2}{d-1}} \tag{8.4}$$

where d is the number of trading days within a year. For cryptocurrencies, d equals to the number of a days within a year. Note that S'_i is the percentage change of the price on day i, rather than the price itself. $\bar{S}'$ is the average value of all S'_is within the d days.

Each asset price $S_{t,i}$ can be calculated directly by $S_{t,i} = S_{0,1} \cdot u^{N_u - N_d}$, where $S_{0,1}$ is the initial asset price, and N_u, N_d are the times of move-ups and move-downs, respectively.

Calculating the premiums for leaf nodes. In the first step, only the asset prices are determined rather than the premiums. This step further determines the premiums for leaf nodes. For each leaf node $\mathcal{T}_{n,i}$, the premium is $C_{n,i} = max[(S_{n,i} - K), 0]$.

Iteratively reconstructing the premiums for earlier nodes. We back-propagate the premiums for leaf nodes to earlier premiums. Each earlier premium is calculated from premiums of the later two nodes weighted by their possibilities. The move-up and move-down possibility are p and q where $p + q = 1$, and the risk-free rate is $r = q$. More specifically, each earlier premium $C_{t-\Delta t,i}$ is calculated from later premiums as:

$$C_{t-\Delta t,i} = e^{-r\Delta t}(pC_{t,i} + qC_{t,i+1}) \tag{8.5}$$

where $\Delta t = \frac{T}{n}$, and p, q, r are computed as

$$p = \frac{e^{(r-q)\Delta t} - d}{u - d} \tag{8.6}$$

$$q = 1 - p \tag{8.7}$$

$$r = q \tag{8.8}$$

such that the premium distribution simulates the geometric Brownian motion [287] with parameters r and σ.

In this way, the earliest premium $C_{0,1}$, which is our targeted estimated premium C - can be calculated by iteratively back-propagating the later premiums.

Table 8.1 summarises all symbols used in the Cox-Ross-Rubinstein model.

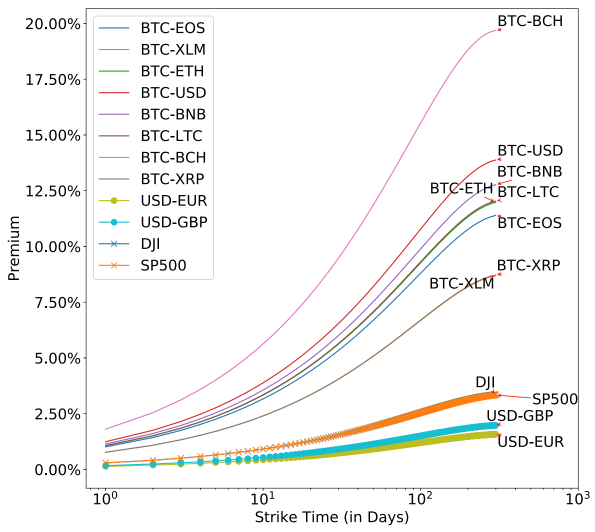


Figure 8.6: Estimated premium with different strike times for each cryptocurrency pair, stock index and fiat currency pair. Lines with the marker "x" are for stock indices; lines with the marker "o" are for fiat currencies; and lines without marker are for cryptocurrency pairs.

Experiments. We use the same data as Section 8.4.2, and choose $n = 36$ for the CRR model. We estimate the premium for the same assets (8 cryptocurrency pairs + 2 stock indices + 2 fiat currency pairs in Section 8.4.2) with the strike time T ranging from 1 to 300. Figure 8.6 shows our pricing results.

First, we observe that the premium of cryptocurrency pairs is much more expensive than of stocks, and the premium of stocks is more expensive than of fiat currencies at any given time. Recall the evaluated unfairness in Section 8.4.2, its results are consistent with the premium pricing results: The more volatile the market is, the more unfair the Atomic Swap will be, and the higher the premium should be. Second, with the default strike time $T = 1$ of the Atomic Swap, the premium for cryptocurrency pairs vary from approximately 1% to 2.3% of the underlying asset value, but the values for stocks and fiat currency pairs are approximately 0.3%. Third, for all evaluated items, the premium values rise monotonically with T increasing. This is because the longer expiration time lets Alice to have more control on the option - he has more time to predict the price and decide to exercise or abort the option.

8.5 Fair Atomic Swaps

In this section, we propose an improvement on the original Atomic Swap to make it fair. It implements the premium mechanism, and supports both the currency exchange-style Atomic Swap and the American Call Option-style Atomic Swap.

8.5.1 Design

Difference between Currency Exchange and Options

We first summarise the design objectives for Atomic Swap.

To our knowledge, the Atomic Swap protocol is originally designed for the fair exchange between different cryptocurrencies. However, according to our analysis, the protocol is unfair due to the Optionality and the free premium. Also, for (crypto)currency exchange, the protocol should have no Optionality. The currency exchange and the American Call Option differ in Finance: The currency exchange is a type of Spots [288], while the American Call Option is a type of Options. The Spot Contract and the Option Contract aim at different application scenarios: The Spot Contract aims at exchanging the ownership of assets, while the Option Contract aims at providing Alice an "option" to trade. More specifically, Spots and Options differ in the following aspects:

- The Spot Contract is exercised immediately, while the Option Contract is exercised on or prior to a specified date in the future.
- The Spot Contract cannot be aborted once signed by both parties, while in the Option Contract Alice can abort the contract with the loss of the premium.
- The Spot Contract itself has no value, while the Option Contract itself has value - the premium.

Premium for Currency Exchange and American Call Options

According to Section 8.5.1, the currency exchange-style Atomic Swaps and the American Call Option-style Atomic Swaps differ in design objectives.

Atomic Swaps for Currency Exchange. For the currency exchange, both parties are not permitted to abort the contract once signed. However, in Atomic Swaps, Alice can abort the swap by not releasing the random secret. Therefore, the protocol should discourage Alice to abort the swap. To achieve this, we can use the premium mechanism as the collateral: Alice should deposit the premium besides her asset when **Initiate**. The premium should follow that: **Alice pays the premium to Bob if Bob refunds his asset**

after his timelock but before Alice's timelock. If Alice's timelock expires, Alice can refund her premium back.

Atomic Swaps for American Call Options. For the American Call Options, Alice should pay for the premium besides the underlying asset, regardless of whether the swap is successful or not. In reality, the option sellers are trustworthy - the option sellers never abort the contract. However, in Atomic Swaps, Bob can abort the contracts like Alice. To keep the Atomic Swap consistent with the American Call Options, the premium should follow that: **Alice pays the premium to Bob if 1) Alice redeems Bob's asset before Bob's timelock, or 2) Bob refunds his asset after Bob's timelock but before Alice's timelock. If Alice's timelock expires, Alice can refund her premium back.**

8.5.2 Our protocol

We propose an improvement on the Atomic Swap, which implements the premium mechanism, to make it fair. It can fulfill design objectives of both the currency exchange and the American Call Option. Figure 8.7) shows the process of our Atomic Swap.

We denote our Atomic Swap protocol $\mathcal{AS}'$ as

$$\mathcal{AS}' = (x_1, Coin_1, x_2, Coin_2, pr)$$

where pr is the amount of the premium measured in $Coin_2$. In our protocol, besides x_1 $Coin_1$, Alice should also lock pr $Coin_2$ on BC_2, which will be described later.

Similar to the original Atomic Swap $\mathcal{AS}$, our protocol consists of four stages: **Initiate**, **Participate**, **Redeem** and **Refund**.

Initiate. Different from $\mathcal{AS}$, Alice also creates Bob's contract script $\mathcal{C}_2$ and its associated transaction $tx_{\mathcal{C},2}$ when **Initiate** in $\mathcal{AS}'$.

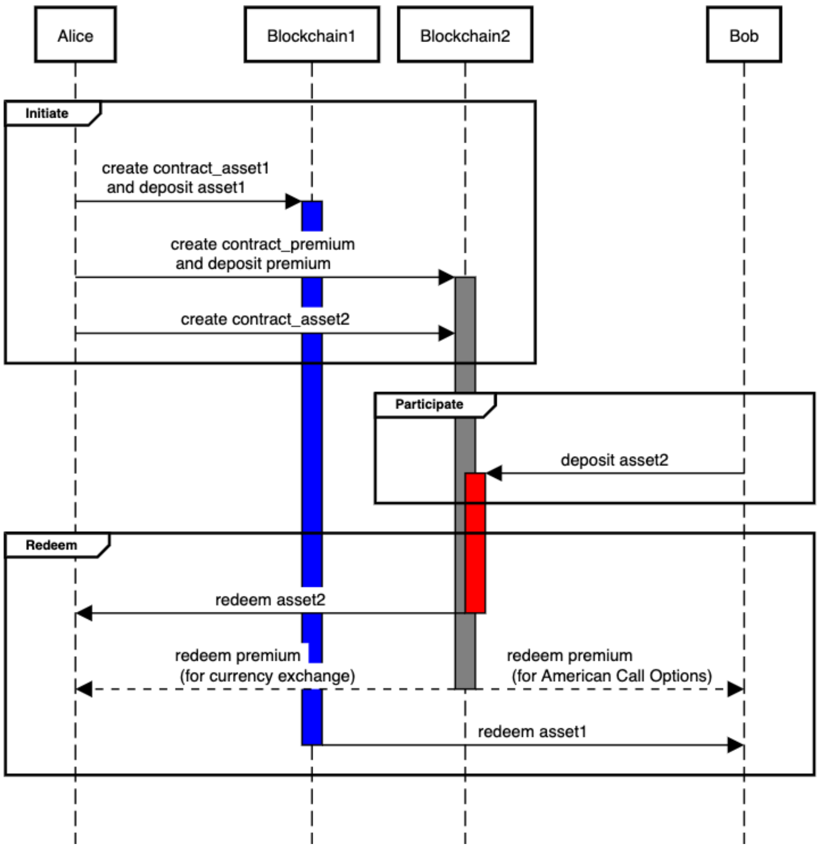

Alice
Blockchain1
Blockchain2
Bob
Initiate
create contract_asset1 and deposit asset1
create contract_premium and deposit premium
create contract_asset2
Participate
deposit asset2
Redeem
redeem asset2
redeem premium (for currency exchange)
redeem premium (for American Call Options)
redeem asset1

Figure 8.7: Sequence diagram of our Atomic Swap. For currency exchange-style Atomic Swaps, the premium will go back to Alice if the swap is successful (the left dotted line). For American Call Option-style Atomic Swaps, the premium will go to Bob if the swap is successful (the right dotted line).

$\mathcal{C}_1$ and $tx_{\mathcal{C},1}$ is the same as in $\mathcal{AS}$, while $\mathcal{C}_2$ and $tx_{\mathcal{C},2}$ are more sophisticated. $\mathcal{C}_2$ contains two coherent sub-contracts $\mathcal{C}_2^{asset}$ and $\mathcal{C}_2^{pr}$.

In $\mathcal{C}_2$, $\mathcal{C}_2^{asset}$ is the contract for the asset x_2 $Coin_2$, which is the same as in $\mathcal{AS}$. $\mathcal{C}_2^{pr}$ is the contract for the premium pr, which implements the premium mechanism in the Atomic Swap. It supports both the currency exchange-style Atomic Swap and the American Call Option-style Atomic Swap. In more detail, the rules of $\mathcal{C}_2^{pr}$ are shown below:

$\mathcal{C}_2^{pr}$ **for currency exchange** Alice pays pr to Bob with the condition: Bob refunds x_2 $Coin_2$ after δ_2 and before δ_1. If δ_1 expires, Alice can refund pr back.

$\mathcal{C}_2^{pr}$ **for American Call Options** Alice pays pr to Bob with one of the two conditions: 1) Alice redeems x_2 $Coin_2$ before δ_2. 2) Bob refunds x_2 $Coin_2$ after δ_2 but before $Delta_1$ (note that $\delta_2 < \delta_1$). If δ_1 expires, Alice can refund pr back.

Alice published $tx_{\mathcal{C},1}$ on BC_1 and $tx_{\mathcal{C},2}$ on BC_2. Note that Alice only triggers $\mathcal{C}_1$ and $\mathcal{C}_2^{pr}$ to execute at this stage. Bob will deposit x_2 $Coin_2$ trigger $\mathcal{C}_2^{asset}$ to execute when **Participate**.

Participate. Bob decides whether to participate in $\mathcal{AS}'$ by auditing $tx_{\mathcal{C},1}$ and $tx_{\mathcal{C},2}$. If Bob thinks contracts are fair, he will participate in $\mathcal{AS}'$, otherwise Bob will not participate and look for more profitable contracts from others. To participate in $\mathcal{AS}'$, Bob deposits x_2 $Coin_2$ in $\mathcal{C}_2^{asset}$, and triggers $\mathcal{C}_2^{asset}$ to execute.

Redeem. Alice redeeming x_2 $Coin_2$ and Bob redeeming x_1 $Coin_1$ are the same in $\mathcal{AS}'$ and $\mathcal{AS}$. But in addition, riles in $\mathcal{C}_2^{pr}$ will work once triggering **Redeem** for $\mathcal{AS}'$.

Refund. Refunding x_1 $Coin_1$ for Alice and x_2 $Coin_2$ for Bob are the same as in $\mathcal{AS}$. But in addition, rules in $\mathcal{C}_2^{pr}$ will work once triggering **Refund** for $\mathcal{AS}'$.

8.6 Implementation

In this section, we describe how to implement our Fair Atomic Swap protocol in Section 8.5 on different blockchains. In particular, blockchains with smart contracts (such as Ethereum) can support our protocol directly, while blockchains with scripts only (such as Bitcoin) require an extra opcode which we call OP_LOOKUP_OUTPUT. In addition, we provide reference implementations in Bitcoin scripts and Solidity smart contracts, and the Solidity implementation is available on Github **github-repo**.

8.6.1 Requirements

To implement our protocol, the blockchain should support 1) stateful transactions, 2) the timelock and 3) the hashlock.

Stateful transactions. Transactions should be stateful: Executing a transaction can depend on prior transactions. In our protocol, whether pr goes to Alice or Bob depends on the status of $Coin_2$. Therefore, the transaction of pr relies on the status of $Coin_2$ payment transaction.

Hashlock. The transactions should support the hashlock: A payment is proceeded only when the payee provides the preimage of a hash. In our protocol, exchanging $Coin_1$ and $Coin_2$ atomically is based on the hashlock - Alice redeems $Coin_2$ first by releasing the preimage, then Bob can redeem $Coin_1$ by using the released preimage.

Timelock. The transactions should support the timelock: A payment will expire after a specified time if the payee cannot redeem the payment. In our protocol, the transactions of $Coin_1$, $Coin_2$ and pr are all timelocked, in order to avoid locking money in transactions forever.

8.6.2 Smart contracts

Smart contracts support all aforementioned functionalities, so can easily implement our protocol. We use Solidity - one of programming languages for Ethereum smart contracts [39] - as an example. Our implementations are based on the original Atomic Swap Solidity implementation [289], but extend the premium mechanism. Extending the premium mechanism includes:

1. The enumeration *PremiumState* for maintaining the premium payment state
2. The modifiers *isPremiumRedeemable()* and *isPremiumRefundable()* for checking whether the premium can be redeemed or refunded
3. The methods *redeemPremium()* and *refundPremium()* for redeeming and refunding the premium

```
enum AssetState { Empty, Filled , Redeemed, Refunded }
enum PremiumState { Empty, Filled , Redeemed, Refunded }
```

Listing 8.1: Maintaining the state of the asset and the premium.

The premium payment state *PremiumState*. In the original smart contract, an enumeration *State* maintains the asset state: **empty** means the asset has not been deposited; **filled** means the asset has been deposited; **redeemed** means the asset has been redeemed; **refunded** means the asset has been refunded.

In our contract, we decouple *State* to the asset state *AssetState* and the premium state *PremiumState*. Both *AssetState* and *PremiumState* are the same as the original *State*. The code is shown in Listing 8.1. *Empty* means Alice has not triggered **Initiate**, and has not deposited the premium yet. *Filled* means Alice has deposited the premium, indicating that Alice has triggered

```
// Premium is refundable when
// 1. Alice initiates but Bob does not participate
//    after premium's timelock expires
// 2. asset2 is redeemed by Alice
modifier isPremiumRefundable(bytes32 secretHash) {
    // the premium should be deposited
    require(swaps[secretHash].premiumState == PremiumState.Filled);
    // Alice invokes this method to refund the premium
    require(swaps[secretHash].initiator == msg.sender);
    // the contract should be on the blockchain2
    require(swaps[secretHash].kind == Kind.Participant);
    // if the asset2 timelock is still valid
    if block.timestamp <= swaps[secretHash].assetRefundTimestamp {
        // the asset2 should be redeemded by Alice
        require(swaps[secretHash].assetState == AssetState.Redeemed);
    } else { // if the asset2 timelock is expired
        // the asset2 should not be refunded
        require(swaps[secretHash].assetState != AssetState.Refunded);
        // the premium timelock should be expired
        require(block.timestamp > swaps[secretHash].premiumRefundTimestamp);
    }
    -;
}
// Premium is redeemable for Bob when asset2 is refunded
// which means Alice holds the secret maliciously
modifier isPremiumRedeemable(bytes32 secretHash) {
    // the premium should be deposited
    require(swaps[secretHash].premiumState == PremiumState.Filled);
    // Bob invokes this method to redeem the premium
    require(swaps[secretHash].participant == msg.sender);
    // the contract should be on the blockchain2
    require(swaps[secretHash].kind == Kind.Participant);
    // the asset2 should be refunded
    // this also indicates the asset2 timelock is expired
    require(swaps[secretHash].assetState == AssetState.Refunded);
    // the premium timelock should not be expired
    require(block.timestamp <= swaps[secretHash].premiumRefundTimestamp);
    -;
}
```
Listing 8.2: The condition to redeem and refund the premium for currency-exchange-style Atomic Swaps.

```
// Premium is refundable for Alice only when Alice initiates
// but Bob does not participate after premium's timelock expires
modifier isPremiumRefundable(bytes32 secretHash) {
    // the premium should be deposited
    require(swaps[secretHash].premiumState == PremiumState.Filled)
        ;
    // Alice invokes this method to refund the premium
    require(swaps[secretHash].initiator == msg.sender);
    // the contract should be on the blockchain2
    require(swaps[secretHash].kind == Kind.Participant);
    // premium timelock should be expired
    require(block.timestamp > swaps[secretHash].
        premiumRefundTimestamp);
    // asset2 should be empty
    // which means Bob does not participate
    require(swaps[secretHash].assetState == AssetState.Empty);
}
// Premium is redeemable for Bob when asset2 is redeemed or
    refunded
// which means Bob participates
modifier isPremiumRedeemable(bytes32 secretHash) {
    // the premium should be deposited
    require(swaps[secretHash].premiumState == PremiumState.Filled)
        ;
    // Bob invokes this method to redeem the premium
    require(swaps[secretHash].participant == msg.sender);
    // the contract should be on the blockchain2
    require(swaps[secretHash].kind == Kind.Participant);
    // the asset2 should be refunded or redeemed
    require(swaps[secretHash].assetState == AssetState.Refunded ||
         swaps[secretHash].assetState == AssetState.Redeemed);
    // the premium timelock should not be expired
    require(block.timestamp <= swaps[secretHash].
        premiumRefundTimestamp);
    _;
}
```

Listing 8.3: The condition to redeem and refund the premium for American Call Option-style Atomic Swaps.

Initiate, but neither Alice nor Bob refunds or redeems the premium. *Redeemded* and *refunded* means Bob redeems the premium and Alice refunds the premium, respectively.

***isPremiumRedeemable()* and *isPremiumRefundable()*.** Checking whether the premium is redeemable or refundable is the most critical part of our protocol. Because the premium payment relies on the $Coin_2$ payment, checking the premium refundability and redeemability involves checking the $Coin_2$ status - *AssetState* in our implementation.

isPremiumRedeemable() and *isPremiumRefundable()* for the currency exchange-style Atomic Swap are shown in Figure 8.2, and for the American Call Option-style Atomic Swap are shown in Figure 8.3. The currency exchange-style Atomic Swap and the American Call Option-style Atomic Swap differ when $AssetState = Redeemed$: In the currency exchange-style Atomic Swap the premium belongs to Alice while in the American Call Option-style Atomic Swap the premium belongs to Bob.

***redeemPremium()* and *refundPremium()*.** *redeemPremium()* and *refundPremium()* are similar to *redeemAsset()* and *refundAsset()*, and their executions are secured by *isPremiumRedeemable()* and *isPremiumRefundable()*. The code is shown in Listing 8.4.

8.6.3 Bitcoin script

Unfortunately, Bitcoin cannot support our protocol directly, because Bitcoin does not support the stateful transaction functionalities. First, the Bitcoin script is designed to be stateless [290]. Second, there is no such things like the Ethereum's "world state" [39] in Bitcoin: The only state in Bitcoin is the Unspent Transaction Outputs (UTXOs) [1].

New Opcode OP_LOOKUP_OUTPUT. In order to make Bitcoin script support our protocol, we use an opcode called OP_LOOKUP_OUTPUT. OP_LOOKUP_OUTPUT was proposed, but has not implemented in Bitcoin

```
function redeemPremium(bytes32 secretHash)
    public
    isPremiumRedeemable(secretHash)
{
    // transfer the premium to Bob
    swaps[secretHash].participant.transfer(swaps[secretHash].
        premiumValue);
    // update the premium state to redeemded
    swaps[secretHash].premiumState = PremiumState.Redeemed;
    // notify the function invoker
    emit PremiumRedeemed(
        block.timestamp,
        swaps[secretHash].secretHash,
        msg.sender,
        swaps[secretHash].premiumValue
    );
}
function refundPremium(bytes32 secretHash)
    public
    isPremiumRefundable(secretHash)
{
    // transfer the premium to Alice
    swaps[secretHash].initiator.transfer(swaps[secretHash].
        premiumValue);
    // update the premium state to refunded
    swaps[secretHash].premiumState = PremiumState.Refunded;
    // notify the function invoker
    emit PremiumRefunded(
        block.timestamp,
        swaps[secretHash].secretHash,
        msg.sender,
        swaps[secretHash].premiumValue
    );
}
```

Listing 8.4: The functions for redeeming and refunding the premium.

yet [291]. It takes the id of an output, and produces the address of the output's owner. With OP_LOOKUP_OUTPUT, the Bitcoin script can decide whether Alice or Bob should take the premium by "¡asset2_output¿ OP_LOOKUP_OUTPUT ¡Alice_pubkeyhash¿ OP_EQUALVERIFY".

Implementing OP_LOOKUP_OUTPUT is easy in Bitcoin - it only queries the ownership of an output from the indexed blockchain database. This neither introduces computation overhead, nor breaks the "stateless" design of the Bitcoin script.

Decoupling the contract creation and the contract invocation. For smart contracts, the contract is created and invoked in separate transactions: Creating the contract is by publishing a transaction which creates the smart contract, and invoking the contract is by publishing a transaction which invokes a method in the smart contract. However, Bitcoin has no smart contracts, and the "contract" is created and invoked in a single transaction. In this way, the timelock starts right after the contract creation rather than the contract invocation. This is problematic: The premium contract should not be triggered until Bob participates in the swap.

Thanks to the multi-signature transaction functionality in Bitcoin, Alice and Bob can first create the contract off-chain, then invoke the contract on-chain.

Multi-signature transactions refer to transactions signed by multiple accounts [290]. A M-of-N ($M \leq N$) multi-signature transaction means the transaction requires M out of N accounts to sign it. If less than M accounts sign the transaction, the transaction cannot be verified as valid by the blockchain. In Bitcoin, constructing a multi-signature transaction requires accounts to create a multi-signature address first [290].

With multi-signature transactions, we can decouple the contract creation and invocation as follows: first, Alice and Bob create a 2-of-2 multi-signature address; second, Alice and Bob mutually construct and sign a transaction which includes the premium payment and the $Coin_2$ payment; finally, they publish the transaction in the name of the 2-2 multi-signature address.

Note that constructing and signing the transaction is done off-chain: first, Bob creates the $Coin_2$ transaction and sends it to Alice; second, Alice creates the premium transaction which uses OP_LOOKUP_OUTPUT to check the ownership of $Coin_2$ transaction outputs; third, Alice merges the $Coin_2$ transaction and the premium transaction to a single transaction, signs the transaction, and sends it to Bob; finally, Bob signs the transaction and sends it to Alice. At this stage, both Alice and Bob have obtained the mutually signed transaction, which consists of both the premium transaction and the $Coin_2$ transaction.

The premium transaction. Listing 8.5 and Listing 8.6 show the premium transaction in the Bitcoin script , for both the currency-style and the American Call Option-style Atomic Swaps, respectively.

```
ScriptSig:
    Redeem: <Bob_sig> <Bob_pubkey> 1
    Refund: <Alice_sig> <Alice_pubkey> 0
ScriptPubKey:
    OP_IF // Normal redeem path
        // the owner of <asset2_output> should be Alice
        // which means Alice has redeemed asset2
        <asset2_output> OP_LOOKUP_OUTPUT <Alice_pubkeyhash>
            OP_EQUALVERIFY
        OP_DUP OP_HASH160 <Bob_pubkeyhash>
    OP_ELSE // Refund path
        // the premium timelock should be expired
        <locktime> OP_CHECKLOCKTIMEVERIFY OP_DROP
        OP_DUP OP_HASH160 <Alice pubkey hash>
    OP_ENDIF
    OP_EQUALVERIFY
    OP_CHECKSIG
```

Listing 8.5: The currency exchange-style Atomic Swap contract in Bitcoin script.

```
ScriptSig:
    Redeem: <Bob_sig> <Bob_pubkey> 1
    Refund: <Alice_sig> <Alice_pubkey> 0
ScriptPubKey:
    OP_IF // Normal redeem path
        // the owner of the asset2 should not be the contract
        // it should be either (redeemde by) Alice or (refunded by
            ) Bob
        // which means Alice has redeemed asset2
        <asset2_output> OP_LOOKUP_OUTPUT <Alice_pubkeyhash>
            OP_NUMEQUAL
        <asset2_output> OP_LOOKUP_OUTPUT <Bob_pubkeyhash>
            OP_NUMEQUAL
        OP_ADD 1 OP_NUMEQUALVERIFY
        OP_DUP OP_HASH160 <Bob_pubkeyhash>
    OP_ELSE // Refund path
        // the premium timelock should be expired
        <locktime> OP_CHECKLOCKTIMEVERIFY OP_DROP
        OP_DUP OP_HASH160 <Alice pubkey hash>
    OP_ENDIF
    OP_EQUALVERIFY
    OP_CHECKSIG
```

Listing 8.6: The American Call Option-style Atomic Swap contract in Bitcoin script.

8.7 Discussion

8.7.1 Security of the Atomic Swap

Although already widely adopted, the Atomic Swap has security issues.

First, the security of Atomic Swaps relies on the security of blockchains: If the blockchains involved in the swaps are insecure, the Atomic Swaps will also be insecure.

Second, the Atomic Swap contracts are written in high-level languages, so the compiled contracts can be insecure if the contract compilers are flawed.

Third, the timelock is unreliable in the cross-chain scenario. Similar to other distributed systems [292], different blockchains are unsynchronised on the time. Blockchains timestamp events by either two approaches: Using the block height or using the UNIX timestamp. The block height can serialise events on a blockchain by time, but cannot serialise events outside the blockchain. In addition, the new block generation is a random process, so the block height cannot indicate the precise time in reality. Using the UNIX timestamp doesn't work, either. This is because the consensus participants are responsible for timestamping events, but the consensus participants can be unreliable: They may use the wrong time, either on purpose or by accident.

8.7.2 Other countermeasures

Besides our proposal, there are some other countermeasures to address the Atomic Swap unfairness. Unfortunately, to our knowledge, all of them either have security flaws or significantly reduce the usability of Atomic Swaps.

The first countermeasure is to make the Atomic Swap costly by charging setting up HTLCs, or increasing the transaction fee of HTLCs. However, these two solutions do not only significantly reduce the usability of Atomic Swaps, but also affect HTLCs not aiming at setting up Atomic Swaps.

The second solution is to use shorter timelock for Atomic Swaps. Unfortunately, short timelocks may cause unexpected consequences. Confirming transactions for setting up Atomic Swaps takes time, and the time required is highly unpredictable. With short timelocks, the transactions for setting up Atomic Swaps may be confirmed after the expiration of timelocks.

The third solution is using a trusted third party (TTP) to implement the premium mechanism. When Alice initiates an Atomic Swap, the TTP forces Alice to deposit the premium. Although this TTP does not require Alice and Bob to escrow their assets, the TTP should be trustworthy and can be a single point of failure.

8.7.3 Limitations of our protocols

Still, our solutions are not perfect. The initiators of Atomic Swaps need to hold some participant's asset to initiate an Atomic Swap, for either collateralising successful swaps or paying for the option itself. Unfortunately, the initiators do not always have participant's asset: They may just hope to get some participant's asset with only his asset. Before doing an Atomic Swap, the initiator should get some participant's asset by arbitrary means. For example, he can buy some participant's asset from cryptocurrency exchanges, or initiate a smaller Atomic Swap with shorter timelocks and no premium.

8.8 Related Work

The Atomic Swap protocol was first proposed on the BitcoinTalk forum informally in 2013 [15]. Herlihy et al. first formalised the Atomic Swap protocol [293]. Meyden et al. first formally analysed the Atomic Swap smart contracts [294]. Several Atomic Swap variants were proposed for sidechains [295] and solving conflicts from concurrent operations [296].

The optionality of Atomic Swaps was first identified by a user with ID "ZmnSCPxj" in the Lightning-dev mail list in 2018 [282]. BitMEX Research [297] and Dan Robinson [298] further claimed that the optionality cannot be eliminated in HTLC-based Atomic Swaps. However, they do not quantify the unfairness from such optionality.

Eizinger et al. first tried to address the optionality problem by implementing the premium mechanism in Atomic Swap [299]. However, their protocol is flawed: If Bob keeps not participating in the swap, he will get the premium. Liu used the Atomic Swap to construct the option [300], but paying for the premium requires an extra blockchain besides the two blockchains, and they do not justify its fairness. IDEX [301] escrows the premium on an Ethereum smart contract for Atomic Swaps. However, this scheme can only support ERC20 tokens. Furthermore, IDEX fully controls the smart contract, so makes no difference with centralised exchanges except for the audibility. Interledger [302] proposed an Atomic Swap protocol based on payment channels. In Interledger, Alice (holding coin A) creates a payment channel with Bob (holding coin B) on the blockchain of coin A, and Bob creates a payment channel with Alice on the blockchain of coin B. After that, Alice gradually pays coin A to Bob, while Bob gradually pays coin B to Alice. After both payments are finished, they settle both payment channels. However, this scheme suffers from time-consuming interactive operations and poor efficiency.

Chapter 9

Conclusion

This thesis has systematically analysed blockchain sharding protocols. Based on the systematic analysis, we formalised and improved two overlooked primitives in blockchain sharding protocols, namely shard allocation and decentralised randomness beacon (DRB), and suggested countermeasures against security issues in cross-chain 51% attacks and atomic swaps. We provided formal security proofs and experimental evaluations on the proposed constructions and countermeasures, demonstrating their security and practicality.

A number of open challenges remain as future work. The first challenge is to design a more scalable blockchain sharding protocol leveraging our proposed primitives. While our proposed primitives are designed to be plug-in replacements compatible with existing blockchain sharding protocols, a concrete design with formal security proofs and experimental results is of interest for the real-world adoption. Compared to other scalability solutions that prefer security than decentralisation, such a sharding protocol prefers decentralisation than security, and thus offers another possible trade-off over the blockchain trilemma. This trade-off might be of interest for certain blockchain protocols and smart contract applications.

The second challenge is to further improve the proposed primitives in terms of security assumptions and overhead. For example, RANDCHAIN still requires permissioned settings, and constructing a DRB for permissionless settings is still an open challenge important for permissionless blockchains. A permissionless DRB protocol can be used for providing secure randomness for other permissionless systems, most notably permissionless blockchains. Meanwhile, in order to employ existing permissioned

DRB protocols to permissionless systems, the system needs to elect a fixed committee, which introduces extra complexity in communication and design.

The third challenge is to define different notions for our identified security properties in shard allocation and DRB. For example, our notion of delivery-fairness is unachievable in partially synchronous and asynchronous networks, thus DRBs that aim to achieve this notion cannot tolerate network partitions. It might be possible to formalise a weaker delivery-fairness notion that is achievable in partially synchronous networks. This allows us to design a delivery-fair DRB that tolerates network partitions and thus can be deployed in more adversarial network conditions.

Bibliography

[1] S. Nakamoto *et al.*, “Bitcoin: A peer-to-peer electronic cash system”, 2008.

[2] coinmarketcap.com, *Top 100 cryptocurrencies by market capitalization*, 2019. [Online]. Available: `https://coinmarketcap.com`.

[3] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, “A secure sharding protocol for open blockchains”, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 17–30.

[4] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding”, in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 583–598.

[5] M. Zamani, M. Movahedi, and M. Raykova, “Rapidchain: Scaling blockchain via full sharding”, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 931–948.

[6] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, “Chainspace: A sharded smart contracts platform”, in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, 2018.

[7] J. Wang and H. Wang, “Monoxide: Scale out Blockchains with Asynchronous Consensus Zones”, in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[8] Z. Team *et al.*, “The ZILLIQA Technical Whitepaper”, *Retrieved September*, 2017.

[9] “Ethereum/wiki”. `https://eth.wiki/sharding/`. (2020).

[10] G. Wang, Z. J. Shi, M. Nixon, and S. Han, “Sok: Sharding on blockchain”, in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019*, 2019.

[11] R. Han, J. Yu, and R. Zhang, “Analysing and Improving Shard Allocation Protocols for Sharded Blockchains”, 2020, https://eprint.iacr.org/2020/943.

[12] A. Zamyatin, M. Al-Bassam, D. Zindros, *et al.*, “Sok: Communication across distributed ledgers”, IACR Cryptology ePrint Archive, 2019: 1128, Tech. Rep., 2019.

[13] nicehash, *Nicehash - largest crypto-mining marketplace*, 2019. [Online]. Available: https://www.nicehash.com.

[14] J. Bonneau, “Why buy when you can rent? - bribery attacks on bitcoin-style consensus”, in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC*, 2016, pp. 19–26.

[15] T. Nolan, *Alt chains and atomic transfers. bitcointalk. org*, 2013. [Online]. Available: https://bitcointalk.org/index.php?topic=193281.0.

[16] J. C. Cox, S. A. Ross, and M. Rubinstein, “Option pricing: A simplified approach”, *Journal of financial Economics*, vol. 7, no. 3, pp. 229–263, 1979.

[17] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony”, *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[18] J. Garay, A. Kiayias, and N. Leonardos, “The bitcoin backbone protocol: Analysis and applications”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015.

[19] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, “Single secret leader election”, in *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*.

[20] P. A. Bernstein, P. A. Bernstein, and N. Goodman, “Concurrency control in distributed database systems”, *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[21] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370.

[23] C. H. Papadimitriou, “The serializability of concurrent database updates”, *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[24] M. J. Franklin, *Concurrency control and recovery.* 1997.

[25] R. Guerraoui, “Non-blocking atomic commit in asynchronous distributed systems with failure detectors”, *Distributed Computing*, vol. 15, no. 1, pp. 17–25, 2002.

[26] E. A. Brewer, “Towards robust distributed systems”, in *PODC*, vol. 7, 2000.

[27] P. Sousa, N. F. Neves, and P. Verissimo, “How resilient are distributed f fault/intrusion-tolerant systems?”, in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, IEEE, 2005, pp. 98–107.

[28] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and M. I. Zablotchi, “Leaderless consensus”, Tech. Rep., 2021.

[29] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication", *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, p. 12, 2018.

[30] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[31] "Ethereum/eth2.0-specs". `https://github.com/ethereum/eth2.0-specs`. (2020).

[32] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults", *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[33] J. Aspnes, C. Jackson, and A. Krishnamurthy, "Exposing computationally-challenged Byzantine impostors", Technical Report YALEU/DCS/TR-1332, Yale University Department of Computer ..., Tech. Rep., 2005.

[34] M. Castro, B. Liskov, *et al.*, "Practical Byzantine fault tolerance", in *OSDI*, 1999.

[35] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing", in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 279–296.

[36] E. Syta, P. Jovanovic, E. K. Kogias, *et al.*, "Scalable bias-resistant distributed randomness", in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[37] S. Sen and M. J. Freedman, "Commensal cuckoo: Secure group partitioning for large-scale services", *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 33–39, 2012.

[38] L. Ren, K. Nayak, I. Abraham, and S. Devadas, “Practical synchronous byzantine consensus”, *arXiv preprint arXiv:1704.02397*, 2017.

[39] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger”, *Ethereum project yellow paper*, 2014.

[40] “Randao: A dao working as rng of ethereum”. `https://github.com/randao/randao`. (2020).

[41] V. Buterin and V. Griffith, “Casper the friendly finality gadget”, *arXiv preprint arXiv:1710.09437*, 2017.

[42] M. Fitzi, P. Gazi, A. Kiayias, and A. Russell, “Parallel Chains: Improving Throughput and Latency of Blockchain Protocols via Parallel Composition.”, *IACR Cryptology ePrint Archive*, vol. 2018, p. 1119, 2018.

[43] H. Yu, I. Nikolic, R. Hou, and P. Saxena, “OHIE: Blockchain Scaling Made Simple”, 2020.

[44] J. Niu, “Eunomia: A Permissionless Parallel Chain Protocol Based on Logical Clock”, *arXiv preprint arXiv:1908.07567*, 2019.

[45] S. Forestier and D. Vodenicarevic, “Blockclique: scaling blockchains through transaction sharding in a multithreaded block graph”, *arXiv preprint arXiv:1803.09029*, 2018.

[46] D. Hensgen, R. Finkel, and U. Manber, “Two algorithms for barrier synchronization”, *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.

[47] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable”, in *International conference on financial cryptography and data security*, Springer, 2014, pp. 436–454.

[48] J. Yu, D. Kozhaya, J. Decouchant, and P. Verissimo, “RepuCoin: Your reputation is your power”, *IEEE Transactions on Computers*, 2019.

[49] Y. Wang, “Byzantine fault tolerance in partial synchronous networks”, 2020.

[50] *Formal analysis of the cbc casper consensus algorithm with tla+*. [Online]. Available: https://blog.trailofbits.com/2019/10/25/formal-analysis-of-the-cbc-casper-consensus-algorithm-with-tla.

[51] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. Raynal, “Fault-tolerant leader election in mobile dynamic distributed systems”, in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, IEEE, 2013, pp. 78–87.

[52] R. Pass, L. Seeman, and A. Shelat, “Analysis of the blockchain protocol in asynchronous networks”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2017, pp. 643–673.

[53] L. Ren, “Analysis of nakamoto consensus.”, *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 943, 2019.

[54] S. Bano, A. Sonnino, M. Al-Bassam, *et al.*, “Consensus in the age of blockchains”, 2019.

[55] J. A. Garay and A. Kiayias, “SoK: A Consensus Taxonomy in the Blockchain Era”, 2020.

[56] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, “A Survey of Distributed Consensus Protocols for Blockchain Networks”, *arXiv preprint arXiv:1904.04098*, 2019.

[57] C. Natoli, J. Yu, V. Gramoli, and P. Esteves-Verissimo, “Deconstructing Blockchains: A Comprehensive Survey on Consensus, Membership and Structure”, *arXiv preprint arXiv:1908.08316*, 2019.

[58] M. Vukolić, “The quest for scalable blockchain fabric: Proof-of-work vs. bft replication”, in *International workshop on open problems in network security*, Springer, 2015, pp. 112–125.

[59] F. Pedone and R. Guerraoui, “On transaction liveness in replicated databases”, in *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, IEEE, 1997, pp. 104–109.

[60] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores”, 2014.

[61] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, “An evaluation of distributed concurrency control”, *Proceedings of the VLDB Endowment*, 2017.

[62] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers”, in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2020, pp. 294–308.

[63] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[64] H. Pagnia and F. C. Gärtner, “On the impossibility of fair exchange without a trusted third party”, Technical Report TUD-BS-1999-02, Darmstadt University of Technology . . ., Tech. Rep., 1999.

[65] “Data Concurrency and Consistency”, https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm.

[66] B. Kemme, G. Ramalingam, A. Schiper, M. Shapiro, and K. Vaswani, “Consistency in Distributed Systems”, *Dagstuhl Reports*, vol. 3, no. 2, pp. 92–126, Jun. 2013. DOI: 10.4230/DagRep.3.2.92. [Online]. Available: https://hal.inria.fr/hal-00932737.

[67] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez, *Consistency management in cloud storage systems.* 2014.

[68] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing", in *28th Annual Symposium on Foundations of Computer Science (FOCS 1987)*, 1987.

[69] D. Dolev and R. Reischuk, "Bounds on information exchange for byzantine agreement", *Journal of the ACM (JACM)*, vol. 32, no. 1, pp. 191–204, 1985.

[70] F. Chang, J. Dean, S. Ghemawat, *et al.*, "Bigtable: A distributed storage system for structured data", *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[71] G. DeCandia, D. Hastorun, M. Jampani, *et al.*, "Dynamo: Amazon's highly available key-value store", *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[72] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores", in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 79–94.

[73] B. Augustin, T. Friedman, and R. Teixeira, "Measuring load-balanced paths in the Internet", in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 149–160.

[74] M. Annamalai, K. Ravichandran, H. Srinivas, *et al.*, "Sharding the shards: managing datastore locality at scale with Akkio", in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 445–460.

[75] "The zilliqa design story piece by piece: Part 1 (network sharding)", 2020, https://blog.zilliqa.com/https-blog-zilliqa-

com-the-zilliqa-design-story-piece-by-piece-part1-d9cb32ea1e65.

[76] "Ethereum sharding: Overview and finality", 2020, https://medium.com/@icebearhww/ethereum-sharding-and-finality-65248951f649.

[77] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks", in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ACM, 2006.

[78] V. Buterin. "Serenity design rationale". https://notes.ethereum.org/@vbuterin/rkhCgQteN?type=view. (2020).

[79] "On sharding blockchains faqs". https://eth.wiki/sharding/Sharding-FAQs. (2020).

[80] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer, "Divide and Scale: Formalization of Distributed Ledger Sharding Protocols", *arXiv preprint arXiv:1910.10434*, 2019.

[81] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, "Shard scheduler: Object placement and migration in sharded account-based blockchains", in *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, 2021, pp. 43–56.

[82] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis", in *International Workshop on Peer-to-Peer Systems*, Springer, 2005, pp. 205–216.

[83] Y. Kulbak, D. Bickson, *et al.*, "The eMule protocol specification", *eMule project, http://sourceforge. net*, 2005.

[84] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Raynal, "Distributed slicing in dynamic systems", in *27th International Conference on Distributed Computing Systems (ICDCS'07)*, IEEE, 2007, pp. 66–66.

[85] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse, “A fast distributed slicing algorithm”, in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, 2008, pp. 427–427.

[86] F. Maia, M. Matos, R. Oliveira, and E. Riviere, “Slicing as a distributed systems primitive”, in *2013 Sixth Latin-American Symposium on Dependable Computing*, IEEE, 2013, pp. 124–133.

[87] D. J. DeWitt, J. F. Naughton, and D. F. Schneider, “Parallel sorting on a shared-nothing architecture using probabilistic splitting”, University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1991.

[88] A. V. Oppenheim, *Discrete-time signal processing*. Pearson Education, 1999.

[89] A. Dembo, S. Kannan, E. N. Tas, *et al.*, “Everything is a race and nakamoto always wins”, in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 859–878.

[90] P. Flajolet and R. Sedgewick, *Analytic combinatorics*. cambridge University press, 2009.

[91] C. Decker and R. Wattenhofer, “Information propagation in the bitcoin network”, in *IEEE P2P 2013 Proceedings*, IEEE, 2013, pp. 1–10.

[92] X. Qian, “Improved authenticated data structures for blockchain synchronization”, Ph.D. dissertation, 2018.

[93] “Warp sync - wiki parity tech documentation”. https://wiki.parity.io/Warp-Sync. (2020).

[94] J. R. Douceur, “The sybil attack”, in *International workshop on peer-to-peer systems*, Springer, 2002.

[95] J. Dinger and H. Hartenstein, "Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration", in *First International Conference on Availability, Reliability and Security (ARES'06)*, IEEE, 2006, 8–pp.

[96] A. C.-C. Yao, "Some complexity questions related to distributive computing (preliminary report)", in *Proceedings of the eleventh annual ACM symposium on Theory of computing*, 1979, pp. 209–213.

[97] J. Zhao, J. Yu, and J. K. Liu, "Consolidating Hash Power in Blockchain Shards with a Forest", in *International Conference on Information Security and Cryptology*, Springer, 2019, pp. 309–322.

[98] B. Awerbuch and C. Scheideler, "Robust random number generation for peer-to-peer systems", in *International Conference On Principles Of Distributed Systems*, Springer, 2006.

[99] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography", *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[100] J. Kelsey, L. T. Brandão, R. Peralta, and H. Booth, "A reference for randomness beacons: Format and protocol version 2", National Institute of Standards and Technology, Tech. Rep., 2019.

[101] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "HydRand: Efficient Continuous Distributed Randomness", in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 32–48.

[102] "Random uchile - random uchile". https://beacon.clcert.cl/en/. (2020).

[103] "Brazilian beacon". https://beacon.inmetro.gov.br/. (2020).

[104] "Distributed randomness beacon — cloudflare". https://www.cloudflare.com/leagueofentropy/. (2020).

[105] "Unicorn beacon by lacal". http://trx.epfl.ch/beacon/index.php. (2020).

[106] "Drand - distributed randomness beacon". https://drand.love/. (2020).

[107] I. Cascudo and B. David, "Albatross: Publicly attestable batched randomness based on secret sharing",

[108] A. K. Lenstra and B. Wesolowski, "A random zoo: Sloth, unicorn, and trx", *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 366, 2015.

[109] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass, "Continuous verifiable delay functions", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2020, pp. 125–154.

[110] R. Han, J. Yu, and H. Lin, "Randchain: Decentralised randomness beacon from sequential proof-of-work", *IACR Cryptol. ePrint Arch.*, 2020.

[111] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source.", *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1015, 2015.

[112] J. Clark and U. Hengartner, "On the use of financial data as a random beacon.", *EVT/WOTE*, vol. 89, 2010.

[113] J. Benet and N. Greco, "Filecoin: A decentralized storage network", *Protoc. Labs*, pp. 1–36, 2018.

[114] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions", in *40th Annual Symposium on Foundations of Computer Science*, IEEE, 1999.

[115] Y. Dodis, "Efficient construction of (distributed) verifiable random functions", in *International Workshop on Public Key Cryptography*, Springer, 2003, pp. 1–17.

[116] S. Goldberg, J. Vcelak, D. Papadopoulos, and L. Reyzin, “Verifiable random functions (VRFs)”, 2018.

[117] S. Hohenberger and B. Waters, “Constructing verifiable random functions with large input spaces”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2010, pp. 656–672.

[118] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda, “Privacy pass: Bypassing internet challenges anonymously”, *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 164–180, 2018.

[119] S Goldberg, D Papadopoulos, and J Vcelak, *draft-goldbe-vrf: Verifiable Random Functions.(2017)*, 2017.

[120] V. Bagaria, A. Dembo, S. Kannan, *et al.*, “Proof-of-stake longest chain protocols: Security vs predictability”, *arXiv preprint arXiv:1910.02218*, 2019.

[121] J. Neu, E. N. Tas, and D. Tse, “Ebb-and-flow protocols: A resolution of the availability-finality dilemma”, *arXiv preprint arXiv:2009.04987*, 2020.

[122] S. Sankagiri, X. Wang, S. Kannan, and P. Viswanath, “Blockchain cap theorem allows user-dependent adaptivity and finality”, *arXiv preprint arXiv:2010.13711*, 2020.

[123] N. Barić and B. Pfitzmann, “Collision-free accumulators and fail-stop signature schemes without trees”, in *International conference on the theory and applications of cryptographic techniques*, Springer, 1997, pp. 480–494.

[124] “Crates/rug”, 2020, `https://crates.io/crates/rug`.

[125] “Crates/bitvec”, 2020, `https://crates.io/crates/bitvec`.

[126] "Schnorr vrfs and signatures on the ristretto group", 2020, https://github.com/w3f/schnorrkel.

[127] "The ristretto group", 2020, https://ristretto.group/.

[128] "Crates/rand", 2020, https://crates.io/crates/rand.

[129] "Cargo-bench", 2020, https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[130] "Criterion.rs", 2020, https://github.com/bheisler/criterion.rs.

[131] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the internet", *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 331–342, 2004.

[132] M. Saad, S. Chen, and D. Mohaisen, "Root cause analyses for the deteriorating bitcoin network synchronization", in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2021, pp. 239–249.

[133] M. Saad, S. Chen, and D. Mohaisen, "Syncattack: Double-spending in bitcoin without mining power", in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1668–1685.

[134] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system", *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[135] B. F. Cooper, R. Ramakrishnan, U. Srivastava, *et al.*, "PNUTS: Yahoo!'s hosted data serving platform", *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[136] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the GPU and some techniques to improve load imbalance", in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IEEE, 2015, pp. 610–617.

[137] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network", in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001, pp. 161–172.

[138] C. Lin, Y. Jiang, X. Chu, H. Yang, *et al.*, "An effective early warning scheme against pollution dissemination for BitTorrent", in *GLOBECOM 2009-2009 IEEE Global Telecommunications Conference*, IEEE, 2009, pp. 1–7.

[139] X. Lou and K. Hwang, "Collusive piracy prevention in P2P content delivery networks", *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 970–983, 2009.

[140] P. Dhungel, D. W. 0001, B. Schonhorst, and K. W. Ross, "A measurement study of attacks on BitTorrent leechers.", in *IPTPS*, vol. 8, 2008, pp. 7–7.

[141] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks", in *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, IEEE, 2006, pp. 117–124.

[142] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system", *arXiv preprint arXiv:1805.04548*, 2018.

[143] A. Cherniaeva, I. Shirobokov, and O. Shlomovits, "Homomorphic Encryption Random Beacon.", *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1320, 2019.

[144] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 66–98.

[145] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies", in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.

[146] D. Galindo, J. Liu, M. Ordean, and J.-M. Wong, "Fully distributed verifiable random functions and their application to decentralised random beacons", in *European Symposium on Security and Privacy*, 2021.

[147] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol", in *Annual International Cryptology Conference*, Springer, 2017, pp. 357–388.

[148] I. Cascudo and B. David, "SCRAPE: Scalable randomness attested by public entities", in *International Conference on Applied Cryptography and Network Security*, Springer, 2017, pp. 537–556.

[149] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures", in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1751–1767.

[150] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems", in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 1999.

[151] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, “Cogsworth: Byzantine View Synchronization”, *arXiv preprint arXiv:1909.05204*, 2019.

[152] O. Naor and I. Keidar, “Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR”, in *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, ser. LIPIcs, vol. 179, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 26:1–26:17.

[153] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hot-Stuff: BFT consensus with linearity and responsiveness”, in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ACM, 2019, pp. 347–356.

[154] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, “Verifiable delay functions”, in *Annual international cryptology conference*, Springer, 2018, pp. 757–788.

[155] “[ANSWERED] Why is bitcoin proof of work parallelizable ?”, https://bitcointalk.org/index.php?topic=46739.0.

[156] K. Pietrzak, “Simple verifiable delay functions”, in *10th innovations in theoretical computer science conference (itcs 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[157] B. Wesolowski, “Efficient verifiable delay functions”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 379–407.

[158] “paritytech/parity-bitcoin: The Parity Bitcoin client”, https://github.com/paritytech/parity-bitcoin.

[159] J. A. Garay, A. Kiayias, and N. Leonardos, “Full analysis of nakamoto consensus in bounded-delay networks.”, *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 277, 2020.

[160] B. Y. Chan and E. Shi, “Streamlet: Textbook Streamlined Blockchains.”, in *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, ACM, 2020, pp. 1–11.

[161] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. Weippl, “RandRunner: Distributed Randomness from Trapdoor VDFs with Strong Uniqueness”, in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021.

[162] C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner, “Craft: Composable randomness beacons and output-independent abort mpc from time”, 2020.

[163] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model”, in *31st International Symposium on Distributed Computing (DISC 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[164] J. Long and R. Wei, “Nakamoto Consensus with Verifiable Delay Puzzle”, *arXiv preprint arXiv:1908.06394*, 2019.

[165] B. Cohen and K. Pietrzak, *The chia network blockchain*, 2019.

[166] J. Brown-Cohen, A. Narayanan, A. Psomas, and S. M. Weinberg, “Formal barriers to longest-chain proof-of-stake protocols”, in *Proceedings of the 2019 ACM Conference on Economics and Computation*, 2019, pp. 459–473.

[167] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, “Proofs of work from worst-case assumptions”, in *Annual International Cryptology Conference*, Springer, 2018, pp. 789–819.

[168] S. Dobson, S. Galbraith, and B. Smith, “Trustless groups of unknown order with hyperelliptic curves”, 2020.

[169] B. Wesolowski and R. Williams, "Lower bounds for the depth of modular squaring", 2020.

[170] J. Katz, J. Loss, and J. Xu, "On the security of time-lock puzzles and timed commitments", in *Theory of Cryptography Conference*, Springer, 2020, pp. 390–413.

[171] N. Döttling, S. Garg, G. Malavolta, and P. N. Vasudevan, "Tight verifiable delay functions", in *International Conference on Security and Cryptography for Networks*, Springer, 2020, pp. 65–84.

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

[173] M. Mahmoody, T. Moran, and S. Vadhan, "Publicly verifiable proofs of sequential work", in *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, 2013, pp. 373–388.

[174] B. Cohen and K. Pietrzak, "Simple proofs of sequential work", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 451–467.

[175] C. Percival, *Stronger key derivation via sequential memory-hard functions*, 2009.

[176] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications", in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 292–302.

[177] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, "Scrypt is maximally memory-hard", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2017, pp. 33–62.

[178] S. Deb, S. Kannan, and D. Tse, *PoSAT: Proof-of-Work Availability and Unpredictability, without the Work*, 2020. arXiv: 2010.08154 [cs.CR].

[179] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake", *self-published paper, August*, vol. 19, 2012.

[180] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas, "Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 913–930.

[181] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake", in *International Conference on Financial Cryptography and Data Security*, Springer, 2019, pp. 23–41.

[182] L. Fan and H.-S. Zhou, "A scalable proof-of-stake blockchain in the open setting (or, how to mimic nakamoto's design via proof-of-stake)", Cryptology ePrint Archive, Report 2017/656, Tech. Rep., 2017.

[183] M. Chen, C. Hazay, Y. Ishai, *et al.*, "Diogenes: Lightweight scalable rsa modulus generation with a dishonest majority", in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 590–607.

[184] M. Chen, R. Cohen, J. Doerner, *et al.*, "Multiparty Generation of an RSA Modulus", in *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 12172, Springer, 2020, pp. 64–93.

[185] V. Attias, L. Vigneri, and V. Dimitrov, "Implementation study of two verifiable delay functions", in *2nd International Conference on*

Blockchain Economics, Security and Protocols (Tokenomics 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[186] D. Boneh, B. Bünz, and B. Fisch, "A Survey of Two Verifiable Delay Functions.", *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 712, 2018.

[187] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability", *arXiv preprint arXiv:1906.08936*, 2019.

[188] T. Dinsdale-Young, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, "Afgjort: A partially synchronous finality layer for blockchains", in *International Conference on Security and Cryptography for Networks*, Springer, 2020, pp. 24–44.

[189] *AMD Breaks 8GHz Overclock with Upcoming FX Processor, Sets World Record*, http://hothardware.com/News/AMD-Breaks-Frequency-Record-with-Upcoming-FX-Processor/.

[190] "Why has CPU frequency ceased to grow?", https://software.intel.com/content/www/us/en/develop/blogs/why-has-cpu-frequency-ceased-to-grow.html.

[191] "supranational/vdf-fpga-round1-results", https://github.com/supranational/vdf-fpga-round1-results.

[192] "supranational/vdf-fpga-round2-results", https://github.com/supranational/vdf-fpga-round2-results.

[193] "supranational/vdf-fpga-round3-results", https://github.com/supranational/vdf-fpga-round3-results.

[194] K. A. Negy, P. R. Rizun, and E. G. Sirer, "Selfish mining re-examined", in *International Conference on Financial Cryptography and Data Security*, Springer, 2020, pp. 61–78.

[195] E. Heilman, “One weird trick to stop selfish miners: Fresh bitcoins, a solution for the honest miner”, in *International Conference on Financial Cryptography and Data Security*, Springer, 2014, pp. 161–162.

[196] R. Zhang and B. Preneel, “Publish or perish: A backward-compatible defense against selfish mining in bitcoin”, in *Cryptographers’ Track at the RSA Conference*, Springer, 2017, pp. 277–292.

[197] H. Abusalah, C. Kamath, K. Klein, K. Pietrzak, and M. Walter, “Reversible proofs of sequential work”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 277–291.

[198] “facebook/rocksdb: A library that provides an embeddable, persistent key-value store for fast storage.”, https://github.com/facebook/rocksdb.

[199] *Protocol documentation - Bitcoin Wiki*, https://en.bitcoin.it/wiki/Protocol_documentation, 2015.

[200] “dstat-real/dstat: Versatile resource statistics tool”, https://github.com/dstat-real/dstat.

[201] H. Yu, I. Nikolić, R. Hou, and P. Saxena, “Ohie: Blockchain scaling made simple”, in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 90–105.

[202] P. Erdős and A. Rényi, “On the evolution of random graphs”, *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[203] “Drand - Distributed Randomness Beacon.”, https://drand.love/.

[204] D. Yakira, A. Asayag, I. Grayevsky, and I. Keidar, “Economically Viable Randomness”, *arXiv preprint arXiv:2007.03531*, 2020.

[205] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum", *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[206] D. Yakira, I. Grayevsky, and A. Asayag, "Rational Threshold Cryptosystems", *arXiv preprint arXiv:1901.01148*, 2019.

[207] M. Andrychowicz and S. Dziembowski, "Distributed Cryptography Based on the Proofs of Work.", *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 796, 2014.

[208] P. Daian, S. Goldfeder, T. Kell, *et al.*, "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability", in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 566–583.

[209] J. Großschädl, A. Szekely, and S. Tillich, "The energy cost of cryptographic key establishment in wireless sensor networks", in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 380–382.

[210] D. Malkhi and M. Reiter, "Byzantine quorum systems", *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[211] T.-H. H. Chan, R. Pass, and E. Shi, "Consensus through herding", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 720–749.

[212] "BCH Avalanche Transactions Show Finality Speeds 10x Faster Than Ethereum", `https://news.bitcoin.com/bch-avalanche-transactions-show-finality-speeds-10x-faster-than-ethereum/`.

[213] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "Randpiper–reconfiguration-friendly random beacons with quadratic communi-

cation", in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3502–3524.

[214] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup", *Cryptology ePrint Archive*, 2021.

[215] "Distributed Randomness Beacon — Cloudflare", https://www.cloudflare.com/leagueofentropy/.

[216] *Chainlink VRF*, https://docs.chain.link/docs/chainlink-vrf/.

[217] *PancakeSwap Lottery*, https://pancakeswap.finance/lottery.

[218] *PolyRoll: Decentralized Games*, https://polyroll.org/.

[219] *The Economic Impact of Random Rewards in Blockchain Video Games*, https://blog.chain.link/the-economic-impact-of-random-rewards-in-blockchain-video-games/.

[220] *CryptOrchids: NFT plants that must be watered weekly*, https://cryptorchids.io/.

[221] *16 Ways to Create Dynamic Non-Fungible Tokens (NFT) Using Chainlink Oracles*, https://blog.chain.link/create-dynamic-nfts-using-chainlink-oracles/.

[222] "Polygon Hermez", https://hermez.io/.

[223] "Celo: Mobile-First DeFi Platform for Fast, Secure, and Stable Digital Payments", https://celo.org/.

[224] "Join Hermez Trusted Setup Phase 2 Ceremony!", https://blog.hermez.io/hermez-trusted-setup-phase-2/.

[225] "Phase 2 setup random beacon of Celo", https://github.com/celo-org/celo-bls-snark-rs/issues/227.

[226] S. Bowe, A. Gabizon, and I. Miers, “Scalable multi-party computation for zk-snark parameters in the random beacon model”, *Cryptology ePrint Archive*, 2017.

[227] R. Cohen and Y. Lindell, “Fairness versus guaranteed output delivery in secure multiparty computation”, *Journal of Cryptology*, vol. 30, no. 4, pp. 1157–1186, 2017.

[228] R. Pass and E. Shi, “Thunderella: Blockchains with optimistic instant confirmation”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 3–33.

[229] “Drand: A Distributed Randomness Beacon Daemon - Go implementation”, `https://github.com/drand/drand`.

[230] A. Kiayias and G. Panagiotakos, “Speed-security tradeoffs in blockchain protocols.”, *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1019, 2015.

[231] R. Pass, E. Shi, and F. Tramer, “Formal abstractions for attested execution secure processors”, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2017, pp. 260–289.

[232] “Drand Specification”, `https : / / drand . love / docs / specification`.

[233] A. Boldyreva, “Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme”, in *International Workshop on Public Key Cryptography*, Springer, 2003, pp. 31–46.

[234] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the Weil pairing”, in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2001.

[235] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang, "Improved extension protocols for byzantine broadcast and agreement", in *34th International Symposium on Distributed Computing (DISC 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[236] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Good-case latency of byzantine broadcast: A complete categorization", *arXiv preprint arXiv:2102.07240*, 2021.

[237] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, *Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency*, 2020. arXiv: `2003.13155 [cs.CR]`.

[238] B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic voting", in *Annual International Cryptology Conference*, Springer, 1999, pp. 148–164.

[239] "HydRand: Python implementation of the HydRand protocol", `https://github.com/PhilippSchindler/HydRand`.

[240] "SPURT implementation, forked from HydRand", `https://github.com/sourav1547/HydRand`.

[241] komodoplatform.com, *The anatomy of a 51% attack and how you can prevent one*, 2019. [Online]. Available: `https://komodoplatform.com/51-attack-how-komodo-can-help-prevent-one`.

[242] T. Zimwara, *Ethereum classic suffers 51% attack again: Delisting risk amplified*, 2020. [Online]. Available: `https://news.bitcoin.com/ethereum-classic-suffers-51-attack-again-delisting-risk-amplified`.

[243] P. Thompson, *Over $1m double-spent in latest ethereum classic 51% attack*, 2020. [Online]. Available: `https://coingeek.com/over-1m-double-spent-in-latest-ethereum-classic-51-attack`.

[244] D. Howarth, *Hackers launch third 51% attack on ethereum classic this month*, 2020. [Online]. Available: https://decrypt.co/40196/hackers-launch-third-51-attack-on-ethereum-classic-this-month.

[245] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "BAR fault tolerance for cooperative services", in *SOSP 2005*, 2005, pp. 45–58.

[246] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable", in *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*.

[247] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack", in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, 2016, pp. 305–320.

[248] Y. Kwon, D. Kim, Y. Son, E. Y. Vasserman, and Y. Kim, "Be selfish and avoid dilemmas: Fork after withholding (FAW) attacks on bitcoin", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 195–209.

[249] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin", in *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*.

[250] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications", in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory*

and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II, 2015, pp. 281–310.

[251] R. Pass, L. Seeman, and A. Shelat, "Analysis of the blockchain protocol in asynchronous networks", in *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, 2017, pp. 643–673.

[252] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol with chains of variable difficulty", in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, 2017, pp. 291–323.

[253] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains", in *CCS*, 2016.

[254] L. Kiffer, R. Rajaraman, and A. Shelat, "A better method to analyze blockchain consistency", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 729–744.

[255] R. Zhang and B. Preneel, "Lay down the common metrics: Evaluating proof-of-work consensus protocols' security", in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P, IEEE, 2019.

[256] M. Carlsten, H. A. Kalodner, S. M. Weinberg, and A. Narayanan, "On the instability of bitcoin without the block reward", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 154–167.

[257] A. Judmayer, N. Stifter, A. Zamyatin, *et al.*, *Pay-to-win: Incentive attacks on proof-of-work cryptocurrencies*, Cryptology ePrint Archive, 2019.

[258] etherchain, *Top miners over the last 24h - etherchain.org*, 2019. [Online]. Available: https://www.etherchain.org/charts/topMiners.

[259] gate.io, *Gate.io research: Confirmed the etc 51% attack and attacker's accounts - gate.io news*, 2019. [Online]. Available: https://www.gate.io/article/16735.

[260] D. Z. Morris, *The ethereum classic 51% attack is the height of crypto-irony*. [Online]. Available: https://breakermag.com/the-ethereum-classic-51-attack-is-the-height-of-crypto-irony.

[261] gate.io, *Gate.io got back 100k usd value of etc from the etc 51% attacker*, 2019. [Online]. Available: https://www.gate.io/article/16740.

[262] I. Chadès, G. Chapron, M.-J. Cros, F. Garcia, and R. Sabbadin, "Mdp-toolbox: A multi-platform toolbox to solve stochastic dynamic programming problems", *Ecography*, vol. 37, no. 9, pp. 916–920, 2014.

[263] S. M. Ross, *Introduction to stochastic dynamic programming*. 2014.

[264] blockchain.com, *Bitcoin hashrate distribution - blockchain.info*, 2019. [Online]. Available: https://www.blockchain.com/en/pools.

[265] komodoplatform, *Security: Delayed proof of work (dpow)*, 2018. [Online]. Available: https://komodoplatform.com/security-delayed-proof-of-work-dpow/.

[266] gate.io, *Gate.io - the gate of blockchain assets exchange*, 2019. [Online]. Available: https://www.gate.io.

[267] D. Z. Morris, *The Ethereum Classic 51% attack is the height of crypto-irony*, 2019. [Online]. Available: https://breakermag.com/the-ethereum-classic-51-attack-is-the-height-of-crypto-irony/.

[268] W. Messamore, *Nicehash to smaller cryptocurrency miners : If you can't beat 51% attackers who lease our hash power, join them*, 2019. [Online]. Available: https://www.ccn.com/nicehash-to-smaller-cryptocurrency-miners-if-you-cant-beat-51-attackers-who-lease-our-hash-power-join-them.

[269] M. Nesbitt, *Deep chain reorganization detected on ethereum classic (etc)*, 2019. [Online]. Available: https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de.

[270] reddit, *How many confirms is considered 'safe' in ethereum?*, 2019. [Online]. Available: https://www.reddit.com/r/ethereum/comments/4eplsv/how_many_confirms_is_considered_safe_in_ethereum.

[271] J. Ray, *Dagger hashimoto*, 2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/Dagger-Hashimoto.

[272] J. Ray, *Ethash design rationale*, 2018. [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethash-Design-Rationale.

[273] E. C. Community, *Tweet of ethereum classic*, 2020. [Online]. Available: https://twitter.com/eth_classic/status/1299832466643931136.

[274] Y. Kwon, H. Kim, J. Shin, and Y. Kim, "Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash?", in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 935–951.

[275] A. Spiegelman, I. Keidar, and M. Tennenholtz, "Game of coins", *arXiv preprint arXiv:1805.08979*, 2018.

[276] K. Liao and J. Katz, "Incentivizing blockchain forks via whale transactions", in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 264–279.

[277] F. Winzer, B. Herd, and S. Faust, "Temporary censorship attacks in the presence of rational miners", in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, 2019, pp. 357–366.

[278] *distribuyed/index: A comprehensive list of decentralized exchanges (DEX) of cryptocurrencies, tokens, derivatives and futures, and their protocols.* 2019. [Online]. Available: `https://github.com/distribuyed/index`.

[279] *evbots/dex-protocols: A list of protocols for decentralized exchange.* 2019. [Online]. Available: `https://github.com/evbots/dex-protocols`.

[280] *DEXWatch — DEX explorer*, 2019. [Online]. Available: `https://dex.watch/`.

[281] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable off-chain instant payments", 2016.

[282] ZmnSCPxj, *An Argument For Single-Asset Lightning Network. Lightning-dev*, 2018. [Online]. Available: `https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-December/001752.html`.

[283] D. J. Higham, *An introduction to financial option valuation: mathematics, stochastics and computation*. Cambridge University Press, 2004, vol. 13.

[284] Almkglor and Harding, *Payment channels - Bitcoin Wiki*, 2018. [Online]. Available: https://en.bitcoin.it/wiki/Payment_channels.

[285] B. M. Smith, “A history of the global stock market: From ancient rome to silicon valley”, in University of Chicago press, 2004, p. 20.

[286] F. Black and M. Scholes, “The pricing of options and corporate liabilities”, *Journal of political economy*, vol. 81, no. 3, pp. 637–654, 1973.

[287] I. Karatzas and S. E. Shreve, “Brownian motion”, in *Brownian Motion and Stochastic Calculus*, Springer, 1998, pp. 47–127.

[288] J. Hull, *Introduction to futures and options markets*. prentice Hall Englewood Cliffs, NJ, 1991.

[289] *AltCoinExchange/ethatomicswap: Ethereum atomic swap*, 2019. [Online]. Available: https://github.com/AltCoinExchange/ethatomicswap.

[290] K. Okupski, “Bitcoin developer reference”, *Eindhoven*, 2014.

[291] M. Brown, *Bitcoin script for a competitive crowdfunding-like contract*, 2015. [Online]. Available: https://bitcoin.stackexchange.com/questions/36229/bitcoin-script-for-a-competitive-crowdfunding-like-contract.

[292] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*, fifth. pearson education, 2012, p. 2.

[293] M. Herlihy, “Atomic cross-chain swaps”, in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, ACM, 2018, pp. 245–254.

[294] R. van der Meyden, “On the specification and verification of atomic swap smart contracts”, *arXiv preprint arXiv:1811.06099*, 2018.

[295] P. Robinson, D. Hyland-Wood, R. Saltini, S. Johnson, and J. Brainard, "Atomic Crosschain Transactions for Ethereum Private Sidechains", *arXiv preprint arXiv:1904.12079*, 2019.

[296] V. Zakhary, D. Agrawal, and A. E. Abbadi, "Atomic commitment across blockchains", *arXiv preprint arXiv:1905.02847*, 2019.

[297] B. Research, *Atomic Swaps and Distributed Exchanges: The Inadvertent Call Option*, 2019. [Online]. Available: https://blog.bitmex.com/atomic-swaps-and-distributed-exchanges-the-inadvertent-call-option/.

[298] D. Robinson, *HTLCs Considered Harmful*, 2019. [Online]. Available: https://cyber.stanford.edu/sites/g/files/sbiybj9936/f/htlcs_considered_harmful.pdf.

[299] T. Eizinger, L. Fournier, and P. Hoenisch, *The state of atomic swaps. diyhpl.us*, 2018. [Online]. Available: http://diyhpl.us/wiki/transcripts/scalingbitcoin/tokyo-2018/atomic-swaps/.

[300] J. A. Liu, "Atomic swaptions: Cryptocurrency derivatives", *arXiv preprint arXiv:1807.08644*, 2018.

[301] *Whitepaper - Idex*, 2019. [Online]. Available: https://idex.market/whitepaper.

[302] S. Thomas and E. Schwartz, "A protocol for interledger payments", *URL https://interledger. org/interledger. pdf*, 2015.