University of Southampton

Faculty of Engineering and Physical Sciences
Electronics and Computer Science

Assessing Security and Performance of Blockchain Systems and Consensus Protocols: Taxonomies, Methodologies and Benchmarking Procedures

by

Stefano De Angelis

ORCiD: 0000-0002-1168-9064

A thesis for the degree of
Doctor of Philosophy

4th April 2022

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
Electronics and Computer Science

Doctor of Philosophy

Assessing Security and Performance of Blockchain Systems and Consensus Protocols: Taxonomies, Methodologies and Benchmarking Procedures

by Stefano De Angelis

Blockchain promises to improve systems security and trust by decentralising computer infrastructures. However, decentralisation also requires higher complexity that may lead to performance issues. With the rapid growth of blockchain adoption, such properties are paramount, and it becomes crucial to assess them in different application scenarios.

In this PhD thesis, we study performance and security of modern blockchain systems. We first refine the standard concepts of security and dependability, defining a set of properties for blockchain systems. We provide a taxonomy of platforms, consensus protocols, and smart contacts vulnerabilities, and we assess their security according to the proposed properties. We show that consensus strictly impacts system's security. We also argue that it introduces trade-offs with performance that must be understood for building secure and efficient systems. So we design METHUS, a systematic methodology to assess blockchain consensus protocols applying qualitative and quantitative methods. Hence we evaluate two families of consensus protocols used in permissioned blockchains, and we show that a traditional Byzantine Fault Tolerant approach is preferable in this context. Extending the study to permissionless blockchains, we propose PETHARD, a framework to measure performance of consensus employed in two famous blockchains, namely Ethereum and Algorand. Despite promising results, PETHARD only simulates testing setups and cannot be used to evaluate realistic deployments. To this extent, we design PERSECUS which defines the standards for blockchain benchmarking. PERSECUS fosters efficient and precise measurements simulating various setups and real-world scenarios. We benchmark two blockchains, namely Parity and GoQuorum, evaluating their security, performance, and scalability properties. We illustrate that, besides consensus, other blockchain components, such as configuration of nodes parameters and transactions serialisation, strictly affect performance and security.

To conclude this thesis, we discuss the possibility of using *elasticity*, broadly adopted in Cloud Computing to automatise the provisioning of a system, to enhance performance and security in blockchain systems.

Contents

List of Figures

List of Tables

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as: *De Angelis et al.* [10, 51], *Lombardi et al.* [122]

Signed:..... Date:.....

To Mum and Dad

Acknowledgements

I would like to express my gratitude to my supervisors, Prof. Vladimiro Sassone and Dr. Leonardo Aniello for their continuous help and invaluable advice during my PhD. Additionally, I would like to thank my colleague and friend, Dr. Federico Lombardi, for having encouraged me through this extraordinary journey, and for his treasury guidance. I would also like to express my gratitude to my family, especially my parents, for always supporting me. Finally, a special thanks goes to Inés, for always being at my side.

"A distributed network must be capable to remain stable, efficient, and secure. And to prove that, we need theory and experimentation."
Leonard Kleinrock

Abbreviations

AVM	Algorand Virtual Machine
BDM	Blockchain Deployment Manager
BFT	Byzantine Fault Tolerant
BP	Block Period
BSUT	Blockchain System Under Test
CLI	Command Line Interface
dApp	Decentralised Application
DeFi	Decentralised Finance
DHT	Decentralised Hash Table
DLT	Decentralised Ledger Technology
DSS	Decision Support System
EOA	Externally Owned Accounts
EVM	Ethereum Virtual Machine
GST	Global Stabilisation Time
HPC	High Performance Computing
KMD	Key Management Daemon
LAN	Local Area Network
MAM	Metrics Analysis Manager
P2P	Peer-to-Peer
PBFT	Practical Byzantine Fault-Tolerance
PoA	Proof of Authority

PoS	Proof of Stake
PoW	Proof of Work
PPoS	Pure Proof of Stake
PRNG	Pseudorandom Number Generator
SDK	Software Development Kit
SMR	State Machine Replication
SUT	System Under Test
TC	Traffic Control
TPC	Transaction Processing Performance Council
TPS	Transactions per Second
TxAS	Transactions Acceptance Schedule
TxPoolS	Transactions Pooling Schedule
TxSS	Transactions Serialisation Schedule
UTXO	Unspent Transaction
VM	Virtual Machine
VR	Viewstamped Replication
VRF	Verifiable Random Functions
WAN	Wide Area Network
WGM	Workload Generator Manager

Introduction

The Internet we use today is under control of Cloud providers. Services and infrastructures are deployed in large and powerful data centres managed by single authorities. With this architectural pattern, users can access high computation and storage resources without maintaining expensive, time consuming, and on-premise setups. Even though Cloud Computing have revolutionised traditional computer systems, it introduces two serious problems, i.e. *centralisation* and *trust*. With the Cloud paradigm, users entrust their data to centralised authorities, accepting the risk of privacy flaws, data breaches, and censorship. Centralised systems therefore represent *single-points-of-failure* where users have no control of their network infrastructure. Moreover, such systems are susceptible to security issues such as service outages and cyber attacks threatening the confidentiality, integrity, and availability of data and applications. Although users and Cloud providers often agree on legal contracts, *i.e.*, Service Level Agreement (SLA), to ensure adequate efficiency and reliability of systems, centralisation and trust remain pitfalls of Cloud-based architectures.

The early Internet was forged over the idea of *decentralisation*. Information was intended to move freely from peer-to-peer (P2P), without any limitation or third-party in control. For this reason, there is a need to '*re-decentralise*' the Internet. In the last decade, researchers and engineers devoted some efforts investigating decentralised alternatives to traditional computer infrastructures. Broadly, a decentralised system is a P2P network of independent authorities controlling services and resources without the need for trust. This approach may prevent the problem of single-point-of-failure and result in more resilient infrastructures harder to compromise. In this context, the *blockchain* technology, born to secure the Bitcoin distributed ledger, represents a driving force. It allows true P2P computation and provides a unique, immutable, and replicated data structure controlled by the entire network. Notwithstanding blockchain promises to play the role of backbone technology for decentralised systems, there are fundamental challenges and open questions, mainly related to security and efficiency, that have undermined its mainstream adoption to date.

Motivation

The complexity of modern applications and programs is growing exponentially. As a result, there is an increasing need for *secure* and *performant* systems able to cope with heavy workloads and high computation requirements. *Security* is often combined with *dependability*. These concepts embrace several properties defining the ability of a system to tolerate adversarial conditions like errors, faults, and malicious actors. *Security* and *dependability* may directly or indirectly affect *performance*; nowadays several trade-offs between those properties exist. An ideal system should be able to maximise *performance*, without sacrificing *security* and *dependability*. Centralised systems foster *performance* by scaling up resources and parallelising computation. Although the infrastructures can be distributed, their control is centralised, and the security concerns on data *confidentiality* and *availability* remain. For this reason, in a scenario where *dependability* and *security* are paramount, a decentralised design is preferred.

Blockchain is the most prominent technology for building decentralised systems. It provides decentralised computation via *smart contracts* - self-executing programs running on the blockchain itself introduced with Ethereum - and enhances security by distributing trust among a set of independent authorities that collectively manage data and process tasks. A consensus protocol governs the whole network and allows the authorities to converge on the same state. Even though the blockchain paradigm improves security, it also increases complexity. Its decentralised design fosters joint computation across many parties that may lead to performance inefficiencies. For instance, the consensus used in public blockchains like Bitcoin and Ethereum, i.e., the *Proof-of-Work* (PoW), drastically slows down the transactions confirmation time. Conversely, private blockchains like Hyperledger Fabric, adopt lightweight, *Byzantine fault-tolerant* (BFT), consensus protocols which ensure better performance than PoW sacrificing scalability and decentralisation.

A vibrant debate over blockchain trade-offs recently arose across the scientific community. Increasingly more research studies and scientific analysis emerged with the aim to evaluate this novel technology. Among many aspects, most of the effort spent to date has been dedicated to the evaluation of blockchain performance and security balances. In that sense, the consensus protocol has drawn particular attention as the defining component behind performance and security trade-offs. Although the problem of consensus has been broadly studied in the field of distributed systems, with blockchains it has experienced renewed interest from a much wider community. Researchers and scientists from different fields have started designing and implementing new consensus protocols, with the aim to maximise blockchain performance while preserving its fundamental security characteristics. However, notwithstanding optimistic claims, such new protocols lack formal justifications and detailed security analysis; in most cases, performance improvement implies strong assumptions on network synchrony and

impractical trust models. As a result, the effectiveness and applicability of such protocols in the real world, where network communications can be asynchronous and nodes malicious, remain questionable.

Besides consensus, other blockchain components like concurrency control, transactions processing, and cryptographic tasks can lead to performance bottlenecks. Nowadays, there are a plethora of blockchain platforms, each one implementing those components with various methods and techniques. However, although some platform introduce itself as '*superior*' blockchain, a solution able to cope with all performance issues without compromising security seems nowadays questionable. More likely, the future Internet will be composed of several interoperable blockchains with different characteristics and trade-offs. In this extent, application developers can choose from a wide range of disparate platforms, and identifying the best solution for their needs becomes a tough task. However, a wrong choice may lead developers to incur in performance issues or security flaws that can compromise their applications and turn into financial loss.

For many years, performance and security of computer systems have been the subject of several studies. However, despite mathematical models and formal proofs that have been proposed to appraise classic computer systems, most of these methods do not straightforwardly apply for decentralised designs like blockchains. Although decentralisation prevents most of the conventional issues that plagued traditional computer systems, it also introduces new ones. Consensus, distributed computation, serialisability, and inconsistent states are just some of the many aspects that may turn into possible threats. Assessing complex blockchain systems is challenging. Despite some research studies attempting to evaluate blockchain, they usually undertake differing approaches standing on ad-hoc assumptions and provisional methodologies; to date, the lack of general methods and comprehensive procedures have led to divergent and blurry results. The massive adoption of blockchain can only be achieved once developers and researches have the appropriate means to evaluate them. Nowadays fair and comprehensive studies of the emerging blockchain systems are urgently needed. Therefore, it becomes crucial to establish foundational notions and standardised approaches to create a common knowledge in this field. Thus, on this basis, design and implement novel methodologies and comprehensive benchmarks to evaluate and compare security and performance aspects of modern blockchain systems.

Research Aims and Objectives

The aim of this thesis is to study *security*, *dependability* and *performance* aspects of blockchain systems and to provide a suite of *taxonomies*, novel *methodologies* and *benchmark* procedures to assess them both theoretically and experimentally. And to lay a foundation for future research and development in the field of blockchain.

To achieve its aim, this thesis has the following objectives:

- evaluating the most prominent blockchain systems and their different components, in particular defining a foundational knowledge of their security and dependability guarantees through the specification of a systematic taxonomy that can be referenced and used for further comparisons of future systems;
- establishing a standardised methodology to evaluate consensus protocols of blockchain systems, and in particular to assess, both theoretically and experimentally, their performance and security guarantees assuming realistic deployment scenarios;
- designing and implementing novel benchmarking procedures fostering optimised measurements of blockchains' behaviour, in particular for addressing their performance, security, and dependability simulating various deployment scenarios and workload traces.

Thesis Structure and Contributions

In Chapter 1 we outline the context by introducing centralised and decentralised systems, their security, dependability, and performance properties, and therefore we discuss the state-of-the-art technology and tools to assess them. Afterwards, in Chapter 2 we present the blockchain technology, describing its fundamental components and introducing the key terminology and properties that we recall throughout the thesis.

The rest of this work is structured in three pieces of contributions, namely *taxonomies*, *methodologies* and *benchmarks*. In Chapter 3 we propose a taxonomy of security and dependability properties appraising blockchain systems over three dimensions, i.e. *platforms*, *consensus* protocols, and *smart contracts*. We first introduce the most prominent blockchain platforms and their consensus protocols that we consider in our taxonomy, namely Bitcoin with Proof-of-Work (PoW) [138], Ethereum 2.0 with Proof-of-Stake (PoS) [64, 65], Algorand with Pure Proof-of-Stake (PPoS) [80], Ethereum private network running Proof-of-Authority (PoA) [78, 149], and Hyperledger Fabric with Practical Byzantine Fault Tolerant (PBFT) [96]. Afterwards, we provide a systematic review of the most common attacks and vulnerabilities for smart contract observed in Ethereum. And then, we evaluate the security and dependability of these components against a set of security and dependability attributes refined for blockchains.

In Chapter 4 we shift to methodologies, in particular focussing on blockchain consensus protocols. In this chapter we introduce METHUS, a novel framework and methodology to assess security and performance of blockchain consensus protocols. METHUS answers to the research question on how to assess performance and security of consensus protocols built for blockchain systems. It defines qualitative and quantitative approaches to

analyse performance, i.e., *throughput* and *latency*, and security, *i.e.*, *safety* and *liveness*, of consensus protocols assuming realistic deployment scenarios, such as a blockchain network deployed over the Internet. The qualitative approach evaluates consensus security through a CAP Theorem-based analysis [81], and performance through a study of the protocols' complexities. Conversely, the quantitative approach defines a systematic procedure for testing consensus protocols reproducing various adverse scenarios, such as network partitions and malicious nodes assuming a Byzantine behaviour. Finally, we validate METHUS proposing the assessment and comparison of three consensus protocols used in permissioned networks, such that two PoAs, namely AuRa [148] and Clique [156], and one PBFT-like protocol, namely Istanbul BFT (IBFT) [117].

In the last part of this thesis, we introduce two novel benchmarking procedures. In Chapter 5 we design and implement PETHARD, a performance benchmarking framework for Ethereum and Algorand consensus protocols. PETHARD adopts the quantitative analysis we defined with METHUS, providing a tool for benchmarking both PoW and PPoS consensus protocols. However, even though METHUS aims at defining a standard methodology that can be applied to any type of blockchain consensus protocol, the experimental evaluation of security for protocols built-in permissionless settings remains challenging. These systems are indeed widely decentralised and the attack surface is much greater than what METHUS simulates in a private instance, a fair evaluation requires deepening the analysis of the security and dependability in permissionless settings and we leave it for future works. For this reason, PETHARD only focusses on the performance evaluation of PoW and PPoS, comparing their *throughput*, *latency*, and scalability under various workloads and network sizes.

Although PETHARD allowed us to compare performance of PoW and PPoS algorithms, the analysis outlined some limitations of the tool. PETHARD only targets consensus protocols performance, however, to obtain a fair performance analysis of such systems, we need to extend the study to all blockchain components. Indeed, PETHARD's evaluation outlined that the nodes' configuration parameters strictly impact performance, and misconfigurations may create measurement unbalances. Therefore, PETHARD only works with private testbeds in which the System Under Test (SUT) is deployed on a single host through container virtualisation. However, this deployment can only reflect the behaviour of a LAN network.

Hence in Chapter 6, we move one step forward in the space of blockchain benchmarking introducing PERSECUS, a comprehensive dependability benchmark to evaluate performance and security of blockchain systems and consensus protocols. PERSECUS implements the concept of *dependability benchmark* [5, 186], a benchmarking standard, to propose a general benchmarking approach for blockchain systems. Its main goal is to overcome the limitations that emerged with state-of-the-art blockchain benchmarks, which have led to results hard to be aligned [200]. PERSECUS is a *flexible* and *extensible* tool that can evaluate any blockchain platform simulating various operating

conditions. In particular, it fosters accurate experimental measurements providing (i) optimised blockchain setups through network emulation and flexible configuration of nodes parameters; (ii) faultolads simulation, such as adverse network conditions and malicious actors; (iii) efficient workload generation; (iv) efficient metrics computation.

We conclude this work in Chapter 7 and Chapter 8, which sum up the thesis and discuss the ongoing works and future research directions.

Contributions

The contributions of this thesis can be summarised as follows:

- we provide a taxonomy of security and dependability properties and an assessment of the most prominent blockchain platforms and consensus protocols; then a taxonomy of the vulnerabilities and attack vectors for the Ethereum smart contracts, and an evaluation of their impact on the application's security;
- we propose METHUS, a framework and methodology to assess blockchain consensus protocols under realistic network scenarios, defining both qualitative and quantitative approaches. The qualitative approach undertakes a CAP theorem-based method to evaluate security, and a time complexity method to measure performance, whereas the quantitative approach defines a systematic experimental evaluation method; as a case study we provide a comparison of two PoA consensus protocols, namely AuRa and Clique, with an implementation of the classic PBFT protocol, namely IBFT;
- we propose PETHARD a framework to benchmark the consensus protocols used by the Ethereum and Algorand public blockchains, i.e. PoW and PPoS; hence, we provide a prototype that reproduces a private instance of both platforms and measures their performance and scalability;
- we propose PERSECUS, a comprehensive tool to benchmark the security and performance of blockchain systems and evaluate possible vulnerabilities and bottlenecks caused by different architectural and infrastructural components. It provides optimised procedures to stress test private blockchain networks under varying configurations, network conditions and workloads. As a case study, we provide a comparative evaluation of two Ethereum blockchain platforms, namely Parity and GoQuorum, and we simulate an Internet-based deployment.

Chapter 1

Background

1.1 Distributed Computing

In the last decades, computer systems evolved from centralised architectures, in which single processes handle operations, to distributed systems, in which a set of interconnected processes cooperate and share resources to achieve a common goal, i.e., *distributed computing*. Assuming the traditional *client-server* paradigm of computer systems, in distributed computing the *server* is represented as a network of separate computers. Such a network appears as a single entity to the *clients*. When a client sends its request, the distributed network cooperates and synchronises its actions to serve the request.

Broadly, in the field of distributed computing, the underlying physical systems can be summarised in two main abstractions i.e., *processes* and *links*:

- *Processes*: represent a computer, a CPU, or simply a thread. Processes cooperate on some common task;
- *Links*: abstract the physical and logical network, or communication channel, which supports interactions and the exchange of messages between processes.

To achieve the common goal and cooperate as a unique entity, processes run a *distributed algorithm*. i.e., a sequence of steps and operations carried on by each process. At the end of the execution, processes agree on the actions to carry out, and their order. This form of agreement is the basis of a fundamental technique used in distributed computing for the achievement of fault tolerance in replicated systems, and it is called *total order broadcast*.

When processes run a distributed algorithm, several properties need to be satisfied for all possible execution of the algorithm, to guarantee the correct execution of the common task. Most of the times such properties fall into two classes: *safety* and *liveness* [109].

Safety. The algorithm should not do anything wrong during its normal execution. It prevents unwanted operations, and ensures properties like *consistency*, *integrity*, *validity*, and *agreement*. If safety is violated at some point in time, it will never be satisfied again after that time;

Liveness. This property ensures that eventually something good happens. It ensures that sooner or later any execution of the algorithms correctly terminates, and includes properties like *availability* and *termination*.

In distributed computing, the algorithms that regulates the operations must be implemented as an extension of the *total order broadcast* (or *atomic broadcast*) primitive [49]. Total order ensures that processes agree on the same order of operations, while guaranteeing safety and liveness even in presence of faults. The atomic broadcast has been defined in [52, 87] as a broadcasting protocol that satisfies the following properties:

Validity If a correct process p broadcasts a message m, then p eventually delivers the message m;

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

Integrity. No correct process delivers the same message more than once; moreover, if a correct process delivers a message m sent by a correct process p, then m was previously broadcasted by p;

Total Order. For messages m_1 and m_2, suppose p and q are two correct processes that deliver m_1 and m_2. Then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.

In large distributed systems, processes may fail or behave maliciously, and moreover, communication links may be subject to desynchronisation and breakdowns. Ensuring the atomic broadcast protocols in all possible configurations is challenging. In the last decades, many implementations of the atomic broadcast protocol have been proposed. Those operate in various network conditions characterised by different synchronisation assumptions. *Cristian et al.*, [49] firstly introduced a new family of atomic broadcast protocols able to tolerate general failures, relying on timeouts of synchronised processes. Subsequently, in [32, 34] the authors address the problem of atomic broadcast removing timing assumptions. Conversely, *Ramasamy et al.*, [158], propose an optimisation of the previous results by replacing the atomic broadcast primitive and reduce the message complexity.

1.1.1 Centralised and Decentralised Systems

A computer system can be defined as a network of interconnected processes, or *nodes*, forming a graph, where the edges of the graph are the communication links FIGURE 1.1 [15]. For the rest of this thesis the terms *process* and *node* are used interchangeably. The control of traditional computer systems is usually *centralised*. Service providers offer system infrastructures where users can deploy their services. Centralised system may be *distributed*, typically to enhance fault tolerance, performance, and scalability. For instance in *Cloud Computing*, providers with large data center take advantage of resources distribution to offer users with reliable and efficient infrastructures, *i.e.*, *Infrastructure-as-a-Service* (IaaS). Despite the advantages of distribution, the control of the infrastructure is centralised. In Cloud-based systems, users need to entrust service providers to handle their data and services. Therefore, *trust* results as a fundamental requirement in centralised systems. Conversely, in a *decentralised* network, the components of a distributed system are under control of different authorities that not necessarily trust each other [185]. Decentralised systems do not act as a unique large server, as in distributed computing, but each node is an independent entity. The whole decentralised network acts as a system fulfilling tasks. Each task is accomplished by exchanging messages between two (or more) nodes; the exchange of messages is usually carried by special nodes called *proxy*, or *relay* [113]. Examples of prominent decentralised systems are: Tor [54], a decentralised network of anonymous relays, and a decentralised, privacy preserving name service infrastructure based on a DHT, and Bitcoin [138] the first blockchain application providing a *decentralised ledger technology* (DLT) for anonymous, peer-to-peer (P2P) payment transactions.

Summarising, a computer system can be classified as follows:

- *centralised*: a central node represents the whole system infrastructure; controlled by a single authority; the central node results in a *single-point-of-failure*, failures of the central node lead outages in which data and services may be inaccessible, tampered, or stolen;
- *distributed*: a system of spatially separated nodes connected via communication links; nodes co-ordinate on the execution of a certain task by exchanging messages; a single authority controls the whole system;
- *decentralised*: a distributed system in which nodes are in control of multiple authority that not necessarily trust each other [185].

The correct execution of task in a distributed system is strictly related to the concept of *time*. Atomic broadcast protocols usually trust on time to synchronise the nodes' operations. Typically, atomic broadcast protocols are built on a set of time assumptions,

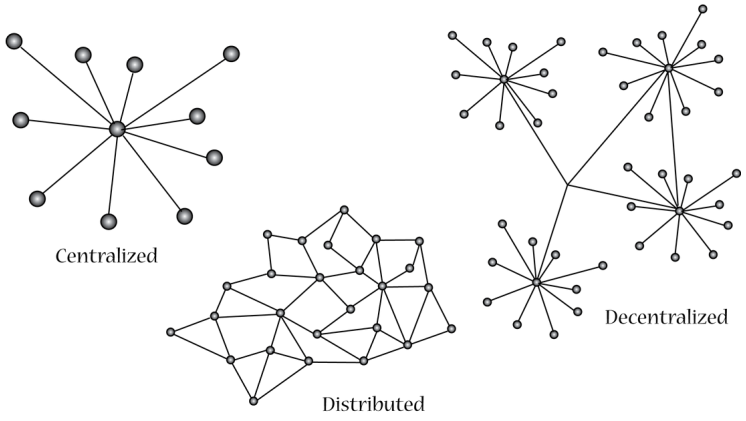


FIGURE 1.1: *Centralised, Distributed, and Decentralised systems*

e.g., finite time bounds on communication delays or synchronisation based on physical clocks and timeouts. Under such assumptions, the atomic broadcast ensures systems' safety and liveness properties throughout the protocol execution.

In so called *synchronous* networks, the upper bounds are fixed and defined *a priori*. In this setting distributed processes take advantage of synchronous communication and computation for atomic broadcast. However, synchronous solutions can be only applied in local deployments like LANs. In a real system deployments, nodes are spatially separated and communications unreliable. Networks can be partitioned, messages may be delayed or lost. Thereafter, in a distributed system, the nodes of the network may be subject to faults, or be subverted by an attacker. In this scenario, the atomic broadcast protocols cannot entrust on timing assumptions.

Real networks are indeed *asynchronous*, clocks are not synchronised and the nodes may have different perceptions of time. *Fischer et al,* [70], proved that in asynchronous networks, it is impossible for a distributed algorithm to terminate in a bounded time; this finding is also known as '*FLP Impossibility Result*'.

The most accepted network model relies on the concept of *eventual synchrony* [60]. It simulates an asynchronous network that '*eventually*' becomes synchronous, after an undefined time bound. Once the network returns synchronous, all messages are guaranteed to be correctly delivered. Under this network it is possible to design robust atomic broadcast protocols that guarantee safety and liveness in presence of faulty nodes [60].

1.1.2 Failure Models

Processes and links that characterise atomic broadcast protocols are subject to faults and alterations that could impact the correct functioning of the system. Random communication delays, network partitions, process failures, and even adversarial attacks may drastically impact atomic broadcast properties leading to a violation of the safety and

liveness of distributed protocols. *Cristian et al.*, [49] proposed a classification of failures in several nested classes, so that the complexity of each algorithm can be measured by referencing the tolerated class of failures. Failures are classified according to processes and links components - *i.e.*, failures occur when a component does not behave correctly.

An *omission* failure occurs when a component never gives the expected output to a specified input. A *timing* failure occurs when a component responds to a request too early, too late or never. A *Byzantine* failure [49, 115] occurs when a component behaviour is completely arbitrary and unpredictable or when a component is subverted by an adversary and acts maliciously. Byzantine faults can be caused by an omission fault, a timing fault or if the component gives a different output from the one specified. An important subclass of this failure is the so-called *authentication-detectable* Byzantine failure, where any corruption of messages is detectable by using a message authentication protocol, e.g. cryptographic scheme based on digital signatures.

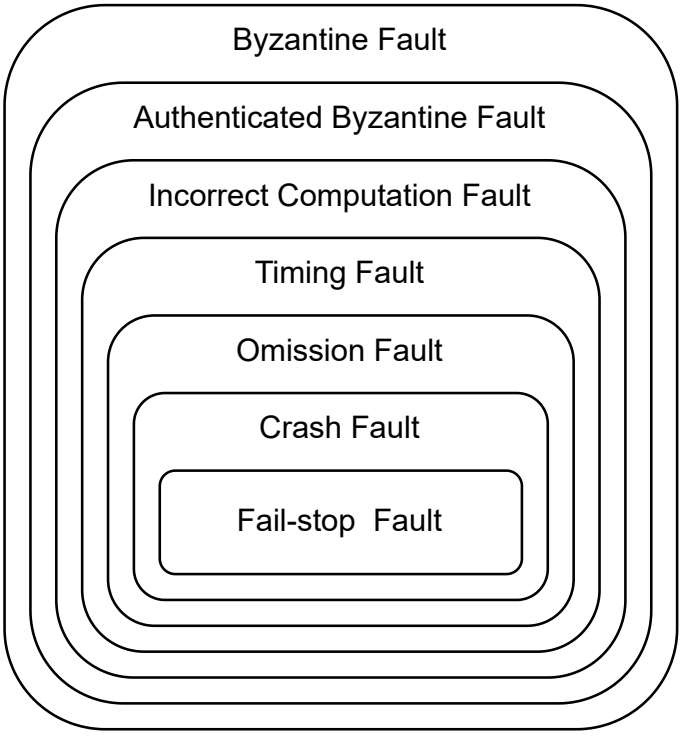


FIGURE 1.2: *Byzantine failure models*

FIGURE 1.2 shows the Byzantine Failures model. Crash failures, *i.e.*, a component stops working, are subclass of omission failures (*Fail-stop* failure is a special class of crash, where after a first omission a component omits to respond to all subsequent input events). Omission failures are a subclass of timing failures, timing failures are a subclass of authenticated-detectable Byzantine failures which are subclass of all the other failures, i.e. Byzantine failures.

1.1.3 Fault Tolerant Replication

In distributed computing the main challenge is to design robust atomic broadcast protocols which are resilient to possible faults or malicious behaviours. A robust and resilient atomic broadcast protocol must ensure safety and liveness under adverse operating conditions. In distributed systems jargon, we refer to this characteristic as *fault tolerance*. Fault tolerance is achieved through *replication*, *i.e.*, distribution of data and computation across several nodes also called *replica*s. A distributed system with n processes is said, f fault tolerant, if it satisfies the *trust assumption* such that up to $f < n/k$ nodes become faulty, for some $k = 2, 3, ...$ The remaining $n - f$ processes are said *correct* or *honest* [171]. Replication mainly refers to: (i) *data replication*, used in distributed databases in which data is stored across several replicas, (ii) *computational replication*, in which replicas execute the same computing task many times.

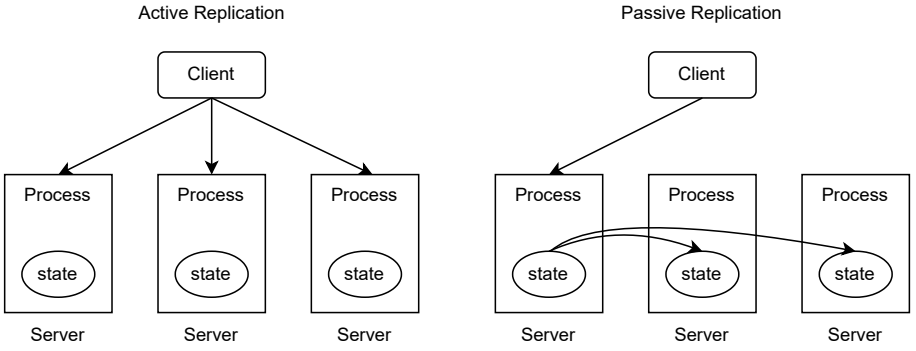


FIGURE 1.3: *Active and Passive Replication schemas*

FIGURE 1.3 depicts two different strategies that can be used for replications, such as:

- *active replication*: replicated processes execute the same request;
- *passive replication*: each request is served by one process, called *primary* or master, and the result is then dispatched to the other replicas.

The simplest passive replication schema is the *primary/backup* (also called *master-slave*) where a primary replica is in charge of processing the requests, updating the state of all the backup replicas and sending back the response to the client. On the other hand, active replication is implemented with the *multi-primary* schema (or *multi-master*), in which requests are processed by all the replicas. Active replication was firstly introduced by *Laslie Lamport* in [113] under the name of *State Machine Replication* (SMR) and then elaborated by *Fred Schneider* in its technical paper [171]. The SMR replicates a deterministic state-machine across the processes of a distributed system. Correct processes start in the same state and are guaranteed to execute the requests in the same order; for a given input the SMR produced a single output which is the same on any

replica. To ensure determinism, replicas agree on the same state by running a *consensus* protocol relying on the atomic broadcast. An SMR system is called f fault-tolerant if, in the case of f faulty nodes, the remaining $n - f$ honest nodes produce the same output. The value of f is determined by the type of network where the SMR operates, and the type of Byzantine faults the SMR tolerates.

Failure Mode	Synchronous	Asynchronous	Partially Synchronous
Fail-Stop	f	∞	$2f+1$
Omission	f	∞	$2f+1$
Byzantine Authentication	f	∞	$3f+1$
Byzantine	$2f+1$	∞	$3f+1$

TABLE 1.1: *Minimum number of correct nodes for which an f-resilient consensus protocol exists*

TABLE 1.1 shows the trust assumptions required to achieve consensus in the failure and synchrony models. All those algorithms can be categorised in two families: (i) *Crash Tolerant* and (ii) *Byzantine Fault Tolerant* (BFT). The SMR approach was firstly studied for synchronous systems relying on processes synchronisation using clocks and timeouts. In presence of Byzantine failures a number of consensus protocols have been proposed that tolerate up to $2f + 1$ faulty processes [49, 111, 113, 114, 153, 171]. However, in a real deployment where processes are geographically spread over a network, no assumption can be made on the synchrony [70]. The partially synchronous model have been adopted in the last decades to address the problem of consensus in real SMR systems. In this setting, the most prominent implementation of crash tolerant algorithms are *Paxos* [110, 112] and *Viewstamped Replication* (VR) [142] who tolerate up to $f < n/2$ faulty processes. In Byzantine settings where nodes may be subverted by an adversary and act maliciously against the common goal, the most prominent consensus protocol is the so called *Practical Byzantine Fault-Tolerance* (PBFT) [37]. PBFT extends of the Paxos/VR family. It uses a single-leader view progress and a three phase commit. It tolerates up to $f < n/3$ Byzantine processes, which is proved to be optimal [37].

1.2 Efficient and Secure Systems

1.2.1 Performance

With the growth of complex applications, there is an increasing need for highly efficient computer systems. To this extent, the assessment of their performance is a fundamental activity. Even though there is not a standard methodology for measuring performance,

there exist standard operational metrics that must be referenced when assessing performance in computer systems [26, 71]. In particular, performance measurement techniques fall in two categories: *system-oriented* and *user-oriented* measures. The former measure how a target system behaves during a certain workload, whereas the latter measures how efficiently users' requests are processed. System-oriented metrics typically fall in the categories of *throughput* and *utilisation*.

Throughput. It is defined as the average number of jobs (e.g. transactions, processes, requests) processed per unit of measured time. Throughput is used to measure the system capacity based on a certain workload by raising such load until the system saturates: *max-throughput*;

Utilisation. It is a measure of the fraction of time that a particular resource is busy. Utilisation metrics proactively affect the performance of a system, these are mainly identified with *network utilisation* and *CPU utilisation*. The former measures the network usage, the latter measures the amount of time the CPU is busy processing during a specific interval and load.

The user-oriented performance is mainly related to the elapsed time from the initiation of a new job till the response the system returns to the user. User-oriented measures typically include *response time*, *turnaround time*, and *latency*.

Response Time. It is the time an interactive system takes to respond to a user or application input. It affects the elapsed time between the actions through the system, impacting the productivity and the performance;

Turnaround Time. It is the total time the system needs to complete all the processes required by a specific job. This metric does not include the time a request needs to reach the server and the response to reach the user;

Latency. It is the time delay between a job request and its competition in the system. Latency strictly depends on the characteristics of the system observed, broadly it is caused by the limited capability of networks and the way a system manage incoming requests.

1.2.2 Security and Dependability

In the previous sections we observed that, properties of safety and liveness are pivotal for the correct execution of distributed protocols. In computer systems, these properties are generalised with the concepts of *security* and *dependability*. Dependability includes a set of attributes that identify the reliability of a system during its execution [12, 13]. Dependability is strictly related to security, and they are commonly referred together

as *Dependability and Security*. FIGURE 1.4 summarises the relationship between the dependability and security concepts.

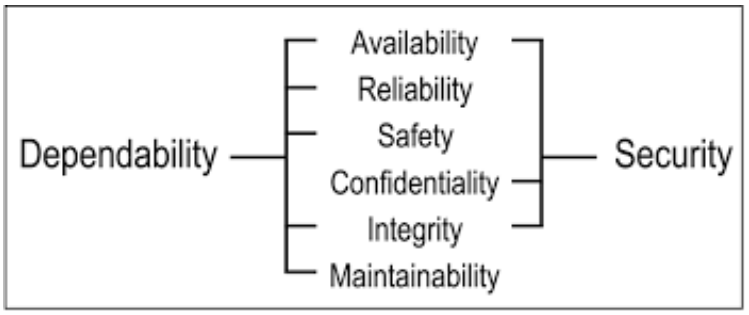


FIGURE 1.4: *Dependability and Security attributes*

The attributes of dependability represents the main properties of a system performing in presence of system and network. Following, the main attributes of dependability:

- *Availability*: willingness of the system to run correct services without interruptions;
- *Reliability*: ability of the system to run correct services without any interruption or failures;
- *Safety*: absence in the system of catastrophic consequences on the users and the environment;
- *Maintainability*: ability of the system to undertake modifications and repairs;
- *Security*: ability of the system to provide *confidenciality* (*i.e.*, absence of unauthorised leaking of sensitive information), *integrity* (*i.e.*, absence of improper system alterations from unauthorised users) and *availability*, which are also referred to as *CIA Triad*.

1.2.3 Scalability

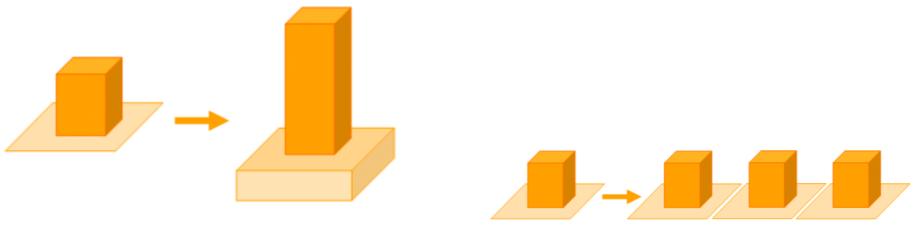
The *scalability* plays a central role in the process of assessing performance of a distributed computing system. It measures the workload a system or application can effectively handle while maintaining an adequate performance level [24, 121]. If a system cannot handle a growing load, it reached the limit of scalability, and we said that the system saturates. There exist two approaches to avoid a system to saturate: increase the hardware resources, distribute computation across more processes, *i.e.*, *parallel computing* [4]. Scalability can be summarised with two approaches as showed in FIGURE 1.5:

- *Vertical Scaling*: it is the action of incrementing or reducing the hardware resources of one (or more) components of the system (e.g. more CPUs or memory), to make

the system more performant and able to handle higher workloads. Vertical scaling improve performance of a system, as long as the underlying software can benefit from powerful resources, despite periods of downtime caused by physical/logical hardware changes;

- *Horizontal Scaling (or Parallelisation)*: it is the action of incrementing or reducing the number of components (e.g. processes, computers, VMs or containers) in a distributed system. In this way, the system leverages more resources and consequently higher workloads can be handled. However, horizontal scaling often implies communication overheads, especially in large-scale distributed systems. This may impact the expected performance gain, and in some scenarios even lead to performance degradation.

There are trade-offs between the two approaches. In the process of assessing applications or computing systems, both approaches should be evaluated to identify the optimal configuration for the expected workload. Although, vertical scaling is usually preferred to horizontal scaling because of its cost-effective deployment, with the advent of virtualisation this difference in prices has been blurred [121].



(A) Vertical Scaling (B) Horizontal Scaling

FIGURE 1.5: *Vertical and Horizontal Scaling approaches*

1.3 Assessment of Distributed Systems

1.3.1 Virtualisation Techniques

The assessment of a distributed system is a fundamental activity. It is required to evaluate how a system behaves under different network conditions and workloads [5, 186]. Usually, the target system is also referred to as SUT. The procedure for assessing a SUT can be summarised as follows:

1. stress the system with a predefined workload;
2. observe performance and measure system-oriented and user-oriented metrics as defined in Section 1.2.1;

3. evaluate resiliency measuring the security and dependability attributes defined in Section 1.2.2.

Usually, distributed systems are deployed over the Internet in large networks spread around the globe. For testing purposes, reproducing such a realistic deployment in a testbed is a time-consuming and expensive activity. In addition, modern systems must be designed to cope with the increasing workloads and networks. Evaluating them in real distributed infrastructures under varying configurations and requirements requires large testbeds. However, the deployment of large scale distributed computing testbeds is tedious, time-consuming and costly. Moreover, testers usually cannot reproduce such an environment due to limited financial and timing resources. For this reason, there exist alternative ways to represent distributed software infrastructures in a testbed [71]. A distributed system can be reproduced in a testbed following three different approaches:

- *Simulation*: the behaviour of a distributed system is simulated using single-threaded simulation [29, 35]. It recreates a large scale network of heterogeneous resources on a single server domain and is managed by a single entity. These tools can model a distributed infrastructure with different capabilities and configurations, for instance, they can be used to model clusters [28], multiprocess grids [72] and P2P network topologies [146]. Simulation tools enable developers to benchmark performance of distributed services in a repeatable and controllable environment and fine-tune their performances for service optimisation. CloudSim [35] and GridSim [29] are two of the most prominent simulation toolkits that provide facilities for the modelling of resources and networks and support primitives for application deployment, evaluation and execution over such models. The former simulates large scale testbeds of interconnected resources, whereas the latter provides simulation functionalities for cloud computing environments [127] in which computing resources are managed by third-party organisations;

- *Hypervisor Virtualisation*: hardware virtualisation (or *hypervisor-based virtualisation*) enables the creation of virtual versions of hardware resources such as computers, storage and network links. This technique is characterised by Virtual Machines (VMs) running their OS on top of a host machine, in a completely isolated execution context. Virtualisation software are called *hypervisors*, and the most prominent solutions are popularised by Xen [195], VMware [187] and KVM [119]. The main benefits of virtualisation include hardware independence, control of resources and isolation. Virtualised systems offer a repeatable and fully controllable environment.

- *Container Virtualisation*: operating-system-level virtualisation (or *container-based virtualisation*) is a lightweight alternative to hardware virtualisation. This technology works at the OS level providing an abstraction of single processes instead

of the whole guest OSs. The containerisation technique partitions resources in a way to create multiple isolated user-space instances, i.e. *containers*. Containers share a single OS kernel, however, programs running within a container can only see the container's content and can not interact with the outside. The isolation of container images is achieved by Linux kernel namespaces [19] while the resource management is normally done by Control Groups (cgroup) [128]. The main implementations of container-based virtualisation for Linux distribution are Linux VServer [120], OpenVZ [143] and Linux Containers LXC [118]. Docker [56] and Kubernetes [106] are easy to use tools enabling the deployment and management of containerised applications.

Even though hypervisor virtualisation is appropriate for many usage scenarios of distributed computation infrastructures, many studies on performance virtualisation show that the VMs virtualisation leads to performance overhead, especially for I/O operations [160, 191]. There exist scenarios in which high efficiency and performance are primary requirements, for example in High Performance Computing (HPC) systems. By contrast, container-based virtualisation represents a valuable alternative offering a lightweight layer of virtualisation with improved performance. Applications in a container are more portable, since can be bundled into a single container and deployed in various environments, can be easily deployed and can run hundreds of different instances over a single host OS. FIGURE 1.6 shows the architectural differences between container-based and hypervisor-based virtualisation.

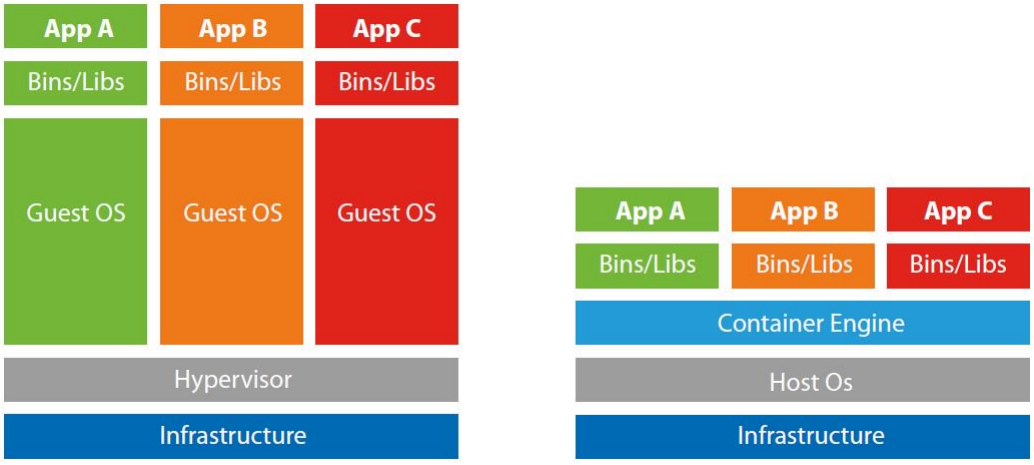


FIGURE 1.6: *Hypervisor Virtualisation and Container Virtualisation*

1.3.1.1 Network Emulators

In virtualisation, both VMs and containers are coordinated and communicate through virtual networks. Usually, virtualised networks are deployed under the same server domain in a local testbed, leading to ideal virtual LANs with almost zero latency and

faults. In general, distributed applications and protocols need to be evaluated and monitored under critical network conditions like the Internet, making virtualised settings with high reliable networks unfeasible. For this purpose *network emulation* tools can be used for the simulation of more realistic network environments. Network emulation is a technique that enables, with software components, the simulation of certain network behaviours like random delays, packet loss and packet distribution. Emulation tools enable the configuration of such parameters to define various network setups for testing purposes. Parameter configuration is offered by network emulation tools as NetEm [89], NIST Net [36] and DummyNet [162]. These are the most prominent tools sharing almost the same design.

- *NetEm*: it is based on the Traffic Control (TC) module of the kernel Linux. It allows developers to operate at the kernel level to manage traffic facilities by acting on delays, packet loss, duplication and other scenarios. It is mainly characterised by two portions, a small kernel module for a queuing discipline and a command-line utility (but also a graphic interface is provided) for its configuration. NetEm main motivation was to provide a way to reproduce long-distance networks in a lab environment [89].
- *DummyNet*: it is a network emulator applicable to any existing protocol stack. It intercepts communication toward the targeted protocol layer simulating the presence of a real network with limited queues, bandwidth limitations, communication delays, and packet loss. Dummynet gives the advantages of real-world testing in a simulated environment thanks to great control of operating parameters, simplicity, integration with real traffic generators [162].
- *NIST Net*: it is another Linux kernel-based network emulator, very similar to NetEm. NIST Net emulates common real-network behaviours as packet loss, duplication or delay, traffic congestion and bandwidth limitations on ingress traffic, requiring only commodity computer hardware and OS. Nowadays most of NIST Net functionalities have been integrated into NetEm.

Recent studies [123, 140] compared the performance and accuracy of network emulators. The authors claim that NetEm achieves more accurate results with respect to the other emulators, in particular with respect to bandwidth, constant and variable delays emulation, and scalability.

1.3.1.2 Chaos Testing Tools

Testing systems and applications resiliency in adversarial conditions enables the teams to learn where apps fail before the customer does. A system can be defined as resilient if the properties above are respected in a certain context. The growing popularity of

distributed and virtualised architecture makes resilience testing critical for applications that now require a large number of operations. Resilience testing is an approach where you intentionally inject different types of failures at the infrastructure level (VM, network, containers, and processes) and let the system try to recover from these unexpected failures that can happen in production. This approach is called *Chaos Engineering* [155] and is used to uncover systemic weaknesses. Simulating realistic failures is the best way to enforce highly available and resilient systems. Following, we present two prominent chaos testing tools, namely *Chaos Monkey* and *Pumba*. Both tools can be used in both hypervisor-based and container-based infrastructures.

- *Chaos Monkey* [139]: it is a testing tool for cloud-based virtualised infrastructures proposed by Netflix. It enforces failures via the pseudo-random termination of instances and services within a certain architecture deployed on Amazon Elastic Cloud Computing (EC2).
- *Pumba* [57]: It is a chaos testing tool that applies at the container level. It connects to every Docker daemon running on some machine (local or remote) and brings a level of chaos to it. Pumba can simulate different failures in the system by (randomly) killing, stopping, pausing and removing running containers. Pumba is built on top of the NetEm network emulator and inherits some of its features like simulating different network failures like delay, packet loss/corruption/reorder, bandwidth limits and more.

1.3.2 Workload Generator

A fundamental component in the assessment of complex computing systems is the generation of workloads traces. Running multiple load-tests can help discover performance issues or network bottlenecks in a system. Workload generators recreate realistic load traces by simulating various users connections.

The main characteristics of workload generators are:

- *Architecture*: a workload can be generated from a centralised architecture, where a single instance of the workload engine runs on a single machine or a distributed architecture where the engine runs on multiple machines. Even if the centralised architecture is easier to deploy, it cannot simulate real traffic where the load comes from different users spatially separated. To reach adequate levels of workload intensity, workload engines can be spread over several machines and run specific load tasks according to a master-slave coordination protocol.

- *Performance metrics collection*: usually workload engines provide their proper collection of measured performance metrics during the workload generation. Collection and storing of such metrics should be conducted with a customisable granularity and such that there is no impact on performance of the target system.
- *Workload trace type*: the workload generator should provide the ability to easily create and execute synthetic and/or real workload traces.
- *Interaction with the target system*: the majority of workload generators provide the possibility to interact with the target system toward HTTP requests. Only a few available tools permit the generation of load against the wide range of distributed storage systems.

Workload Generator Tools

Following we present the most prominent workload generator tools, their main features and differences:

- *Apache JMeter* [97]: it is an open-source workload generator fully written in Java programming language. It is mainly used for load testing and performance assessment of Web-services, Databases, FTP Servers and other resources. It relies on a full multi-thread engine able to generate concurrent threads of requests toward a target system. JMeter provides a GUI for facilitating the configuration of performance tests as well as the analysis of test results. Furthermore, it provides documented APIs for extensibility purposes and the development of customised plugins. It can be deployed either in a centralised or a distributed master-slave architecture when the desired workload requires more connections than the ones supported by one single machine. It provides the ability to collect metrics from the system under test and generate reports with measurements.
- *Rain* [17]: it is a statistic-based workload generation toolkit that uses parameterised or empirical probability distributions to mimic different classes of load variations. Rain provides three workload generation classes (open-loop, closed-loop and partly-open loop), that can be used by user-defined ad-hoc request generators targeting new systems and applications. Request generation and request execution are separated processes that produce traces that can be consumed by more performant load-delivery clients. With this separation, Rain offers more flexibility and adaptability to the generated workloads.
- *Locust* [183]: it is a distributed load testing tool fully written in Python, for websites and other systems. It is completely event-based and can run thousands of concurrent users on a single installation. Locus can run a swarm of requests and monitor the behaviour of a system through a real-time web UI.

- *Httperf* [136]: it is a flexible load tool for generating HTTP requests and measuring web server performance. Workloads can be configured in a way to simulate different user connections and request rates. Httperf does not support a distributed architecture and cannot be used with real workload trace.

- *Faban* [77]: it is a performance and load testing tool that supports multi-tier server benchmarks such as Apache httpd, Sun Java System Web, Portal and Mail Servers, Oracle RDBMS, mem-cached benchmarks. Faban permits building and modifying realistic workload traces starting from a log file. Due to its distributed and scalable architecture, Faban is well suited for generating cloud computing workloads. However, it does not provide high flexibility and its integration with other systems is not easily developed.

- *Tsung* [184]: (or *IDX-Tsunami*) it is a distributed load testing tool fully protocol-independent, that can be used to stress HTTP, SOAP, PostgreSQL, MySQL, LDAP and Jabber/XMPP servers. Tsung is developed in Erlang[1], a concurrency-oriented programming language, and inherits several characteristics from Erlang like high performance, scalability and fault-tolerance. Tsung strength is its ability to simulate a large number of simultaneous users even on a single instance. When used in a distributed mode, it is possible to generate a huge workload intensity even with a modest generator cluster. However, it does not provide high flexibility and its integration with other systems is not easily developed.

1.3.3 Benchmarking Tools

Benchmarking tools can be used for the assessment of performance and scalability of a particular system or application. Typically, such tools are characterised by a web interface for test monitoring, integrated with some workload generator that creates artificial session-based request loads to the system under test. Examples of currently available benchmarking tools, along with their main features are the following:

- *RUBiS* [179]: it is an auction site prototype modelled after eBay.com that is used to evaluate application design patterns and application servers performance scalability. RUBiS implements the core functionalities of an auction site (i.e. selling, browsing and bidding) and distinguishes three user sessions (i.e. visitor, buyer and seller). It is a three-tier web application characterised by an Apache web server, a JBoss application server and a MySQL database server. This benchmarking tool provides a client that emulates users behaviour for various workload patterns and calculates statistics. The last update of the RUBiS project was in 2008, but it is still a valid solution when it is needed a complete web-based multi-tier application is used in testing and benchmarking activities.

[1] https://www.erlang.org.

- *TPC-W* [129]: this tool has been proposed by the Transaction Processing Performance Council (TPC) [48] as a benchmark for web servers and databases. It specifies an E-commerce workload that simulates the activities of a retail store website. The system simulates three different interactions: searching, browsing and ordering items. The performance metric reported is the number of web interactions processed per second. It was declared obsolete in 2005. Although, it is still being used by the research community.

- *CloudStone* [174]: it is a multi-platform, multi-language performance measurement tool for Web 2.0 and Cloud Computing, developed by the Rad Lab group at the University of Berkeley. CloudStone is a toolkit that provides a set of automation tools for workload generation and performance measuring of an open-source Web 2.0 social application (*Olio*. The application metric is the number of active users of the social networking application, which drives the throughput or the number of operations per second.

- *VMmark* [93]: is a benchmark tool suite proposed by VMWare [187] for measuring performance, scalability and resource consumption of applications running under hypervisor-based virtualised infrastructures. It runs several workloads simultaneously on VMs, each of them configured according to a template that mimics typical software applications found in corporate data centers. The default templates are provided: email servers, database servers, and Web servers. VMmark collects relevant performance metrics such as commits per second of a database server, or page access per second for web servers. VMs are grouped into logical units called *tiles*. VMmark first calculates a score for each tile, based on performance statistics produced by each VM, and aggregates the per-tile scores into a final number.

- *YCSB* (Yahoo! Cloud Serving Benchmark) [46]: it is a framework for evaluating and comparing the performance of multiple types of cloud data serving systems, including NoSQL stores such as Apache HBase, Apache Cassandra, Redis, MongoDB, and Voldemort. The framework is composed of an extensible workload generator client called YCSB client, and a package of common workloads scenarios to be executed by the generator for evaluating the performance of different data stores. Performance is mainly measured in terms of throughput and latency. Although widely used to compare performance of different systems, the YCSB workload generator component permits to generate load only in terms of a given number of operations that have to be executed in a given amount of time, or at a given fixed rate. There is no means to generate workload according to neither a synthetic nor a real workload trace.

Chapter 2

Blockchain Technology

Blockchain technology firstly appeared within the Bitcoin's cryptocurrency [138]. Nowadays, it is considered as the backbone technology for multi-party decentralised systems. It is characterised by a P2P decentralised network in which the nodes share a replicated data structure without a central authority. The replication approach used in blockchain is the *active-replication*, with a *multi-primary* scheme. The shared data structure is called *ledger* (or *blockchain*). The nodes update the ledger and agree on the same state by executing a consensus protocol. In Bitcoin, the consensus was called *Proof-of-Work* (PoW). In PoW the process of block creation is called *mining process* and is carried out by special nodes of the network called *miners*. The advantages of blockchain are manifold: (i) it distributes trust across a set of nodes leveraging a consensus protocol rather than a centralised authority, (ii) it enhances security: a decentralised system is harder to compromise, (iii) it enforces data availability and immutability, thanks to the replicated data structure. Moreover, its particular design enables P2P interactions between parties without relying on trusted authorities, *e.g.* the role of banks in traditional payment transactions.

Blockchain disrupted the field of distributed computing with the advent of *smart contracts*, *i.e.*, self-executable programs that run on the blockchain. Pioneered by the blockchain platform Etherem [45, 194], smart contracts are nowadays the foundational technology behind *decentralised applications* (dApps).

2.1 Key Terminology

2.1.1 Ledger Data Structure

The blockchain data structure is a distributed ledger of consecutive chained blocks. Each block is linked with the hash of the previous. Blocks contain a list of records that

witness transactions that occurred among participants (see FIGURE 2.1). Participants are usually referred to as *accounts*. Accounts are cryptographic identities that have access to and interact with data the blockchain. Usually, accounts hold some assets, *e.g.*, cryptocurrency, tokens, etc. Nodes update the data structure by appending new blocks following the consensus agreement. The ledger is an append-only data structure in which each block depends on its predecessors. This important characteristic ensures data immutability. Tampering with a block of the ledger generates an invalidation of all its following blocks. To cheat with the system, an attacker should compromise the majority of the nodes in the network, so that all agree on the new version of the ledger. In a decentralised system this is improbable, hence the ledger data structure is said *tamper-proof*.

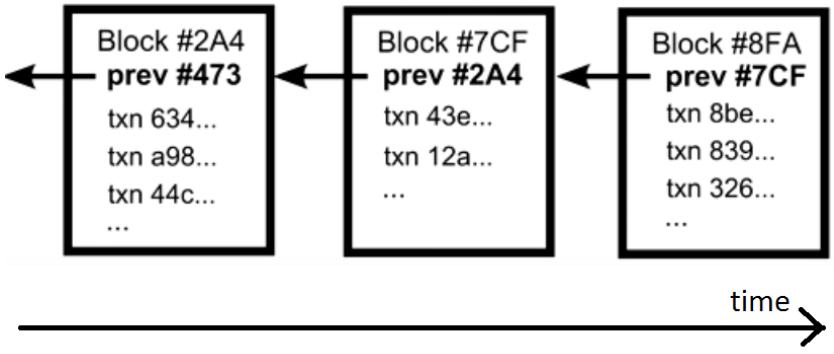


FIGURE 2.1: *Blockchain data structure representation*

Transactions. A blockchain transaction is a function executed on the blockchain that determines an exchange of value (*e.g.*, cryptocurrency, token, etc.) between a sender and a receiver. For instance, a Bitcoin transaction transfers the ownership of a certain amount of asset between a sender and a receiver account. Specifically, a transaction specifies a list of inputs recalling on previous *unspent transactions* (UTXO) [103] of the sender and a list of outputs that determines the transferred amount which must be equal to the UTXOs claimed by the inputs. The balance of an account is the total amount of UTXOs. This particular approach to transactions enables any user to verify the history of transactions spent from a particular account. Transactions are organised into the blocks with the *Merkle tree* data structure [131]. The leafs of the tree indicate the hash of transactions, while the root node is a unique representation of the transaction set. The Merkle tree root is then stored into the block's header together with the hash of its predecessor block.

2.1.2 Cryptography

Digital Signature Scheme. This cryptography primitive is used in blockchain to represent the accounts. Accounts are represented with a cryptographic key pair composed by a

public key (pk) and a private key (sk). The public key can be shared publicly, and it represents the unique address where the account receive the assets, while the private key is known only to the account owner. Public/private key pair are at the basis of the blockchain *digital signature scheme*, which performs important operations such as signing transactions and verifying signatures. Digital signatures are used to verify the validity of transactions and authenticate the sender with a unique and verifiable function [165]. A digital signature scheme consists of three cryptographic algorithms: (i) *key-generation (G)*, (ii) *signing (S)*, and (iii) *verifying (V)*. G use a random seed to generate a $< pk, sk >$ key-pair; S produces the signature of a transaction with the secret key sk; V it verifies the validity of a transaction by checking its digital signature with the sender's pk. The digital signature scheme is proven to be *existentially unforgeable*, *i.e.,* it is impossible for an attacker to cheat without having access to the key-pair $< pk, sk >$. Blockchains like Bitcoin and Ethereum use a particular digital signature algorithm, namely the *Elliptic Curve Digital Signature Algorithm* (ECDSA), which requires less computation while preserving security [98].

Hash Function. A collision resistant cryptographic algorithm used to uniquely represent data of arbitrary size as a string of fixed size, called *hash*. Hash functions are also called *one-way* function given their irreversible nature. The hash can be used to verify the integrity of data, but it is impossible from the hash to retrieve data. Blockchain uses hash functions to represent transactions and blocks, and to verify the integrity of the ledger. Moreover, the hash it typically used to encode the account addresses.

2.1.3 Smart Contracts

A smart contract is a computer program deployed on the blockchain. Smart contracts are executed collectively on the blockchain to process transactions according to a transparent, conflict-free, and secure logic without the need of third-party regulators. First generation blockchain like Bitcoin provides basic scripting functionalities that can be used to approve, reject, or program transactions between two accounts. Successively, the pioneering Ethereum blockchain introduced a Turing-complete programming language that enabled the implementation of complex code for more specific computation. Smart contract code is typically executed on each node of the blockchain within a virtual machine, *e.g.* the *Ethereum Virtual Machine* (EVM) [194].

2.2 Permissionless versus Permissioned Blockchains

Systems like Bitcoin and Ethereum are so-called *permissionless*, i.e. any node on the Internet can join the network and become a miner for the blockchain. In permissionless settings distributed consensus is achieved via the so-called *Proof of Work* (PoW), a lottery-based computational intensive mathematical challenge executed across the miners. The

miner who solves the challenge wins the lottery and is allowed to propose a new block. Conversely are called *permissioned* blockchains such systems where participants need to be authenticated to join the network and the access to information is limited under privilege constraints. Blockchain systems can be classified on the basis of access rules under different permission models. Participants of a blockchain network have rights of: *(i)* accessing data on the blockchain (*Read*), *(ii)* submitting transactions (*Write*) and *(iii)* running a consensus protocol ad updating the state with new blocks (*Commit*) [94]. The works proposed by BitFury and Garzik in [21, 22], defines blockchain systems based on these rights. Specifically, regarding Read operations a blockchain can essentially be divided into two classes:

- *public blockchain*: no restrictions applied on Read operations;
- *private blockchain*: a predefined list of entities is allowed to run Read operations.

The Write and Commit operations in turn identify other two classes of blockchain:

- *permissionless blockchain*: no restrictions on Write and Commit operations;
- *permissioned blockchain*: only a predefined list of entities is allowed to Write and Commit operations.

In other words, in permissionless blockchain, anyone can join the network and run Commit and Write operations. On the opposite side, nodes of a permissioned blockchain are known at the outset, thanks to authentication mechanisms, and only those authorised nodes can participate in Commit and Write on the blockchain network. Regardless of the identification process, a permissioned blockchain can be either public or private, according to whether only authorised nodes can execute Read operations. Table 2.1 shows a comparison between the identified models.

	Read	**Write**	**Commit**
public permissionless	anyone	anyone	anyone
public permissioned	anyone	authorised participants	all or subset of authorised participants
private permissioned	restricted to a subset of authorised participants	authorised participants	all or subset of authorised participants

Table 2.1: *Types of blockchain system*

Public permissionless blockchains, e.g., Bitcoin, operate in hostile environments and require the deployment of crypto-techniques to coerce participants to behave honestly. These crypto-techniques involve the usage of a cryptocurrency (e.g. ether on Ethereum)

to reward participants. Contrarily, *private permissioned* blockchains operate in environments where participants are authenticated. For this reason, permissioned blockchain can hold participants accountable for misbehaviour in ways that permissionless implementations cannot. Thanks to accountability, systematic violations can be detected over time and resolved optimistically. This is a substantial simplification, and permissioned systems benefit from fairness property that derives from it.

2.3 Consensus

Consensus is the core component of blockchain systems. It ensures total order on transactions and regulates the proposal of new blocks to be appended on the blockchain. Hence consensus directly impacts performance and security of modern blockchain protocols. Although in the field of distributed systems consensus has been widely studied (Section 1.1.3), with the advent of blockchain it has experienced a renewed interest. In the previous chapter we outlined the fundamental safety and liveness properties of traditional systems. Hereunder we refine those concepts introducing two new properties of safety and liveness which fit the context of blockchain:

> *Persistency.* If an honest node appends a block b to its local copy of the blockchain, then b will be eventually appended by any other honest node. Persistency ensures consistency among the nodes, and data integrity;
>
> *Termination.* Any valid transaction submitted to a correct node n will be eventually inserted into a block b, and successively b appended to the blockchain of every correct node.

There exist two approaches to consensus in blockchain, specifically:

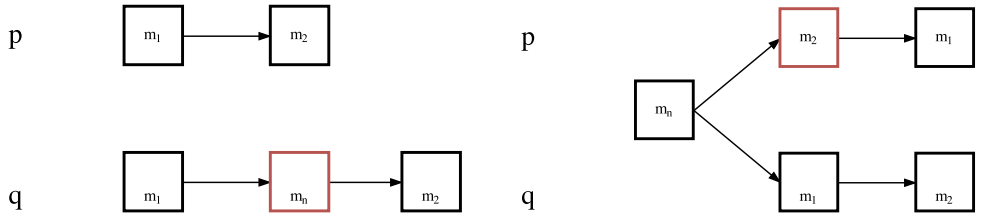
- *lottery-based*, whereby a randomly elected leader proposes new blocks on the chain;
- *voting-based*, whereby a voting mechanism is carried out to elect a new leader.

These approaches target different blockchain models. Lottery-based algorithms provide a probabilistic mechanism to elect new block proposes, also called *leader*, and are usually employed in public-permissionless blockchains. Indeed, these algorithms can scale on large networks without worsening performance. For instance, Bitcoin implements a lottery-based algorithm with the PoW. However, the election of a leader can be extremely expensive in terms of time and resource consumption, lowering the block production rate. Moreover, these algorithms allow multiple leaders simultaneously, with a certain probability, causing splits of the blockchain, i.e. forks. In the case of forks, the network

needs to re-synchronise on the same blockchain. For this reason, blocks are usually considered part of the blockchain only after a certain period in time. For instance, in Bitcoin, a block is considered part of the blockchain only when six further blocks are appended on its chain. These blocks are said *finalised*. The lottery-based approach differs from classical strong consistent consensus protocols, which implement total order broadcast and state machine replication (Section 1.1.3). In this sense, such protocols can only guarantee a sort of *eventual consensus*, where the forks that potentially arise are eventually resolved in the future. *Eventual consensus* is a fundamental concept in blockchain systems. It is often referred to as absence of *consensus finality*, whereby a valid block can never be removed from the blockchain once its block is appended to it [188, 189]:

> *Finality (or Consensus Finality)*: If a correct node n_i appends a valid block b on its local copy of the blockchain before block b', then no correct node n_j will append b' before b on its local copy of the blockchain. Final blocks cannot be removed from the blockchain.

Finality ensures strong consistency among replicas. It determines the possibility of having forks. By referencing traditional distributed computing properties (introduced in section 1.1), finality can be represented as the combination of *agreement* and *total order*. If one of the two is violated, then finality cannot be guaranteed. FIGURE 2.2 represent the scenarios in which two correct nodes p and q, violate *agreement* FIGURE 2.2a and *total order* FIGURE 2.2b. In the former case, p and q do not agree on the block following m_1, whereas in the latter case they reverse the order of blocks m_1 and m_2.



(A) No Agreement (B) No Total Order

FIGURE 2.2: *Fork scenarios that violate agreement and total order*

Both scenarios represent a fork, having two nodes with different views of the blockchain. This events violate finality. For the rest of this thesis we will analyse the finality property of blockchain assuming the following:

Theorem 2.1. *A blockchain system achieves finality if and only if it guarantees agreement and total order.*

Corollary 2.2. *If a blockchain system violates finality, it can only guarantee eventual consistency.*

Conversely to lottery-based, the voting-based approach is used in permissioned networks, where the number of nodes is fixed. Voting consensus protocols afford lower latencies than the lottery-based: as soon as a majority of nodes agrees on a block, the consensus is achieved, hence consensus finality is guaranteed. However, this advantageous characteristic comes at the cost of scalability. Voting-based algorithms typically require intense message exchanges to reach consensus. Therefore, the higher the number of nodes in the network, the higher the message exchanges is required. For this reason, in large-scale networks, performance may degrade due to long communication processes, hence scalability is not guaranteed. Voting-based algorithms implement traditional BFT protocols, which ensure high performance and consensus finality while remaining in small networks.

Due to the inherent scalability limits of BFT, many hybrid protocols have been recently proposed by the blockchain community. These aim to boost the power and applicability of blockchain platforms by overcoming both the scalability issues of BFT protocols and the performance issues of lottery-based protocols. These algorithms are dubbed collectively as *Proof-of-X* (PoX). PoX protocols can be based on either voting or a lottery approach. In the next chapter we introduce some of the most famous blockchain platforms and consensus protocols.

Chapter 3

Assessing Secure and Dependable Blockchain Systems: A Taxonomic Approach

Decentralised infrastructures offer the possibility to realise efficient and secure P2P collaborative infrastructures. The blockchain is the backbone technology for the realisation of secure and reliable decentralised applications. However, nowadays there is a wide range of blockchain solutions, and their adoption entails several infrastructure and architectural choices such as the use of either a permissionless or permissioned network and the underling *consensus* protocols. Notwithstanding its promising advantages, blockchain raises issues of security, scalability, and performance that undermine its deployment in real systems. Blockchain systems are distributed over large networks of nodes, presenting a broad attack surface, as each one of the participants can come under attack. A crucial aspect of such decentralised system is their chosen consensus algorithm, i.e., the underlying protocol that in each blockchain ensures the ordering of transactions on each replica. Typically, in a distributed system, the consensus protocols have been employed to ensure *dependability* and *security* properties of a system. This also applies to blockchains, in which trust must be ensured by design. To this extent, security and dependability are paramount characteristics to build reliable and secure blockchain systems. In the previous chapters, we introduced dependability and security, and we explained how their attributes can be generalised into two sets of security properties, dubbed respectively *safety* and *liveness*. State-of-the-art consensus protocols differ on the properties they afford to users and the assumptions they make on the underlying network model. Several works have been proposed in the literature to evaluate such differences, as *e.g.*, [33, 173, 189, 197], although a fair comparison is elusive due to several contrasting assumptions. Consensus is a paramount component of blockchain systems, since it strongly affects not just security, but also scalability and performance.

As a matter of fact, there currently is no optimal approach, as each of the existing protocols offers a suboptimal tradeoff between all desired properties. Fair comparisons of these protocols are today needed to assess their properties and consequently foster their adoption. In this context, the evaluation of security and dependability is a primary need. To assess them, we need to establish a common knowledge and foundational definitions that can be used as milestone for further analysis.

In this chapter, we face the problem of assessing security and dependability of blockchain systems. We define a set of foundational properties that determines the security and dependability of a blockchain, then we structure them into a general taxonomy. The taxonomy evaluates blockchains under three dimensions, i.e. *platforms*, *consensus*, and the *smart contracts*. For the analysis of *platforms*, we focus on five prominent blockchains that cover various consensus protocols, namely *Bitcoin* [138], *Ethereum 2.0* [64], *Algorand* [80] *Hyperledger Fabric* [96] and *Ethereum private* [78]. We classify these platforms according to security and dependability properties, outlining strengths and weaknesses. Then, we deepen the analysis facing *consensus* protocols, namely *Proof-of-Work* (PoW) [138], *Proof-of-Stake* (PoS) [65], *Pure Proof-of-Stake* (PPoS) [80], *Practical Byzantine Fault Tolerance* (PBFT) [37], and *Proof-of-Authority* (PoA) [51]. Hence we propose a taxonomic study of their security and dependability properties. Finally, we provide a systematic review of the most critical vulnerabilities and attacks of blockchains *smart contract* applications. We evaluate through a taxonomic classification how these threats impact security and dependability of blockchains, providing a fair classification with respect to the properties they violate. In this analysis we focus on vulnerabilities and attacks emerged in Ethereum's smart contracts. Although there exist other blockchain platforms with smart contracts capabilities (*e.g.*, Algorand), these systems are extremely new and they lack of large and consolidated ecosystem of use cases. Conversely, Ethereum was the first pioneering smart contract platform, and nowadays it hosts thousands of dApps, ranging from *decentralised finance* (DeFi), to *gaming*, and *non-fungible-tokens* use cases. Consequently, such a massive adoption increased the threat surface and thus hacks. In the recent years the number of dApps attacks increased exponentially, causing serious damage in terms of economic loss. For instance in 2021 an equivalent of US$ 1.3B got stolen in DeFi application [38]. For this reason, understanding the smart contacts security became a paramount activity for developers and researchers in the blockchain space. Despite some works have been recently proposed to reviews and systematise Ethereum's vulnerabilities and attacks [11, 39, 130, 168], none of them evaluate how these threats may impact the reliability of a blockchain systems violating security and dependability properties.

Contributions. The contributions of this chapter are following summarised:

- we systematise the assessment of security and dependability properties in the context of blockchains, we split the analysis on three blockchain fundamental components, such as the *platforms*, *consensus* protocols, and *smart contracts*;

- we provide a taxonomy of security and dependability properties relevant to blockchains; we organise the properties in two categories, *i.e.*, (i) security properties for consensus protocols, and (ii) security properties for platforms and smart contracts;
- we propose a security evaluation of *platforms*, *consensus* protocols, and *smart contracts*, we evaluate how these components meet the security properties outlining their strengths and weaknesses.

Chapter Structure. Section 3.1 and Section 3.2 introduce the blockchain platforms and consensus protocols considered in this chapter, while Section 3.3 presents a taxonomy of attacks and vulnerabilities for smart contracts. Then Section 3.4 defines a taxonomy of security and dependability properties which in Section 3.5 are evaluated with platforms, consensus, and smart contracts taxonomies. Finally, Section 3.6 discusses the results.

3.1 Blockchain Platforms

In this section we describe the blockchain platforms considered in our analysis. We select three platforms belonging to permissionless blockchains typology, namely Bitcoin implementing Proof-of-Work (PoW), Ethereum implementing Proof-of-Stake (PoS), and Algorand implementing Pure Proof-of-Stake (PPoS). As for permissioned blockchains, we pick Ethereum implementing Proof-of-Authority (PoA) and Hyperledger Fabric implementing PBFT.

Bitcoin

Bitcoin [138] was the first popularised application of permissionless blockchain for a virtual currency. It is an electronic payment system based on cryptographic proof, which avoids the need of a centralised authority, *e.g.*, bank. Normally, payment systems rely on trusted third-parties to process transactions, and such mediation is often subject to transaction costs. This limits the minimum practical transaction size and reduces the viability of small casual transactions.

The Bitcoin protocol focusses on the concept of *machine-to-machine* payment. The system relies on a flood propagated peer-to-peer network that processes transactions in a fully decentralised system. The order of transactions is guaranteed by an underlying *lottery-based* consensus mechanism. To maintain a consistent, immutable, and therefore trustworthy, ledger of transactions, nodes share the Bitcoin *public ledger*, *i.e.*, a blockchain with the list of all transactions ever made. Transactions are managed by asymmetrical cryptography as a chain of digital signatures. Each owner transfers value by digitally signing a hash of the previously received transactions with the public key of the new owner. In this way, only users who have previously received coins, can instigate

new transactions. In Bitcoin, every node has a balance of the cryptocurrency it owns, identified by the history of its previous transactions. When a node receives a transaction, it verifies the sender's balance of crypto-currency by checking their previously UTXOs. Hence, only transfers backed by existing funds can be accepted. Also, the Bitcoin protocol prevents '*double-spending*' of crypto-currency, by only validating transactions not previously used. Only unused transactions are accepted.

Ethereum

Ethereum [45, 194] is the second main open source blockchain project, following Bitcoin. It builds on the idea of *smart contracts*, immutable programs deployed and executed autonomously on the so called *Ethereum Virtual Machine* (EVM). It is equipped with a Turing-complete programming language allowing anyone to write distributed applications. This widens the application domain of blockchain, making it a *programmable* computational infrastructure with no centralised control. The EVM can interpret smart contracts written in Solidity [63], which is a Javascript-like, object-oriented programming language, or Vyper [190] a pythonic programming language. Like in Bitcoin, the shared state is managed by 'enrolled' accounts, which feature a balance of the Ethereum's cryptocurrency, *i.e.*, the *ether* or ETH. ETH is the main internal crypto-fuel of Ethereum, and is used to pay transactions fees. Like in Bitcoin, nodes run PoW consensus to reach agreement on the order of transactions, and to maintain the consistency of the shared state.

There are two types of accounts: *externally owned accounts* (EOA), controlled through the use of asymmetric cryptography, and *contract accounts*, controlled solely by their smart contract code. The former are similar to standard Bitcoin accounts, whereas the latter can perform read and write operations on the blockchain through the action of smart contracts. Ethereum transactions are activated by EOAs. They have two critical additional components over Bitcoin:

1. *gas price*: a scalar value representing the number of ether to be paid for each computational step of the transaction's execution;
2. *gas limit*: a scalar value representing the total amount of gas that can be consumed by the transactions in a block. It determines the amount of transactions handled in a block, *aka* block size.

Each instruction executed 'consumes' a given amount of *gas*. Each transaction comes equipped with an overall limit to the gas it can consume while executing. This prevents runaway transactions, which accidentally or maliciously engage in never-ending computations. We leave to the reader to consider the crucial importance of this restriction

for the mining process. Complementarily, gas price sets a fee system for computation, proportionally to its overall resource consumption. This includes not just bare computation, but also bandwidth and storage. Ethereum also introduces the concept of *message* to describe a transaction. Messages are produced by contracts and essentially represent a call to a smart contract. These are virtual objects that only exist in the Ethereum execution environment, and are sent by contract accounts. A message contains:

- the sender of the message;
- the recipient of the message;
- the amount of ether to transfer alongside the message;
- the gas price.

Ethereum 2.0

Ethereum 2.0 is the most important update of the Ethereum protocol. It aims at addressing the problems and limitations that plagued Ethereum in the past years, such as scalability and performance. Ethereum 2.0 proposes a set of improvements to scale up the network while preserving security and enhancing usability. In particular, the update to Ethereum 2.0 brings two major improvements in the protocol, such as the shift *from PoW to PoS* consensus algorithm, and the introduction of *Shard Chains* for scalability.

- *From PoW to PoS*: The first implementation of Ethereum ran the PoW, relying on *mining* to build blocks on the blockchain. However, PoW is well known to be an inefficient and energy consuming protocol. With the upgrade to PoS, Ethereum 2.0 will evolve to a more secure, scalable, and energy efficiency platform. Indeed, instead of relying on physical miners and electricity, PoS relies on validator nodes that reach consensus by executing a leader election among stakeholders.
- *Shard Chains*: it is a new scalability feature that drastically improves the throughput of the Ethereum blockchain. Shard Chains change the way the blockchain is replicated across the nodes of the network. The traditional Ethereum implementation is characterised by a single blockchain replicated over thousands of nodes. This feature made up on an incredibly secure ledger of blocks, hard to tamper with. However, this architectural choice requires an extreme waste of nodes, indeed the nodes are required to process and validate each transaction sequentially. This represented a bottleneck in Ethereum computation capabilities and efficiency. Shard Chains introduce a mechanism through which the blockchain is divided into smaller pieces, *i.e.*, *shards*. Hence, shards are distributed among many nodes which are only responsible to handle with data of its shards. This allows for transactions to be processed in parallel rather than consecutively, speeding up the transactions processing, and obtaining better throughputs.

Ethereum Private Networks

Currently, there are many implementations of the Ethereum protocol, most of them offer the possibility to be used in private settings, and we call it *Ethereum-private*. Two of the most interesting Ethereum clients are *Geth* [82], the Ethereum implementation in GOLANG language, and *Parity* [151], a RUST-based implementation. Both of them offer, through special configurations, the possibility of building permissioned private blockchain environments in which transactions are visible only to a subset of network participants. These Ethereum clients for private networks enable the integration of pluggable lightweight consensus algorithms. These type of chains are mainly used in development *testnets*, where distributed applications are tested and debugged before deployment on the main Ethereum blockchain. Nevertheless, private networks are getting increasingly popular in the industry sector for business and enterprise use cases which require lightweight implementation, higher performance and higher privacy guarantees.

Algorand

Algorand [2] is a novel permissionless blockchain platform that aims at solving the scalability, decentralisation, and security issues that plagued most of the blockchain systems to date. Algorand was founded by the MIT professor and Turing award winner, Silvio Micali. Micali's research interest mainly focus on cryptography and distributed computing, and he is co-inventor of *zero-knowledge proofs* [23], and *Verifiable Random Functions* (VRF)[132].

Like Ethereum, Algorand is built on the idea of smart contracts running on the blockchain. It embeds a distributed computation engine, *i.e.*, *Algorand Virtual Machine* (AVM), that runs on every node of the network and executes smart contracts. Algorand's smart contracts are self-verifiable pieces of code that run on the blockchain and automatically approve or reject transactions according to a certain logic. Algorand's smart contracts are classified in two main categories: *stateful* and *stateless* smart contracts. The former handles the primary logic of applications and have functionalities to read and write data on the local state of a node or in a global state of the blockchain. The latter, are usually used in conjunction with their stateful counterpart, and represent static programs used to sign transactions according to a specific logic. The AVM interprets smart contracts written in an assembler-like language called TRANSACTION EXECUTION APPROVAL LANGUAGE (TEAL).

Algorand's core innovation is its new consensus protocol, PPoS, which can reach a BFT agreement in large networks without giving up scalability or security. Besides the PPoS details that we discuss in Section 3.2, Algorand's core innovation is to bring cryptographic randomness with VRF, in a PoS system in which no mining is required. The core advantages of randomness are:

- *cryptographic sortition:* we can define it as a *self* leader election. Users run VRF and select themselves as leaders in a completely autonomous and secret fashion; similarly a committee of users is elected to validate and confirm blocks;
- *consensus security:* Algorand's protocols ensures BFT as soon as a majority of the stake in the network remains honest; thanks to randomness, it is impossible for an attacker to target users who join the consensus as they are self elected; when the attacker realises the selected leader, they have already fulfilled their responsibility in the consensus protocol;
- *scalability:* consensus proceeds running VRF that randomly select one leader and a committee of fixed size; there is not computation required for mining; the number of selected users doesn't change as the total number of network users increases;
- *finality:* Differently from PoW based blockchain, Algorand does not fork and once a new block is appended to the blockchain it cannot be removed; transactions are considered final as soon as the Algorand protocol executes them.

Hyperledger Fabric

Hyperledger Fabric [96] is a permissioned blockchain framework featuring a modular architecture in which each software component can be plugged, including the consensus algorithm. The distinguishing characteristic of Hyperledger Fabric is to offer an *authentication* and *authorisation* layer. Indeed, the system is operated by known entities, which can either be members of a consortium spanning multiple organisations or simply come from a single organisation. Each network participant is identified. They are enabled to issue transactions or to be involved in the consensus process only under proper permissions. Hyperledger Fabric provides identification by leveraging on a Certificate Authority that issues a digital identity encapsulated in a X.509 digital certificate to each network participant. As the operating environment is more trusted, it allows to employ consensus schemas lighter than PoW, as for instance PBFT. This of course results in better performances.

Like Ethereum, Hyperledger is built on the idea of smart contracts running on the blockchain – *aka chain-codes* – able to update the ledger state appending transactions that take place among network participants. Assets can range from the tangible (*e.g.*, real estate and products) to the intangible (*e.g.*, contracts and intellectual property), and are represented in Hyperledger Fabric as a collection of *key-value* pairs. In such a permissioned context, the risk of a participant intentionally introducing malicious code through a chain-code is diminished. All actions, whether submitting transactions, modifying the configuration of the network or deploying a smart contract, are first endorsed and then recorded on the blockchain only if deemed valid. Therefore, the malicious party can be easily identified and the incident handled in accordance with the terms of governance

model. Differently from other blockchain platforms, Hyperledger Fabric introduces the concept of *channel*. A channel represents a private blockchain *overlay*. In this context the overlaid blockchain can typically restrict access to authorised members of a consortium only. Basically, organisations on a Fabric network can establish a channel among the subset of participants who are granted visibility to a particular set of transactions. Each channel comprises a channel-specific ledger distributed across its nodes to store transactions occurred on it, immutably. Communications and transactions exchanged within a channel remain private and shared across only channel participants, enabling data isolation and confidentiality. The network of Hyperledger Fabric is composed by various actors with different roles:

- *organisation administrators*: in charge of deploying smart contracts on selected peers and releasing permissions to client applications;
- *client applications*: invoking smart contracts to perform read or write operations to a channel ledger, if holding proper permissions;
- *peers*: maintaining a ledger for each channel they are registered in; in addition, and if so designated, running smart contracts in order to query or update the channel-specific ledger;
- *orderers*: responsible for consensus, i.e., ordering the transactions which occur in all channels, packaging them in blocks and then distributing such blocks to nodes of appropriate channels.

Among these actors the orderers play a vital role, because they validate transactions and implement consensus, so realising the immutable storage of blocks on a channel ledger. Differently from Bitcoin and Ethereum 2.0, which rely on probabilistic consensus algorithms and may cause forks in the ledger, consensus in Fabric is carried out in through voting, and therefore avoids forks at the outset. In other words, any block generated by the ordering service is guaranteed to be final and correct. Thus any network participant has the same view of the accepted order of transactions.

3.2 Blockchain Consensus Protocols

Proof-of-Work

The Bitcoin blockchain system is regulated by a new consensus protocol which ensures transaction order and prevents double-spending in a trust-less network: the *Proof-of-Work* (PoW) algorithm. This protocol consists of a computationally-intensive hashing task executed by the nodes. With PoW, the miners are in charge to do some computations to find a random number (also called *guess*) such that, if concatenated with

the content of a block, 'hashes' to a number lower than a specifically-set target (see Fig. 3.1). Such target number is fixed according to the so-called blockchain *difficulty*, which regulates the average time spent by miners to solve the puzzle.

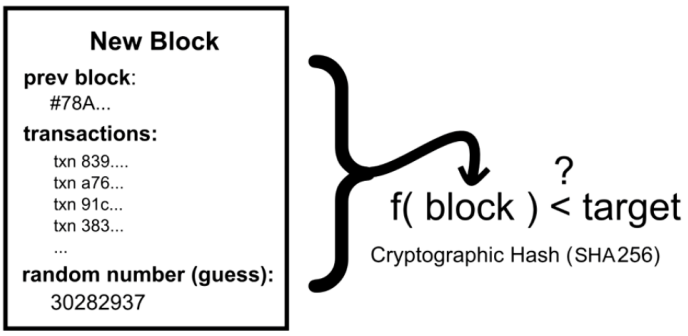


FIGURE 3.1: *Proof-of-Work as a computational puzzle to solve a block*

Once a miner solves a block, i.e., succeeds in the finding the 'guess,' that block is broadcast through the network for other nodes to accept. Once the block is accepted, all the correct nodes consider it as the latest in the blockchain, and start mining new blocks on top of it. For simplicity, we can say that once a miner creates a new block this becomes part of the chain. However, if multiple miners concurrently create and propose new blocks, a transient fork is created. It this is the case, a fork is resolved over time because, by design, miners concatenate blocks on top of the longest chain. With the PoW, miners are incentivised to support the network honestly by a rewards mechanism. Namely, for every mined block, a miner receives a reward which is proportional to the fees each transaction include in its execution [20, 138].

When a node appends a new block to its local chain, it verifies its content and eventually accept it. Hence, once a node has validated and appended a block on the blockchain, the block becomes practically *non-repudiable* and *persistent*, unless an attacker can co-opt the majority of the miners' hash power. In this case, the adversary is able to create a chain fork or reverse a transaction. Assuming a majority of hash power controlled by honest miners, the probability of fork with depth n is $\mathcal{O}(2^{-n})$ [25]. Since the probability of forks decreases exponentially with their length, a suitable strategy for users to be reached a high degree of confidence that their transactions are permanently included in the blockchain (*i.e.*, beating the double spending attack), is to wait for a small number of blocks to be appended to the one that contains them (actually, 6 blocks in Bitcoin). However, [69] shows that "*majority is not enough: Bitcoin mining is vulnerable*" if only 25% of the computing power is controlled by an adversary.

Although they provide strong integrity properties, PoW-based blockchains have a main drawback: *performance*. Their lack of performance is mainly due to the broadcasting latency of blocks on the network and to the time-intensive task of PoW. Indeed, thousands of miners spread all over the network are required to concur if PoW is to render tampering with transactions computational infeasible. To broadcast blocks over network

of this size and topology can take very long. Thus, as a matter of fact, each transaction stored on a blockchain has a high confirmation time, which causes an extremely *low transaction throughput*. In Bitcoin, the average latency is 10 minutes, and the throughput is about 7 transactions per second, while for Ethereum the latency is on average 10 seconds with 24 transactions per second as throughput [25]. Moreover, PoW's mining requires high energy consumptions, hence a huge waste of money and resources.

Proof-of-Stake

Ethash is the PoW implementation of Ethereum. Like in Bitcoin, it suffers from the main drawbacks of probabilistic algorithms. Because of that, the Ethereum consortium proposed a consensus algorithm alternative for public blockchain to the classical PoW, namely PoS, which was first exemplified with the cryptocurrency Nxt [141]. PoS employs a deterministic (pseudo-random) protocol to select the next proposer. The PoS algorithm works as follows. The blockchain keeps track of a set of validators. Any node holding some Ethereum's cryptocurrency, the *ether*, can become a validator by sending a special type of transaction that locks up their ether into a deposit. The validators propose and vote on the next block, and the weight of each validator's vote depends on the size of its deposit (*i.e.*, their stake). In the Ethereum's PoS implementation, called *Casper* [65], each validator's turn is determined by one of the following techniques:

1. *Chain-based PoS*: the algorithm pseudo-randomly selects a validator during each time slot (e.g., every period of 10 seconds might be a time slot) to propose a block. The block is then appended to the blockchain;

2. *BFT-style PoS*: validators are randomly assigned the right to propose blocks. However, the agreement on which blocks to add is done through a multi-round process, where each validator sends a 'vote' for a specific block. During each round and at the end of the process, all (honest and online) validators permanently agree on whether or not any given block is part of the chain. Note that blocks may still be chained together; the key here is that the consensus on a block is reached immediately, and its finality is not influenced by the length of the chain after it.

Differently from PoW, the PoS algorithms causes no waste of energy, because the algorithm demands no high computational effort. Importantly, a validator risks losing their deposit if the block they staked it on is rejected by the majority of validators. Conversely, validators earn a small reward, proportional to their deposit stake, for every proposed block that is accepted by the majority. Thus, PoS induces validators to act honestly and follow the consensus rules by a system of reward and punishment. As expressed by Vitalik Buterin, the creator of Ethereum, in a blogpost: *"in proof-of-stake,*

security comes not from burning energy, but rather security comes from putting up economic value-at-loss" [27]. Despite the PoS security strengths, many blockchain based on PoS algorithms are being suggested which support a reviewed reward policy without penalties, as e.g. [154]. This leads to the *nothing-at-stake* problem, whereby validators are incentivised to sign blocks on any possible chain, as it does not cost them anything to do so. Worse still, in this scenario a validator is actually encouraged to act badly and create as many blocks as possible, opening the way to multiple forks and, as a consequence, to double spend attacks.

Pure Proof-of-Stake

The PPoS [80] it the underlying consensus of the Algorand permissionless blockchain. It achieves consensus applying the concept of VRF. PPoS replaces with randomness the high volume message exchanges employed in traditional voting-based consensus algorithms and the mining process of lottery-based protocols, to elect leaders and confirm blocks. Specifically, the VRF similar to a weighted lottery in which instead of mining, the probability depends on cryptography and the stake. Indeed, the more stake a user owns, the better chance the user has to be elected as leader and propose a block. The PPoS works as follows: it proceeds in rounds, and for each round there are three phases: *block proposal*, *soft vote*, and *certify vote*. When the round starts, users use the VRF to select themselves as leader and committee members. Thus, the leader propose a block to the blockchain, and the committee vote on the block proposal. If a super majority of the votes are from honest participants, the block can be certified. Using randomly selected committees allows the protocol to still be performant while allowing anyone in the network to participate. Assuming a normal execution of the protocol without network partitions and malicious actors, the protocols phases work as follows:

1. *Block Proposal:* the leader selected by the VRF propagates the proposed block along with the VRF output, which proves that the account is a valid proposer;
2. *Soft Vote:* in this phase a selected committee of users vote on the block proposals. Given that for each round, more users might be elected as leader and propose a block, this phase aim is to filter the number of proposals down to one, guaranteeing that only one Block gets certified in that round. Users only select the block proposal with the lowest VRF output. This phase terminates when a quorum of votes from the committee members is reached;
3. *Certify Vote*: a new committee checks the validity of the block proposed at the *soft vote* stage. If valid, the new committee votes again to certify the block. When a quorum of certify votes is reached, the block is committed and the round terminates. If a quorum is not reached by a certain timeout, then the network will

enter recovery mode. In a recovery mode, users agree on a new leader and initiate a new round.

Practical Byzantine Fault Tolerance

In a Byzantine fault tolerance replication, where nodes may be subverted by an adversary acting maliciously, the most prominent consensus protocol is the so-called *PBFT* [37]. PBFT is an extension of the Paxos/VSR (*ViewStamped Replication*) [110, 142] family and it is characterised by a single-leader, view-change protocol. The algorithm proceeds in views, for each view there exists a leader and a set of replicas. Each view executes a *three-phase commit* protocol where replicas exchange messages to reach total order on transactions. In case of leader misbehaviour, all the correct replicas run a view change operation which starts a new view and elects a new leader. In an eventually-synchronous network, where messages are delayed and network partitions may happen but are eventually resolved, if an adversary controls f of the N network nodes, the PBFT consensus protocol guarantees strong consistency provided that $f < N/3$. It has been proved that in this scenario $N \geq 3f + 1$ nodes is a condition necessary and sufficient to guarantee Byzantine fault tolerance [37].

Proof-of-Authority

PoA is a family of consensus algorithms whose claim to fame is its increased performance over typical BFT algorithms; this indeed arises from lighter demands in terms of message exchanges. PoA was originally proposed as part of the Ethereum consortium for private networks and implemented with the protocols called *AuRa* and *Clique* [148, 156]. PoA algorithms rely on a set of trusted miners called *authorities*. Each authority is identified by a unique *id*. Consensus in PoA algorithms relies on a *mining rotation* schema, a widely used approach to fairly distribute the responsibility of block creation among authorities [21, 75]. Time is divided into *steps*. In each step, an authority is elected as block proposer, *i.e.*, leader. The way an authority is elected as leader differs in the two consensus protocols. AuRa proposes a deterministic function based on UNIX times, which requires strong synchronisation assumptions on the network. Conversely, leaders in Clique are elected according to the height of the blockchain, and the elected leader is strictly related to the block number. These two PoA implementations work quite differently: both have a first round where the new block is proposed by the current leader, the so-called *block proposal*; then AuRa requires a further round, called *block acceptance*, while Clique does not. The PoA is a hybrid consensus protocol between the lottery-based and voting-based approaches, where leaders are elected according to a deterministic function. PoA protocols guarantee eventual consensus on transactions. Indeed, the lightweight leader election may lead to forks that eventually are resolved. Consequently, PoA cannot achieve instant finality but this is delayed in time. According

to the concept of the longest chain, a block in PoA is considered final when a majority of further blocks have been proposed, under the assumption that blocks are proposed at constant rate [148].

3.3 Smart Contracts Security: A Taxonomy

In this section, we systematise twenty-five Ethereum smart contracts vulnerabilities. For each vulnerability, we provide a brief description and we outline the prevention or mitigation methods. Then, we present eight famous attacks that exploited those vulnerabilities. We refer to invoked smart contracts as *callee*, whereas to invoker smart contract as *caller*. We refer to the EOA creating a contract as the *owner* of that contract.

3.3.1 Smart Contracts Vulnerabilities

(V_1) Reentrancy. This vulnerability occurs when a caller contract invokes a function of an external callee contract. Specifically, a malicious actor can call back from the callee contract a funds withdraw function of the caller contract, *i.e.*, *reentrancy*, before the execution of the caller triggering an infinite loop of calls. This allows the attacker to bypass the validity checks of the caller and iterate infinite withdrawal calls until the caller contract is drained of Ether or the transaction runs out of gas. Two reasons causes this vulnerability [163]: (i) validity checks are handled by state variables that the contract do not updates until the execution of other transactions terminates, (ii) no gas limit required when handling interactions between external smart contracts. Prevention methods:

- update the state variables before calling external contracts;
- introduce a `mutex lock` [63] in the contract state so that only the owner can update such state; a `mutex` avoids reentrant calls to an external contract from a caller;
- use the `transfer` [63] method to approve payment transactions to external contracts; this function avoids infinite loops because only provides 2300 *gas* to the callee.

(V_2) Integer overflow and underflow. This vulnerability occurs when a function of a callee smart contracts computes an arithmetic operations that falls outside a specific Solidity datatype [39]. This cause an *out-of-bound* error that enables an attacker to manipulate the balance or the state of the contract. This vulnerability is caused by a lack of integer overflow/underflow detection that let exploit a flaw in the Solidity source code that does not provide validation checks on numeric inputs. Prevention methods:

- use the *SafeMath* [144] library used in SOLIDITY to perfroem arithmetic operations with checks on underflow/overflow errors;
- use *Mythril* [44], a SOLIDITY library that checks the security of EVM bytecode before execution;
- use Vyper instead of Solidity; Vyper offers a better control on overflow/underflow issues.

(V_3) `Delegatecall` injection. The EVM provides the opcode `delegatecall` to embed a piece of bytecode of the callee contract inside the caller [39]. This opcode fosters code-reuse, however it also allows an attacker owning a malicious callee contract to inject payloads into the caller and directly gain access to its state and balance. This vulnerability is caused by the possibility of accessing the caller contract variables through the bytecode of the callee. Prevention method: create a contract that shared via the `delegatecall` as a stateless library.

(V_4) Frozen Ether. This vulnerability causes users' funds deposited in contract account to be locked and impossible to withdraw back, effectively freezing them into the contract. The causes of this vulnerability are twofold: (i) the deposit contract account does not provide any function to spend funds using a function from an external contract as library, (ii) the callee contract `selfdestruct` function is executed without checks. Prevention method [39, 168]: a deposit contract shall assure that the mission-critical functions, or money-spending functions are not outsourced to external contracts.

(V_5) DoS with unexpected revert. This vulnerability occurs when the execution of a transaction is reverted due to a thrown error, or due to a malicious callee contract that deliberately interrupts the execution intentionally. Prevention method: as a best practice, the transaction sender provides to the callee a certain amount of money as a reward for the execution of a transaction, so that the callee does not revert it.

(V_6) DoS with GasLimit. This vulnerability causes transactions to be aborted due to exceeding gas limit. This vulnerability is caused by unbounded operations in a contract [39, 168]. Prevention methods:

- contracts should not execute loops on EOA accessible data structures;
- loops, when necessary, should be controlled; the execution of a loop must always terminate, even if the transaction is aborted by the caller.

(V_7) Forcing Ether to contracts. This vulnerability occurs when a contract control-flow uses the `this.balance` or `adresss(this).balance`. This operators can be leveraged to manipulate the receiver of money-spending functions [39]. Prevention method: do not use contract's balance operators in condition checks.

(V_8) Insufficient signature information. This vulnerability causes a digital signature to be valid for multiple transactions. This happens when a sender uses a *proxy* contract rather than sending multiple transactions [39, 168]. Basically, a proxy contract acts as a deposit which stores funds for one or more authorised receivers. An authorised receiver owns a digitally signed message delivered off-chain from the sender. The receiver withdraws funds from the proxy, which must verify the validity of the digital signature. However, if the signature is malformed (missing nonces, or proxy contract address), a malicious receiver can reuse the signature to reply the withdraw transaction and drain the proxy balance. Prevention method: The contract shall check the contract address and the nonce for each withdraw transaction used with digitally signed messages.

(V_9) Generating randomness. This vulnerability concerns smart contract using pseudorandom number generators (PRNG) to create random numbers for application specific use cases. Specifically, this vulnerability affects methods using random numbers created by a PRNG, in which the base seed of the generator is a parameter controlled by miners, *e.g.*, `block.number`, `block.difficulty`. A malicious miner can manipulate the PRNG in order to generate an output that is advantageous for itself. Prevention methods:

- do not use mining variables for control-flow decision;
- use off-chain approaches to PRNG as oracles, like the OracleRNG OraclRNG or the collaborative commit-reveal scheme proposed by RANDAO [159].

(V_{10}) Block Timestamp manipulation. This vulnerability affects smart contracts that use `block.timestamp` parameter in the control-flow (*e.g.*, timestamp used to schedule periodic payments) or as source of randomness [39]. As miners can control this parameter, a malicious miner could adjust its value so that to change the logic of functions that use the timestamp, to take profit. Prevention method: avoid the parameter `block.timestamp` in any control-flow decision logic.

(V_{11}) Transaction ordering dependence. This vulnerability is caused by a malicious manipulation of the transactions priority mechanism used in Ethereum. Transactions include a gas price which determines the reward a miner receive to execute a transaction. It is usually used to prioritise the execution of certain transactions over others [194]. However, malicious miners can alter this procedure and always prioritise its own transactions regardless the gas price, hence manipulate the global state of the blockchain in its favour [124, 168]. Mitigation method: hide the gas price from transactions using cryptographic committees or through guard conditions [39].

(V_{12}) Under-priced opcodes. This vulnerability is caused by under-priced opcodes that can be executed at low cost and that consume a large amount of resources. A misuse of the opcodes, or a malicious actor, might trigger several invocation of these opcodes wasting the majority of resources. This vulnerability is caused by misconfigured gas price

parameters. Mitigation method: the vulnerability remains unresolved [39, 41], however the Ethereum protocol has been upgraded so that to limit opcodes under-pricing [62].

(V_{13}) Ether lost to orphan address. This vulnerability is caused by a lack of validation checks on Ethereum's payment transactions. Indeed, Ethereum only checks the recipient's address format but not if such an address is valid neither if it exists. If a user sends money to non-existing addresses, Ethereum automatically creates a new *orphan* address [11]. An orphan address is neither an EOA nor a contract address, hence it is impossible for the user to move the money which are indeed lost. Prevention method: at the time of writing, the only fix to orphan addresses it to manually assuring the correctness of recipient's address [39].

(V_{14}) Short address. This vulnerability is caused by the EVM missing validation check on addresses. Recall that inputs are expressed as an ordered set of bytes, in which the first four bytes identify the callee's function, then other inputs are concatenated in chunks of 32 bytes. In case of arguments with less bytes, EVM auto-pads with zeros in order to generate the 32 bytes chunk. An attacker could manipulate this process to execute malicious actions. For instance, as showed by *Chen et al.* [39], if we consider the `transfer(address addr, uint tokens)` function and a bad formatted `addr` with one missing byte, the auto-pad adds extra zeros at the end of addr, and consequently increases the number of tokens in the transfer. Prevention method: write functions that check and validate the length of transaction's inputs.

(V_{15}) Outdated compiler version. It occurs when a smart contract uses outdated/buggy versions for the compiler, making the compiled smart contract vulnerable. Prevention method: use up-to-date EVM compiler [39].

(V_{16}) Type casts. This vulnerability is caused by a vulnerable type system in Solidity that may lead a caller smart contract to invoke the function of an unwanted callee. For instance, we can consider a caller contract C invoking the function $f()$ declared by the callee contract C_2. When the EVM compiles C, Solidity only checks that the callee contract declares function $f()$, but does not checks the address of C_2. Hence, an attacker can create a malicious contract C_x and declare the same function $f()$, which instead runs malicious code. At the time of writing, the current version of Solidity does not have a fix to this vulnerability [11, 39].

(V_{17}) Secrecy failure. This vulnerability was firstly exploited in a multi-player game [53]. By nature, data stored on a public, permissionelss blockchain, such as Ethereum, cannot be secret. Although permissioning techniques can be used to restrict the access on a certain state, the transaction issued to write on these state is publicly accessible and anyone can see the attached value. This limit outlines the lack of secrecy and confidentiality in public blockchains. Prevention method: use cryptography to write encrypted data on the state of a blockchain [11, 39].

(V_{18}) Unchecked call return value. Firstly introduced in [124], this vulnerability can be classified according to variants: *gasless send* and *unchecked send* [11, 39, 99]. It is caused by a divergent error handling used in Solidity according to different smart contract calls. Specifically, in the event of an exception, if the caller directly referenced the callee contract, Solidity propagates the exception to the caller and the transaction is reverted. However, if the caller invoked the transaction through a low-level method, *i.e.*, `send`, `call`, `delegatecall`, and `callcode`, Solidity only returns a FALSE message to the caller. Hence, if this discrepancy is not handled properly, it may lead to the execution of unintended or buggy transactions. Prevention method: the caller contract must check the callee responses and address both scenarios described above.

(V_{19}) Uninitialised storage pointer. This vulnerability is caused by a bad organisation of uninitialised complex variables (like struct, array, or mapping) in memory. Solidity usually stores variables in the memory stack starting from slot 0. When complex variables are initialised, a reference is assigned to a free memory slot in the stack. However, if a complex variable is not initialised properly, Solidity automatically assigns that variable to a default space in memory, which starts from slot 0, overwriting the content of the slot. Starting from Solidity version 0.5.0 this vulnerability has been fixed [39].

(V_{20}) Erroneous constructor name. The constructor of a contract is defined by Solidity with a function declared with the same name of the contract. The constructor function determines the owner of a smart contract when the contract is created. Solidity lacks of standard syntax for constructor functions with respect normal functions. Hence, a misspelled constructor name is equivalent to a generic function, which is public and accessible by any EOA. This vulnerability allows any users to become the owner of the contract and consequently take any decision, e.g. move funds, destruct the contract. This vulnerability has been eliminated, starting from Solidity 0.4.22, introducing the *constructor* keyword [39].

(V_{21}) Call-stack depth limit. The EVM execution mode uses a call-stack capped at 1024 frames for contract's external calls [194]. When a caller contract invokes another contract, the call-stack depth increases by one, until reaching 1024. When the limit is reached, Solidity throws an exception and aborts the transaction; from this point any subsequent transaction from both contracts is aborted. An attacker can take advantage of this vulnerability creating a malicious contract and call it 1023 times, and then use the last call to overflow the stack depth limit of a victim contract. This vulnerability has been fixed with the Ethereum upgrade EIP-150 [62].

(V_{22}) Leaking Ether to arbitrary address. This vulnerability allows any unauthorised caller to withdraw funds from a smart contract. It was caused by a lack of caller's identity validation in the money-spending functions to external addresses. Prevention method: enforce adequate authentication on the money-spending functions [39, 168].

(V_{23}) Erroneous visibility. This vulnerability takes advantage of Ethereum's public nature. Transactions (including data, balances and contract codes) are visible to any user. However, Solidity provides four types of visibility to restrict the access to a contract's functions, namely `public` open to everyone, `external` only externally from the contract, `internal` only internally (the contract and its related contracts) , and `private` only within the contract. By default, Solidity assigns the type `public` to functions, hence in case of erroneous visibility configuration, an attacker might be able to call the function from the external [39]. To avoid this, with Solidity 0.5.0 and above, it is mandatory to specify the function visibility.

(V_{24}) Unprotected suicide. This vulnerability is related to V_{20}. Solidity contracts can be killed or deleted using the `suicide` or *self-destruct* methods. Usually, only the contract's owner, or authorised external users, can invoke these functions. However, there might be cases in which the owner is not verified by the functions, or the owner itself is malicious, in that case an attacker can invoke one of these methods and kill the contract. Prevention method: enhance security checks with, multi-factor authentication mechanisms, to assure that the `suicide` or `self-destruct` calls are approved by different parties.

(V_{25}) `tx.origin` authentication. The `tx.origin` global variable is associated to Ethereum's transactions and indicates the EOA who initiate them. The use of this variable for authentication exposes the contract to phishing attack [39, 168]. Indeed, some functions may use this value to authorise sensitive transactions like payments. However, `tx.origin` refers to the creator of a transaction rather than the contract's owner. An attacker may create a malicious contract to bypass function checks of a victim contract and let the owner to call it with phishing techniques. Prevention method: use the global variable `msg.sender` instead of tx.origin for authentication.

3.3.2 Smart Contracts Attacks

In this section we describe the most relevant attacks detected on Ethereum applications based on smart contracts. The attacks exploited a few of the vulnerabilities introduced above. A comprehensive review of Ethereum attacks can be found in [11, 39].

a) DAO attack. The Decentralised Autonomous Organization (DAO) [181] is a financial application running on Ethereum. The DAO manages a decentralised crowdsourcing system: democratic voting for proposed investments projects. When the majority agree on the proposed investment, the proposed account gains the funds, while the minority gets refunded. It was attacked in 2016 provoking a loss of US$ 60M [39]. The contract had a *reentrancy (V_1)* vulnerability in the function `splitDAO()` used to fund projects and handle the refunds. The exploit allowed the attacker to recursively call the `splitDAO()` function causing a malicious investor to draw more money than it deserved [86].

b) Parity multisignature wallet attack. Parity multisignature wallet, *aka*, Party *Multisig*, is an implementation of the Ethereum multisignature wallet, *i.e.*, a smart contract which requires multiple private keys to unlock a wallet and consent withdrawals. A multisignature wallet safeguards Ether or tokens with an additional layer of security. In 2017, the Parity Multisig wallet was compromised twice. The first hack exploited the `delegatecall` *injection* vulnerability (V_3) and the *erroneous visibility* vulnerability (V_{23}) to drain Ethers worthy of ≈ US$ 31M [11, 39]. The attacker took the ownership of a smart contract used as a library for the Multisig implementation, *i.e.*, the *WalletLibrary*, and drained the wallet of 26,793 ETH [85], which at that time was worth about US$5M. Parity fixed this vulnerability by using the `require()` opcode to avoid the contract re-initialisation that was exploited by the attacker to take control of the contract. However, a second attack was carried out on this fixed contract. This time, the attacker exploited the *unprotected suicide vulnerability* (V_{24}) and the *frozen Ether* vulnerability (V_4), freezing approximately US$ 280M in the affected wallets forever [7]. The attacker managed to take again the ownership of the fixed contract and then destroyed the contract that was shared by thousands of users by calling the function *kill()*. Indeed, although Parity developers patched the *WalletLibrary* to avoid contract re-initialization via `delegatecall`, they left the contract uninitialised. The attacker simply claimed the ownership initialising the contract bypassing the patch.

c) GovernMental attack. The GovernMental contract is a Ponzi scheme game [16]. Players participate by depositing Ether in the contract, until for 12 hours after the last partitipant, no more players join the game. At this point the last participant wins the entire jackpot. The contract suffered of *unchecked call return value* vulnerability (V_{18}) and the *call-stack depth limit* vulnerability (V_{21}), which were exploited to cancel the payment transaction of the jackpot [39]. Broadly, the attacker called 1024 external contracts to reach the limit of the stack, so that the FALSE response message from the callee contract got lost. Hence the jackpot winner never received the money, which remained locked in the GovernMental contract.

d) HYIP attack. HYIP contract is another Ponzi scheme game which rewards investors with funds coming from investors that join the scheme at the end of each day. The HYIP contract suffered of a *DoS with unexpected revert* (V_5) vulnerability which caused the contract to be drained by its funds [39].

e) Fomo3D attack. The Fomo3D is another Ponzi game in which the participants rush in to buy a key before a timeout, when the timer runs out, the last participant that completed a purchase of a key wins the jackpot; the more participants the highest is the key's price; when a new payment is done, the timeout is extended by 30s. In addition, Fomo3D implemented an incentive mechanism to attract participants: all buyers joined a lottery that randomly rewarded one of them with a small prize. The contract had a *generating randomness* (V_9) vulnerability that was exploited by an attacker to cheat with the game [39]. Indeed, Fomo3D computed the lottery winner using a PRNG and as seeds

the current block state (`block.number`, `block.difficulty`, etc.) and the `msg.sender` values. However, since the block state is predictable, the attacker brute-forced the function to win the lottery.

f) BEC Token attack. The ERC-20 Token is a standard smart contract that issues tokens on the Ethereum blockchain. It contains several functions that a compliant token smart contract must implement, such as `totalSupply()`, `balanceOf()`, `transfer()`, and more. In 2018 a famous Ethereum token, the BeautyChain (BEC) was attacked and a substantial amount of tokens got stolen [1]. The attacker exploited the *integer overflow and underflow* (V_2) vulnerability. The BEC contract contained a function called `batchTransfer()` to enable multiple token transfers to different users simultaneously. The function's first argument was a dynamic array of addresses, and the second was the number of tokens to transfer. The attacker succeeded to inject an overflow in so that to transfer a considerable amount of tokens via the buggy function.

g) ERC-20 signature replay attack. This attack exploits the *insufficient signature information* (V_8) vulnerability. It was exploited to steal tokens through proxy services. A use case of proxies in Ethereum arose for the transfer of tokens, *i.e.*, *proxytransfer*. Since Ethereum transfers require the payment of Ether fees, token owners that do not own Ether cannot move their tokens. The proxytransfer method overcomes this by executing token transfer transactions on behalf of authorised users using their digital signatures. However, in this attack the signature was not related to any specific contract address, consequently, a malicious receiver was able to replay the signed message over other contracts and steal additional tokens from the sender [39].

h) Rubixi attack. The Rubixi contract is a famous Ponzi scheme that suffered from *erroneous constructor name* (V_{20}) vulnerability. The contract original name was DynamicPyramid, which was later renamed Rubixi. However, the developers forgot to update the name of the contract constructor function, leaving the `DinamicPyramid()` function. Consequently, a malicious user calling that function was able to become the contract owner and therefore steal the funds of the contract [39].

3.4 Taxonomy of Security Properties for Blockchain

In order to evaluate the security and dependability aspects of blockchain systems, in this section we produce a taxonomy of security properties, inspired by the classical tripartite definition of security as CIA: confidentiality, integrity and availability. Our proposed properties are organised in two categories: *(i)* security properties of the *consensus protocols* employed in blockchain systems; and *(ii)* security properties affecting the *platforms and smart contracts* underpinning blockchain services.

A blockchain can be described summarily as a *secure* distributed protocol which aims to regulate and store the transactions happening in a distributed network of parties. Such a system must satisfy the important properties of *safety* and *liveness*. Besides these, another important semantic metric in the evaluation of security and trustworthiness is the so called *fairness* property. Following seminal work by Francez [74], we distinguish two aspects of the fairness guarantees provided by a blockchain protocol: *validator fairness* and *client fairness*. The former relates to the consensus mechanism, while the latter is dictated by the architectural choices of the platform. Section 2.3 introduced the fundamental properties used to measure safety and liveness of blockchain consensus protocols, that we recall as follows:

Persistency: the consensus algorithm should not do anything wrong during its normal execution; it prevents unwanted executions and ensures properties like *integrity*, *consistency*, *validity* and *agreement* (Section 1.1). When an honest node accepts a transaction, then all the other honest nodes will make the same decision, which results irreversible. If persistency is violated at some point in time, it will never be satisfied again after that time.

Termination: this property is aimed to prevent systems to meet persistency requirements simply by staying idle and doing nothing. Termination is used to ensure that consensus protocols keep making progress towards an end, hence transactions to correctly terminate, *i.e.* be inserted into a finalised block.

Validator fairness: in a consensus algorithm, the leader election mechanism is fair if any honest node can be potentially selected to the set of nodes that will participate in the agreement to select the next block. In the context of blockchain, this property usually measures the degree of decentralisation.

Turning to the security evaluation of blockchain platforms and smart contracts, we define six properties based on the definitions of security and dependability introduced in Section 1.2.2, such as the *CIA triad* and user *profiling*. The whole blockchain indeed acts as trusted system as long as the network is populated by a majority of honest nodes. A trustworthy system must be dependable and secure, ensuring properties introduced with the CIA triad such as confidentiality, integrity and availability [12]. In addition we identified the relevance of the properties of *accountability* and *authorisation*. Such properties lead to fairness constraints which can be used to detect misbehaving participants. Hence, our suggested taxonomy of security and dependability properties to evaluate blockchain platforms and smart contracts is as follows:

Confidentiality: possibility for blockchain nodes to keep some of their transactions confidential; absence of unauthorised leaking of sensitive information owned by one or more nodes;

Integrity: absence of improper alterations of the blockchain data from unauthorised users;

Availability: ability of the system to run correct services without interruptions;

Authorisation: ability of the system to specify access rights and privileges to resources and to define permission roles to participants;

Accountability: ability of the system to trace back the operations and the behaviour of a certain user identity/physical entity;

Client fairness: willingness of the system to democratically accept transactions from any client without any preference.

3.5 Security Evaluation

In this section, we propose a taxonomy to evaluate the blockchain systems discussed in this chapter over three dimensions, *i.e.*, *platforms*, *consensus*, and *smart contracts*. Each dimension is characterised by security flaws and inherits security features from the layers underneath. Smart contracts and consensus are usually built on top of the platforms. Verifying and establishing trust in the security semantics of a blockchain protocol is challenging. Broadly, a secure system should be resilient in the presence of adversarial conditions, and tolerate threats to its normal execution. We propose an analysis of the security properties introduced in Section 3.4 with respect each blockchain dimension. Specifically, we first evaluate how the consensus protocols meet the security properties, then we extend the analysis to platforms. Finally, in TABLE 3.3 we evaluate how smart contract's vulnerabilities and attacks threaten security.

Evaluation of security properties

In this security analysis we assume a deployment of n nodes responsible for consensus, which communicate over an eventually synchronous network. The eventually synchrony model [60] simulates a real world network in which messages are arbitrarily delayed, but eventually delivered within a fixed and known time bound. We compare the security properties of the protocols *PoW, PoS, PPoS, PoA* and *PBFT* and of the platforms *Bitcoin, Ethereum 2.0, Ethereum-private, Algorand* and *Hyperledger Fabric*.

TABLE 3.1 summarises the consensus resilience of the four targeted algorithms. The table shows whether the security properties of persistency, termination and validator fairness are met by the consensus protocols. Firstly, we observe that PoW, and PoS enjoy strong termination. This property follows because the probabilistic leader election based on mining, and staking. Transactions are guaranteed to be added on the blockchain

	PoW	**PoS**	**PPoS**	**PBFT**	**PoA**
persistency	eventual	eventual	yes	yes	eventual
termination	yes	yes	eventual	$n \geq 2f + 1$	eventual
validator fairness	hw dependent	stake dependent	stake dependent	yes	yes

TABLE 3.1: *Security evaluation of blockchain consensus protocols*

as soon as the mining proceeds, or there is a majority of stakeholders. Conversely, probability in leader election affects persistency of PoW and PoS, because the possibility of having forks. In this case persistency is not guaranteed due to inconsistent views of the blockchain. However, such forks will eventually be resolved, according to the specifics of the platform. For example, in Bitcoin a block is considered final when other six blocks are proposed after it, as the probability of a fork longer than that is vanishingly small [25]. For this reason, the persistency of such protocols must be classified as *eventual*. Conversely, the PPoS protocols does not allows forks, and blocks are instantly finalised, prioritising persistency over termination [80]. Indeed, to ensure persistency, the PPoS allows stalls from malicious behaviours or network issues. However, PPoS' recovery mechanism ensures the protocols continuation after a stalling problem, hence its termination is classified as *eventual*. Both PoS protocols ensure security until a majority of the stake remains in honest hands. If this condition is not verified, such systems can be easily compromised. Validators with the majority of the stake can determine the next blocks straightforwardly *i.e.*, richest users will be advantaged in the proposal of new blocks. This behaviour is reflected with the validator fairness property in TABLE 3.1. Differently in PoW, such property is strongly related to the hardware capabilities of a miner. PoW's miners with a lot of computational power will have more chances to solve new blocks.

Moving to permissioned blockchains, these system rely on a higher level of trust than the permissionless ones, due to the presence of node authentication. The consensus protocols used in this context are either classical voting-based ones, such as PBFT, or hybrid, for PoA. PBFT has been broadly demonstrated to guarantee persistency in the eventually synchronous model, as long as there are $n \geq 3f + 1$ active nodes in the network, where f represents the number of potentially faulty nodes. Whereas, as soon as $n \geq 2f + 1$ remain honest, termination can be also guaranteed [51]. On the other hand, in PoA, persistency is only *eventually* guaranteed, because of the way PoA reaches consensus finality. Termination is instead dependent on the number of honest validators, hence classified as eventual. As long as a majority of validators is active, termination is guaranteed, otherwise the protocol stalls, and transactions are not finalised. In both protocols, validator fairness is guaranteed, since every honest node has the same chance of being elected as leader as the others.

	Bitcoin	Ethereum 2.0	Algorand	Hyperledger	Ethereum-private
confidentiality	no	no	co-chains	channels	private txs
integrity	majority hashpower	majority stake	majority stake	up to $3f+1$	eventual
availability	yes	yes	yes	up to $2f+1$	eventual
accountability	partial	partial	partial	yes	yes
authorisation	no	no	no	yes	yes
client fairness	no	no	yes	yes	yes

TABLE 3.2: *Security evaluation of blockchain platforms*

Our analysis of blockchain security continues in TABLE 3.2, in which we turn the attention to *platforms*. This table illustrates the security aspects of Bitcoin, Ethereum 2.0, Ethereum-private, Algorand, and Hyperledger Fabric. We observed that the integrity of Bitcoin, Ethereum 2.0, and Algorand is strongly linked to where hash power and stake lie. Indeed, an attacker owing the majority of the hash power (or stake), could break the consensus protocol as we already mentioned above, hence maliciously inject s hard fork with a subverted chain [25]. In contrast, in Hyperledger and Ethereum-private, the integrity property is strongly tied to the persistency property of their underlying consensus algorithms. Fabric employs PBFT, which ensures persistency under the assumption of $n \geq 3f + 1$, while Ethereum-private adopts PoA algorithms, which can only guarantee eventual persistency. Despite strong availability, the full replication of the blockchain in the Bitcoin, Ethereum 2.0, and Algorand platforms leads to a lack of confidentiality. Although permissionless blockchain's are usually associated with cryptocurrencies and anonymous payment transactions, users are susceptible to privacy attacks caused by the public nature of the information stored on these blockchains [90]. Indeed, each node in the network has access to the entire ledger of transactions, and it is possible with forensics activities tracing the behaviour of specific users [101]. However, if for Bitcoin and Ethereum 2.0 there is no way to mitigate this issue, Algorand recently designed a solution which combine the public, permissionless network with several private network interconnected, *aka, Co-Chains* [133]. Contrarily, confidentiality in both Hyperledger and Ethereum-private can be guaranteed through the use of channels and private transactions, respectively. Hyperledger Fabric and Ethereum-private can enforce the so-called profiling properties of authorisation and accounting. This is because nodes are authenticated. Authorisation is guaranteed by managing the permission of each node. Accountability is achieved by tracing the interaction of nodes with the blockchain [92]. This is not so in public permissionless blockchains like Bitcoin, Ethereum 2.0, and Algorand where identities are pseudo-anonymous [90] and users are not authenticated. However, although actions are difficult to attributable to specific entities, it is possible to conduct analysis on the behaviour of specific accounts. Therefore, the public, permissionless nature of these blockchains ensures that anyone can access the history of transactions and trace behavioural patterns. We deduce that permissionless blockchain

offer partial accountability [101, 137]. On the other hand, being these systems public and decentralised, authorisation is not provided.

Lastly, we evaluate the property of client fairness. Permissioned blockchain benefits from fairness guarantees in that each client's transactions are processed without any preference or priority. On the contrary, the execution of transactions in Bitcoin and Ethereum 2.0 is costly (either hardware intense or staking), hence making incentive mechanisms for miners and validators necessary. Bitcoin miners receive fees for processing transactions, while Ethereum 2.0 transactions are embedded with a reward (*gas*). This means that low-rewards transaction may be stalled forever waiting to be processed [192]. Incentives mechanisms for permissionless blockchain, like Bitcoin and Ethereum 2.0, lead therefore to a lack of client fairness. Differently, in the Algorand blockchain every transaction counts the same, and there is not a mechanism to incentivise users to behave honestly. Everything in Algorand is handled by PPoS' cryptography and the computation of VRFs. This allows Algorand to have very low transactions fees, which are thus distributed to rewards accounts for the users, and to ensure client fairness.

We conclude our analysis with the smart contracts. Table 3.3 shows a classification of the *CIA triad* and *profiling* security properties against the vulnerabilities and attacks of our taxonomy. We outline how an exploit of the vulnerability affected, or may affect, those properties. From the table emerges that most of the smart contract hacks cause a violation of confidentiality, integrity, and authorisation properties.

3.6 Discussion

In this chapter, we propose a taxonomical study of security and dependability properties of modern blockchain systems. Specifically, we define the foundational properties that these systems must guarantee, thus we study whether such properties are satisfied. We firstly defined a taxonomy of the four most prominent blockchain platforms and their consensus protocols, namely Bitcoin with PoW, Ethereum 2.0 with PoS, Ethereum private networks (Ethereum-private) with PoA, Algorand with PPoS, and Hyperledger Fabric with PBFT. We organised the analysis across three dimensions, namely platforms, consensus, and smart contracts. Thus, in Section 3.4, we defined security and dependability properties to matter in blockchain systems, and we used such properties to assess the consensus protocols, platforms, and smart contracts. Specifically, we evaluated the consensus resilience of each protocol by measuring its persistency and termination properties. Then, for the evaluation of the blockchain platforms, we instead considered the well known CIA triad, and we evaluated whether the platforms satisfy its properties, in addition to the so defined *profiling* properties of accountability and authorisation. In this analysis, we also introduced an important property, called *fairness*, for both the

Vulnerability	Attack	Security Issues
V_1	DAO [181]	CI + authorisation
V_2	BEC Token [1]	CI + authorisation
V_3	Parity Wallet [85]	CI + authorisation
V_4	Parity Wallet [7]	A + authorisation
V_5	HYIP, DoS [39]	A
V_6	DoS [39]	A
V_7	unexploited [39, 125]	CI + authorisation
V_8	ERC-20 replay [39]	CI + authorisation
V_9	Fomo3D [39]	I + authorisation
V_{10}	GovernMental [168]	IA
V_{11}	ERC-20, Bancor [125]	IA + accountability
V_{12}	DoS [39]	A
V_{13}	unexploited [11]	I
V_{14}	ERC-20 short [39, 182]	I + authorisation
V_{15}	unexploited	all
V_{16}	unexploited [11]	all
V_{17}	Odd-Even Game [39]	C + authorisation
V_{18}	GovernMental [39, 168]	CI + authorisation
V_{19}	OpenAddressLottery, CryptoRoulette, [125]	I
V_{20}	Rubixi [39]	CIA + authorisation
V_{21}	GovernMental [39]	A
V_{22}	unexploited	CI + authorisation
V_{23}	Parity Wallet [39]	CI + authorisation
V_{24}	Parity Wallet [7, 39]	all
V_{25}	unexploited	CI + authorisation

TABLE 3.3: *Taxonomy of vulnerabilities, attacks, and security issues of Ethereum smart contracts. The CIA triad, accountability, and authorisation properties are classified according the vulnerabilities and attacks*

consensus algorithms and the platforms. Finally, we reviewed the most common vulnerabilities and attacks to the Ethereum smart contracts. The choice of Ethereum was driven by its massive adoption in the space of decentralised applications. Hence, we evaluate the impact of such vulnerabilities (and attacks) on the CIA triad and profiling properties. It emerges from the results illustrated in Section 3.5 that permissioned blockchain like Hyperledger Fabric and Ethereum-private can guarantee good fairness and confidentiality, and also provide properties such as accountability and authorisation. However, these systems may require strong assumptions on the underlying network and the number of possibly subverted nodes to guarantee integrity and availability. This is also reflected in the consensus protocol they adopt, specifically PBFT ensures persistency at the cost of eventual termination, whereas PoA can ensure both properties only eventually. Conversely, permissionless platforms such as Bitcoin, Ethereum 2.0, and Algorand offer better integrity and availability, despite failing on profiling, confidentiality

and fairness properties. Finally, we observed how bugs and malformed smart contracts may be susceptible to attacks that can violate confidentiality, integrity, and availability of specific application use cases and therefore the profiling properties. We also observed how severe the consequences of such an attack can be in terms of economic loss.

Chapter 4

METHUS: A Framework and Methodology for Studying Blockchain Consensus Protocols

The underlying consensus protocols of a blockchain system plays a fundamental role when it comes to its performance and security guarantees. For instance, the permissionless blockchains of Bitcoin [138] and Ethereum [194] are driven by the PoW which ensures strong data integrity by resolving computation intense cryptographic operations, while permissioned blockchains like Hyperledger Fabric [31] or GoQuorum [43], employ lightweight consensus protocols at the cost of Byzantine fault tolerance for the former [110], or scalability for the latter [135].

Recently, a plethora of new protocols have been proposed by both the scientific community and private companies, promising to overcome those limitations and maximise the performance of blockchains without affecting security. Therefore, public blockchains are successfully switching from PoW to a more resilient and rapid consensus protocol, namely the PoS, while for private blockchains there is not a common agreement on the next generation protocols [196]. In the field of distributed systems, the most prominent consensus protocol is the PBFT [37]. PBFT offers strong integrity guarantees tolerating the presence of malicious participants, but it does not scale well, *i.e.* in large networks the performance of PBFT drastically decreases. Consequently, over the years a renewed interest arose around the family of BFT protocols, and several fascinating alternatives to PBFT for private blockchains have been proposed, trying to improve current limitations. Most of these protocols combined the concepts of both PoW and PBFT resulting in hybrid solutions [108, 149]. Although these alternatives promise to achieve excellent performance while preserving security, there is a lack of formal justifications of such claims leading these protocols to be dismissed. Indeed, evaluating protocol's performance and security is crucial, especially when deployed in realistic, untrustworthy,

network scenarios (like the Internet) where fault tolerance is necessary to preserve data integrity. This introduces the following research question:

> *How can we assess performance and security of consensus protocols built for blockchain systems?*

Notwithstanding some effort has been spent in the comparison of different blockchain consensus protocols [134, 169], most of these works are limited to a qualitative comparison of protocols characteristics without deepening the analysis of their guarantees under common realistic assumptions like an Internet-based blockchain. The assessment of a consensus protocol is a challenging task in which several aspects must be considered like the network model, trust assumptions, procedures and methods used to observe and validate the protocols' behaviour. However, nowadays there is a lack of common definitions and methodologies, indeed each protocol is built using its terms and definitions relying on unrealistic synchronous ad-hoc network models within specific system failure assumptions. To date, there is an urgent need for standardised methodologies to evaluate and compare blockchain consensus algorithms.

In this chapter, we present METHUS a framework that establishes a systematic *METHodology for the assessment of blockchain consensUS protocols*. Specifically, METHUS' methodology defines *qualitative* and *quantitative* approaches to measure performance and security with the common assumption of a blockchain deployed under untrustworthy conditions. METHUS applies traditional notions from distributed systems to formalise the concepts of performance and security in blockchain systems. The qualitative approach is characterised by (i) a CAP Theorem-based security analysis which classifies a consensus protocol according to consistency, availability, and partition tolerance [81], and (ii) a performance analysis based on protocol complexities and message latencies. Conversely, the quantitative approach defines the metrics to measure both performance and security through experimental evaluation. Therefore, this approach is carried on through a testing procedure that defines four testing phases, namely the *deploy* of a blockchain system, the *load* generation, the data *collection*, and the *measure* of performance and security metrics. Finally, we evaluate METHUS by proposing a comparison of two PoA consensus protocols, namely *AuRa* [148] and *Clique* [156], against the revised blockchain version of PBFT, namely the IBFT [135]. Firstly, we describe these three protocols presenting their components and characteristics, then we use METHUS to evaluate their performance and security under adversarial conditions.

Contributions. The contributions of this chapter can be summarised as follows:

- We describe the functioning of AuRa, Clique, and IBFT protocols under the same blockchain model;

- We introduce METHUS, a framework enhancing a systematic methodology for the evaluation of blockchain consensus protocols. It combines qualitative and qualitative approaches to assess security and performance of protocols under common deployment scenarios;
- We propose an in-depth evaluation of METHUS presenting a comparative study of three blockchain consensus protocols, namely AuRa, Clique and IBFT.

Chapter Structure. Section 4.1 discusses the related works. Then Section 4.2 defines the system model and Section 4.3 introduces METHUS and both qualitative and quantitative approaches. Section 4.4 presents the protocols analysed in this chapter, so then Section 4.5 and Section 4.6 present respectively the qualitative and quantitative comparisons. Finally, Section 4.7 sums up the results.

4.1 Related Works

With the advent of blockchain, the consensus problem has gained a renewed interest from the scientific community. In the recent years, several consensus protocols have been designed for different blockchain systems relying on various settings and configurations. The increasingly number of protocols has led to an urgent need for comparative studies and evaluation procedures. Most of the works have tackled this need by assessing fundamental properties like performance and security, via either qualitative or quantitative approaches.

Kannengießer et al., propose in [100] a classification of the most relevant properties of different blockchain systems and their trade-offs. The authors state that the properties of security and performance, dictated by the consensus protocol, exhibit the more trade-offs, resulting in the more relevant properties for blockchain systems. In [196], *Xiao et al.*, present a comprehensive analysis of performance and security of the most prominent blockchain consensus protocols. The authors introduce an evaluation framework based on five components of a blockchain system that impact the consensus. Hence, they use such a framework to compare the protocols with respect to different performance metrics. A qualitative approach is also used in [134] by *Mingxiao et al.*, to extensively compare blockchain protocols like PoW, PoS, DPoS, PBFT and Raft, and their performance and security properties. Similarly, *Sankar et al.*, investigate in [169] the main differences between SCP - Stellar Consensus Protocol - and the consensus algorithms employed in R3 Corda and Hyperledger Fabric.

In [33], *Cachin and Vukolić* present a thorough security analysis of most-known permissioned systems and their underlying consensus algorithms in term of safety and liveness guarantees. In this work the authors reject all those protocols proposed by the blockchain community without any formal description and detailed security model. *Kiffer et*

al., propose in [102] a formal approach to analyse the consistency properties of blockchains by applying a Markov-chain based method. The study analyses a series of attacks on the protocols and evaluates how those attacks affect the protocols. In [152], *Pass et al.*, formally prove that PoW-based consensus protocols can guarantee consistency and liveness even in asynchronous networks, while in [166, 167], the author demonstrates a security flaw in the first open source implementation of the IBFT protocol. These works present a formal definition of the IBFT and demonstrate that in presence of an eventually synchronous network the protocol is not BFT, violating both safety and liveness. Afterwards, the author proposes a new version of the protocol fixing the identified issues[2]. Similarly, in [6], the authors analyse the correctness of Tendermint - a new consensus protocols that combines the PBFT voting mechanism with the PoS validators election. The authors claim that Tendermint satisfies the consensus security properties under the assumptions of $f < n/3$ Byzantine nodes, but state also that the leader election algorithm is not fair. Differently, *Ekparinya et al.* [61], evaluate the security of two PoAs consensus algorithms by means of experimental evaluation. The authors reproduce a double-spending attack and claim that PoAs are vulnerable to such an attack.

In literature some effort has been spent also on quantitative evaluations of performance and security in blockchain. *Zheng et al.* [199], propose a real-time performance monitoring framework for blockchain systems using a log-based approach reducing computation overheads. *Tuan et al.*, propose in [55] a comprehensive study of performance and security of three blockchain platforms, namely Hyperledger Fabric, Ethereum, and Parity, and their underlying consensus protocols. The authors propose a framework for benchmarking permissioned blockchains measuring latencies and throughputs under several workloads and simulating system faults. As a result, the authors show substantial performance gaps between the blockchains and the classic database systems. Despite the promising features of this tool, there are several elements not considered with the evaluation, such as the network type, message delays and the presence of Byzantine nodes, leading to arguable results. *Hao et al.*, present in [88] a quantitative analysis approach that measures throughput and latencies of Hyperledger Fabric and Ethereum under different transactions workloads. The study shows the performance bottlenecks caused by the consensus protocols. More focussed on Hyperledger Fabric, in [178] *Sukhwani et al.*, propose a model based on Stochastic Rewards Nets (SRN) to measure throughput, latency, mean queue lengths, and resources utilisation of the different components involved in the *execute-order-validate* architecture of Fabric. Likewise, *Baliga et al.* [14], study the performance features of the GoQuorum blockchain under different configuration and with two different consensus protocols; Raft and IBFT. As an approach, this work adapts the Caliper benchmarking tool[3], to enable distributed workloads and measure the transactions throughput and latencies in GoQuorum. More centred on security aspects, a work by *Weber et al.* [192], shows that under certain configurations, the

[2]This is nowadays considered the ultimate implementation of IBFT.
[3]A tool for testing specific Hyperledger-related blockchain projects.

Papers	Qualitative Analysis		Quantitative Analysis	
	Security	*Performance*	*Security*	*Performance*
[100, 134, 169] [196]	✓	✓		
[6, 33, 102] [152, 166, 167]	✓			
[61]	✓		✓	
[14, 88, 178] [199]				✓
[55]			✓	✓
[192]	✓		✓	
METHUS	✓	✓	✓	✓

TABLE 4.1: *Summary of the related works*

Ethereum and Bitcoin blockchains might be affected by availability issues. The authors show experimentally that some transactions never terminate.

Notwithstanding in the recent years a lot of efforts have been devoted to evaluate the trade-offs between performance and security in blockchain consensus protocols, a comprehensive study of these properties applying both qualitative and quantitative analysis is missing. The qualitative analysis proposed so far do not follow standard reasons and methods resulting in confusing approaches, while the quantitative analysis mainly focused either on performance or on security. Moreover, the experiments are usually carried on local testbed or LAN networks that does not reflect realistic Internet-bases deployments. In our work we fill this gap proposing a systematic methodology providing both qualitative and quantitative approaches, and considering a realistic untrustworthy deployment scenario. TABLE 4.1 summarises the related works presented above, and shows how our work arises at the state-of-the-art.

4.2 System Model

We consider a set Π of nodes geographically distributed across a P2P network, running the same blockchain protocol. We assume the nodes to be authenticated through an external authentication manager system.

a) Network Model. The nodes communicate through an eventually synchronous network [60]. Such a network allows the presence of *partitions* in which communications become asynchronous and messages delayed of a certain, unknown, time-bound that holds for an unknown period, called Global Stabilisation Time (GST). In the event of GST, the communications return synchronous, and messages are correctly delivered.

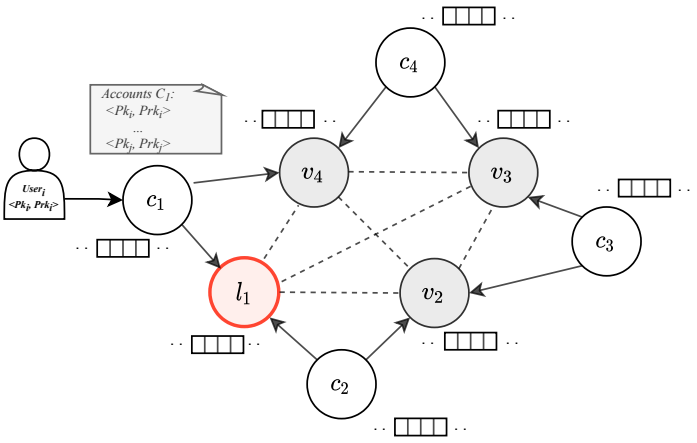


FIGURE 4.1: *Model of a private blockchain networkt*

b) Private Blockchain Model. We define a *private blockchain* as a distributed protocol executed through the set Π. The nodes read and update a replicated data structure, namely the *blockchain*, via read/write operations. The blockchain is a ledger that collects a set $\Omega = \{b_1, .., b_b\}$ of records, called *blocks*, linked together with cryptography. We indicate with i the index of block b_i. Each block carries a number of details such as the cryptographic hash of its previous block, the creation *timestamp*, and the list of *transactions*. The list of transactions is of fixed size and determines the dimension of a block, *i.e.*, *block-size*. We represent with $\Gamma = \{tx_1, .., tx_t\}$ the set of transactions of the blockchain system and we indicate with i the index of transaction tx_i. The blockchain data structure is of type read-only/write-once in which blocks are appended at a certain rate, *i.e.*, *block-period*. The first block of the blockchain is called *genesis block* and determines the initial configuration of a blockchain. The set of nodes Π is divided into three subsets - the set $\mathcal{C} = \{c_1, .., c_c\}$ of *client nodes*, the set $\mathcal{V} = \{v_1, .., v_v\}$ of *validator nodes*, and the set $\mathcal{L} = \{l_1, .., l_l\}$ of *leader nodes* such that $\mathcal{L} \subseteq \mathcal{V}$ and $\Pi = \mathcal{C} \cup \mathcal{V} \cup \mathcal{L}$. We indicate as i the index of nodes c_i, v_i, l_i respectively. FIGURE 4.1 illustrates a private blockchain model. The clients are simple nodes hosting one or more *blockchain accounts* and represent an entry point for external users. A blockchain account is a cryptographic identity represented with the pair $< Pk, Prk >$, respectively public and private cryptographic keys. Each account has a certain *balance* of a generic asset (*e.g.*, cryptocurrencies or digital tokens).

c) Blockchain Protocol. The blockchain protocol can be expressed as a combination of the components in TABLE 4.2, such that $\mathcal{B} :=< \mathcal{V}, \mathcal{L}, \mathcal{C}, \Gamma, \Omega, \alpha >$. Broadly, the protocol proceeds as follows. A user requests the execution of a transaction tx_i via its blockchain account toward the client node c_i. c_i verifies the validity of tx_i via a deterministic function $\gamma : \Gamma \rightarrow \{True, False\}$, such that $\gamma(tx_i) \rightarrow \{True, False\}$[4]. If $\gamma(tx_i) = True$, the transaction is accepted and forwarded to the validators. The validators store

[4]The validity was firstly introduced by *Cachin et al.*, with the notion of *external validity* [33].

transactions into a local pending queue. Successively, they run a distributed consensus protocol α that determines the order at which pending transactions are added into new blocks. The consensus α proceeds in *rounds* - *i.e.*, pre-determined periods of time; and for each round one (or more) validator is elected as leader nodes. During one round the leader l_i creates a new block b_i and proposes it to the network as next block to be appended on the blockchain. The round terminates when every node in Π updates its state with b_i. The *state* of a node is the tuple containing the last appended block b_i, and its relative height h; the heigh represents the distance between the block b_i and the genesis block. Summarising, a blockchain protocol must satisfy the following definition:

Definition 4.1. A blockchain protocol $\mathcal{B}$ is defined as a set of nodes $\Pi = \mathcal{C} \cup \mathcal{V} \cup \mathcal{L}$; a set of transactions Γ and a set of blocks Ω; such that: (i) Every valid transaction $tx_i \in \Gamma$ will be eventually inserted into a block $b_i \in \Omega$; (ii) every node in Π will append the block b_i to its local blockchain at a certain height h; (iii) Eventually, every node in Π will converge to the same state.

Notation	Definition
$\mathcal{V}$	set of validator nodes
$\mathcal{L}$	set of leader nodes, $\mathcal{L} \subseteq \mathcal{V}$
$\mathcal{C}$	set of client nodes
Γ	set of transactions submitted to the system
Ω	set of blocks processed by the the systems
α	underling consensus protocol

TABLE 4.2: *Notations used for representing a private blockchain*

d) Transactions and Blocks Lifecycle. FIGURE 4.2 illustrates the lifecycle of transactions in Γ and the blocks in Ω. When a new transaction tx_i is submitted, it enters in a *submitted (s)* state; at this time $tx_i \in \Gamma$. tx_i is firstly validated by the client hence accepted or rejected according to the function $\gamma(tx_i)$; if $\gamma(tx_i) = False$ then the state of tx_i is *rejected (r)*, otherwise the transaction is accepted. Accepted transactions are sent to validators that add them to a local pending queue. Transactions hold in a *waiting (w)* state until they are inserted into a new block. Periodically, a leader creates a new block b_i with some waiting transactions; at this time, $b_i \in \Omega$. When every node of the network appends b_i to its local copy of the blockchain, than the state of b_i (and of all its transactions) is *confirmed (c)*. Being the network distributed, the new block might reach different nodes at different points in time. In some cases this leads to blockchain *forks*. The blockchain protocols handle forks with two distinct approaches, either by implementing an ad-hoc fork-resolution mechanism, or simply denying by design the possibility of forks [104]. The former maintains new blocks into a commit state until the fork is solved, then it consider blocks as *finalised (f)*; the latter confirms and finalises blocks simultaneously, having new blocks instantly with *finalised* state. The protocol correctly terminates when all the transactions in Γ (and the blocks in Ω) get finalised. Transactions (and blocks) can not hold in a waiting (neither confirmed) state forever.

For this reason, α usually embeds a timeout for the finalisation time. If the timeout expires for a certain transaction (or even block), it is considered *rejected (r)*.

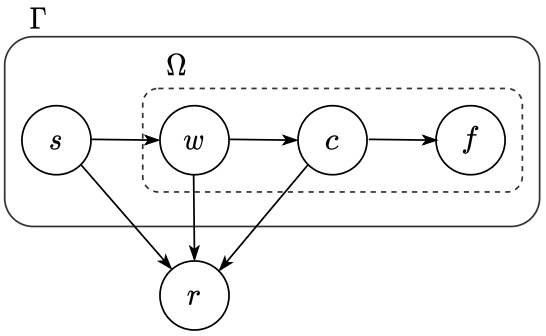


FIGURE 4.2: *Transactions and Blocks lifecycle*

4.2.1 Security Properties

A blockchain must act as a dependable, resilient and secure system, distributing trust over the network even in presence of faults and malicious nodes (Chapter 2). This is accomplished by running a BFT consensus protocol that ensures transactions order and fault tolerance, assuming that at most f validators are Byzantine [33]. Non-Byzantine validators are called *honest*, or *correct*. We refine the blockchain consensus protocol properties introduced in Section 2.3:

> *Persistence*: If an honest node appends a block b_i to its local blockchain at a certain height h, such that $state = (b_i, h)$, then b_i will be appended at heigh h of any other honest node; forks are eventually resolved;
>
> *Termination*: If a valid transaction tx_i is accepted by a validator, eventually tx_i will be inserted into the local blockchain of every honest node;
>
> *Finality*: if a correct node appends a block b_i to its local blockchain, eventually b_i will be finalised and never removed or modified.

4.2.2 Threat and Attacker model

We define a threat model based on the general definition of Byzantine fault model introduced in Section 1.1.2. It considers three types of failures, *i.e.*, the *omission failure*, *timing failure*, and *Byzantine failure*. The former two failures are caused by crash faults, link breakdowns, or network partitions; the latter by nodes acting maliciously against the protocol. A Byzantine failure is the most general type of system fault. We consider a blockchain $\mathcal{B}$ communicating over network subject to partitions caused by unpredictable events like link breakdowns or message delays. A network partitions

causes timing failures, in which the set Π is divided into disjoint groups that cannot communicate. Eventually, the partition is resolved after a GST and the connections between the groups restored. In addition to partitions, we consider the scenario in which an adversary aims at compromising the consensus protocol of the blockchain system taking control of f validators. Subverted validators start behaving arbitrarily, and we consider them as Byzantine. We assume the client nodes to be always honest because they are not involved into the consensus protocol.

Attacker model. The attack surface of complex systems like the blockchain, is extremely wide. A Byzantine node could assume a multitude of arbitrary and unpredictable behaviours. Proving the security in any scenario is a challenging task, requiring the definition of complex models and formal mathematical reasoning. In this work, we consider the most general scenario of a Byzantine failure in which an adversary takes advantage of *adversarial conditions* to corrupt the security properties ot the system. The adversarial condition is modelled as follows: the consensus protocols accomplish its task by reaching a quorum, the largest is the set of nodes who participate in consensus, the hardest is the quorum to compromise. Nevertheless, in the event of network partitions, reaching such a quorum could be challenging due to communication issues. In this scenario, an adversary could attempt to attack the protocol. Hence, we define an adversarial condition when the attacker attempts to control up to f validators of the blockchain during a network partition. We model an attack with a *pure* Byzantine failure (*i.e.*, unrelated to omission or timing faults) considering a *corruption attack*. Such an attack aims at corrupting the messages exchanged by validators, attempting to compromise the consensus protocol and its security properties.

4.3 METHUS Framework

METHUS is a framework that establishes a standardised methodology for conducting comprehensive analysis and comparative studies of blockchain consensus protocols. It evaluates the protocols behaviour in terms of *performance* and *security* guarantees, and their tradeoffs, under untrustworthy deployment scenarios like the Internet. In literature, a lot of efforts have been devoted to define those properties in blockchain systems [161, 176, 197]. Accordingly, METHUS proposes *qualitative* and *quantitative* methods to validate such properties respectively under analytical and experimental evaluations. The METHUS approach is summarised in TABLE 4.3 which illustrates how performance and security are represented and measured. Performance is divided in *throughput*, *latency* and *resources utilisation*; security is divided in *safety* and *liveness*. The throughput is the number of transactions processed by the system per unit of measured time, the latency is the time delay between a transaction request and its finalisation while the resources utilisation is the usage of CPUs and Memory.

		Qualitative Approach	Quantitative approach
Security	*Safety*	Validated with *consistency* metric of CAP Theorem	Measured with metrics of *integrity*, *consistency*
	Liveness	Validated with *availability* metric of CAP Theorem	Measured with metrics of *liveness*
Performance	*Throughput*	Algorithm Complexity	Measured with metrics of *throughput*
	Latency	Measured with number of message hops per commit	Measured with metrics of *latency*
	Utilisation	n/a	Measured with metrics of *utilisation*

TABLE 4.3: *METHUS framework - security and performance measurements with qualitative and quantitative approaches*

The safety and liveness were firstly introduced in [109] to prove the correctness of multi-process systems. Typically, safety represents the correct execution of a protocol, *i.e.*, a correct input produces a correct output; while liveness indicates the correct (eventual) termination of a protocol (Section 1.1). Given theorem 4.1, we state that a blockchain protocol $\mathcal{B}$ violates safety if (i) two nodes never converge to the same state - forks are unresolved, or (ii) if blocks appended to the blockchain are altered. Furthermore, $\mathcal{B}$ violates liveness when a valid transaction is never finalised. METHUS decomposes the safety and liveness into the blockchain security properties presented in section 4.2.1. Specifically, persistency and finality are intended to preserve safety while termination preserves liveness. This distinction is used in both qualitative and quantitative approaches of METHUS.

4.3.1 Qualitative approach

The METHUS qualitative approach defines the methods to theoretically analyse performance and security of a blockchain consensus protocol. Specifically, this approach has two main objectives: (i) a CAP Theorem-based security analysis, (ii) a complexity-based performance analysis.

CAP Theorem-based Security Analysis. The CAP Theorem [81], states that in a distributed protocol only two out of the three properties can be ensured from: *Consistency* (C), *Availability* (A) and *Partition Tolerance* (P). Hence, any protocol can be characterised based on the (at most) two properties it can guarantee, either CA, CP, or AP. Accordingly, METHUS applies this theorem to characterise the security of a blockchain consensus protocol. The CAP consistency is a safety property, represented in METHUS with persistency and finality; while availability is a liveness property and is represented as termination. The terms consistency, availability and partition tolerance of the CAP are thus refined for a blockchain consensus protocol as follows:

Consistency: Achieved if and only if the persistency and finality properties are guaranteed, otherwise the system has *no consistency*. In case of temporary forks, the system has *eventual consistency*;

Availability: Achieved if and only if the termination property is guaranteed;

Partition Tolerance: A system tolerates partitions if it continues to deliver consistency or availability even in presence of network partitions.

The analysis characterises the blockchain consensus protocols according to the revised properties above. Assuming an Internet-deployed blockchain, each protocol must be evaluated considering these adverse situations: (i) network partitions where the network behaves asynchronously; (ii) a (bounded) number of f Byzantine validators carrying the *corruption attack*. In this scenario, a blockchain must ensure partition tolerance (P), giving up either consistency (C) or availability (A).

Complexity-based Performance Analysis. The qualitative measurement of performance involves two metrics such as throughput and latency. The throughput metric measures the asymptotic upper bound of the communication complexity of the protocol; the latency metric measures the number of message hops required by a process to terminate a protocol round. The evaluation must consider $n = |\mathcal{V}|$ validators that participate in the consensus protocol, and the execution of a single round of the protocol. Both metrics are measured assuming a period of synchrony in which messages are correctly delivered.

4.3.2 Quantitative approach

The METHUS quantitative approach establishes a technique to measure performance and security metrics for blockchain consensus protocols by mean of experimental evaluation. It consists in the execution of an *offline-based testing technique* composed by four phases, namely *(i) deploy, (ii) load, (iii) collect*, and *(iv) measure*. The goal is to stress a consensus protocol with a controlled workload, and successively extract consensus-related data from the system logs of the validators, *i.e.* the nodes who participate in the consensus. This data must be stored and processed *offline*, *i.e.*, outside the blockchain system. METHUS uses such an offline-based technique to separate *computation* to *measurement*. In literature there exist several works using a real-time measurement approach [14, 55, 88], in which data is acquired implementing complex monitoring systems (like polling services or profiling). Generally, this approaches flood the blockchains with hundreds of request toward the APIs exposed by the systems, leading to communication and computation overheads that consequently affect the measurements. Differently, an offline approach avoids those overheads, guaranteeing more accurate data.

Offline-based Testing Technique. The first phase of the offline technique is the *deploy*. It defines the setup of the blockchain system hosting the consensus protocol to be tested.

This phase specifies the nodes who participate in the network, such as the set of clients $\mathcal{C}$ and validators $\mathcal{V}$ (the leaders in $\mathcal{L}$ are a special subset of $\mathcal{V}$); and the network topology. The nodes need to be geographically spread around the world, and the communications delayed. The number of validators $n = |\mathcal{V}|$ is obtained considering at most f Byzantine nodes. Typically, consensus protocols guarantee BFT assuming at most $f < n/k$ nodes become Byzantine, for some $k = 2, 3, ..$ (the value k is determined by the protocol). Therefore, we consider $n = kf + 1$ the optimum number of validators to run a blockchain. Successively, the *load* phase determines the set of transactions Γ, sent over the blockchain as a *workload*, during a certain time frame, called *load duration*. It needs to be equally balanced among $\mathcal{C}$ to avoid imbalances that may affect measurements. During the workload, the blockchain network starts processing transactions by running consensus. At this stage, the protocol needs to be stressed to observe its behaviour under adversarial conditions, and its capacity to recover. The METHUS' approach consists in simulating for a certain time-window, called *attack duration*, the adversarial scenario presented in Section 4.2.2. The test must have an attack duration lower than the load duration, in this way we can observe how protocols recover from the attacks. After the load phase, we look at validators local queues. If all the queues result empty, then every transactions in Γ has been either finalised or rejected by the system. At this time the *collect* phase can start. This phase collects the validator logs which are then stored offline and processed to extract consensus-related data. Finally, these data are used in the *measure* phase to compute the METHUS performance and security metrics.

Data collection and metrics. The logs need to be processed by a parser to extracts relevant data. Such a parser can be any kind of external software module that takes in input the logs of the validators and produces as output a dataset D containing the timestamps ts^{δ} of every transactions' state. For each state δ, the parser collects as many timestamps as the number of validators $n = |\mathcal{V}|$: $ts^{\delta}_n = (ts^{\delta}_1, ts^{\delta}_2, .., ts^{\delta}_n)$; where $ts^{\delta}_i < ts^{\delta}_j$ iff $i < j$. Given that at most f validators may be Byzantine, for each timestamp the parser must keep the $k - th$ latest value ts^{δ}_k, such that $k = \frac{n-1}{f}$. With this approach we ensure that for each timestamp the parser stores in D the earliest value produced by honest nodes. Summarising, D must be composed by the tuple $(tx_i, b_j, b^n, ts^s, ts^c, ts^f)$; such that tx_i is the i-th transaction in Γ, b_j is the j-th block in Ω storing tx_i, b^n is the block number[5], ts^s is the timestamp of tx_i submission, ts^b is the timestamp of tx_i confirmation, and ts^f is the timestamp of tx_i finalisation. According to these data, METHUS defines the following performance and security metrics:

- *Integrity*: It measures the persistency and finality properties of a blockchain consensus protocol. It is as a boolean value $(True, False)$. Integrity is $True$ if at the end of a test any correct node has the same state of the blockchain;

[5]The block number also represents the blockchain height.

- *Consistency*: It measures the persistency property of a blockchain consensus protocol. It can assume three values: (i) *strong consistency* if forks never happen, (ii) *eventual consistency* if forks may happen but are eventually resolved, (iii) *no consistency* if forks may happen and are never resolved. If $Integrity = True$ then consistency is guaranteed and can be either *eventual* or *strong*.
- *Liveness*: It measures the termination property of a blockchain consensus protocol. A transaction tx_i reaches termination if and only if the timestamp $ts^f_{tx_i}$ is not null.
- *Latency*: It measures the finalisation time of a transaction tx_i. Because blockchains store transactions into blocks, latencies are calculated as the average latencies of a block, such as $block\_latency = \frac{1}{bs}\sum_{i=1}^{bs} latency_{tx_i}$, with bs = block-size, and $latency_{tx_i} = ts^f_{tx_i} - ts^s_{tx_i}$.
- *Throughput*: It measures the number of transactions successfully finalised per second (TPS).
- *Utilisation*: It measures the CPU and memory usage (percentage) of any validator.

4.4 Consensus Protocols Description

We evaluate METHUS by assessing performance and security of three consensus protocols, namely two PoAs, *i.e.*, *AuRa* [148] and *Clique* [156], and one PBFT-like protocol, *i.e.*, IBFT [135]. In this section we describe in depth their functioning and main components. All three protocols are implemented into Ethereum-based blockchains. Respectively, AuRa is provided by Parity [151], while Clique and IBFT are both provided by GoQuorum [43]. Parity is the Ethereum blockchain platform implemented with `Rust` programming language; GoQuorum is an extension of Geth [82], offering privacy features in private networks. We consider a private blockchain $\mathcal{B} :=< \mathcal{V}, \mathcal{L}, \mathcal{C}, \Gamma, \Omega, \alpha >$ with $n = |\mathcal{V}|$ validators and $m = |\mathcal{C}|$ clients, in which $\alpha = \{AuRa, Clique, IBFT\}$. For sake of simplicity, we assume any transaction $tx_i \in \Gamma$ to be valid (*i.e.*, $\gamma(tx_i) = True$) and added to the set Γ.

4.4.1 AuRa Proof-of-Authority

AuRa (Authority Round) [148] is the PoA protocol implemented in Parity [151]. AuRa's native implementation assumes a synchronous network where all the nodes are synchronised within the same UNIX time. In contrast to this, we consider the realistic scenario where the blockchain system is deployed over a partially synchronous network and all the nodes can be desynchronised. AuRa is a mining rotation algorithm, where each

round of the protocol is characterised by one leader $\mathcal{L} = \{l\}$. Leaders are elected deterministically by validators that calculate the current round r of the protocol according to their UNIX time t_{v_i} with eq. (4.1)

$$r = \frac{t_{v_i}}{block\text{-}period} \tag{4.1}$$

where:

block-period: is a constant value (in seconds) defined in the genesis block. Given r, the leader l is the validator v_i such that $\{l = v_i | v_i \in \mathcal{V}; i = r \ mod \ n\}$.

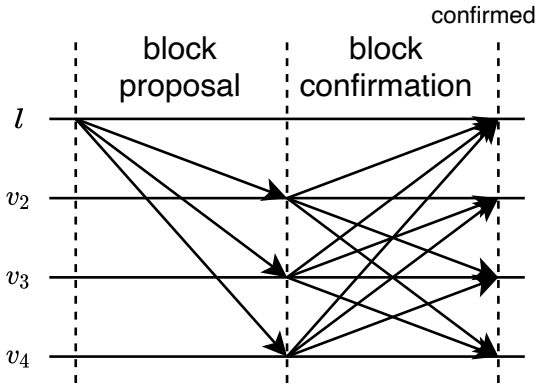


FIGURE 4.3: *AuRa PoA message exchange in one consensus round*

Each round of the protocol is splatted in two phases, (i) the block proposal, and (ii) the block confirmation, in which validators make use of two separated queues, one for transactions Q_{tx} and one for blocks Q_b. Any submitted transaction is stored into Q_{tx}, entering in its *waiting* state. Then, for each round, the leader l creates a new block b picking some transactions in Q_{tx}. FIGURE 4.3 shows the message exchanges of both phases. At the beginning of the *block proposal* phase the leader l proposes a new block b to the network. Hence, when the other validators receive b, they enter in the *block acceptance* phase. Here, the validators exchange b to verify that they received the same block for that specific round. If so, the block is confirmed. Confirmed blocks are enqueued into Q_b with state *confirmed*. A block b holds in such state until it is finalised. AuRa relies on block finalisation to guarantee BFT. It ensures that a majority of honest validators agree on the same blockchain state in each round. The finalisation is a fundamental process in AuRa because it guarantees BFT until a majority of validators remains honest, *i.e.* $n \geq 2f + 1$ [148]. AuRa finalisation is detailed in theorem 4.2.

Definition 4.2. ***AuRa Finality.*** Given a set of $n = |\mathcal{V}|$ validators and a confirmed block b, then b is considered *finalised* if and only if a majority of $\frac{n}{2} + 1$ further blocks are confirmed.

AuRa's finality approach prevents that any minority of Byzantine leaders successfully finalises blocks. We represent and therefore evaluate theorem 4.2 with eq. (4.2).

$$finality_{AuRa} = block\text{-}period * (\frac{n}{2} + 1) \tag{4.2}$$

AuRa has the possibility to detect Byzantine behaviours by implementing a smart-contract based voting mechanism [150][6]. In this case, a voting is triggered by validators against the malicious leader l_m. In AuRa a leader is considered malicious if (i) it has not proposed any block, (ii) it has proposed more blocks than expected, (iii) it has proposed different blocks. If the voting reaches a quorum, then l_m is kicked out from the validators set, and all the blocks in Q_b previously proposed by l_m are discarded by validators, before getting finalised - *rejected* state.

4.4.2 Clique Proof-of-Authority

Clique [156] is the PoA algorithm implemented in GoQuorum [43]. Differently from AuRa, here is introduced the concept of *epochs*, *i.e.*, a prefixed sequence of rounds. When a new epoch begins, a special *transition block* is propagated through the network. It specifies the set of validators $\mathcal{V}_e$ acting in that specific epoch. Each epoch has $n_e = |\mathcal{V}_e|$ validators running the consensus. This set may change dynamically during epochs. Clique computes rounds with the blockchain height h at which a new round r begins, such that $r = h$. The validators compute the round leader l via a deterministic function that combines h, with the number of validators n_e. This function identifies the index i of the validator v_i elected as round leader; $i = h \ mod \ n_e$ such that $l = v_i$. Most of all, in addition to the round leader, other validators are allowed to propose blocks.

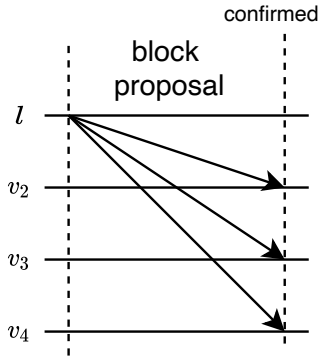
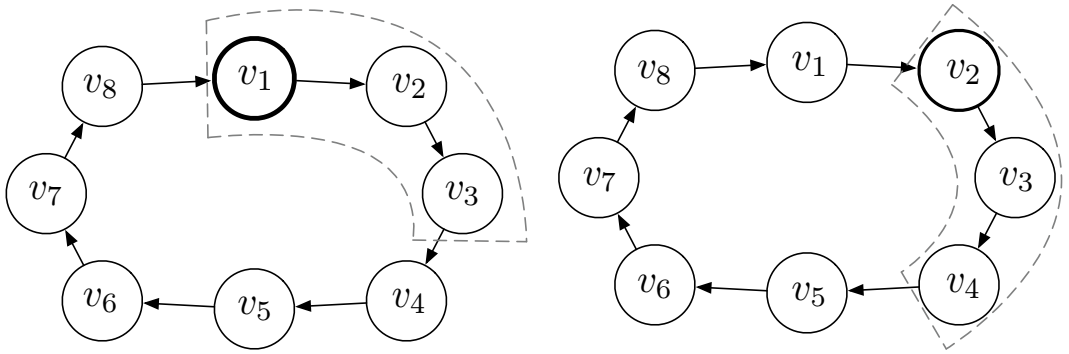


FIGURE 4.4: *Clique PoA message exchange in one consensus round*

In every round there are at most $n_e - (n_e/2 + 1)$ validators allowed to propose a new block, thus, at any point in time a subset of validators is part of the leader set $\mathcal{L} \subseteq \mathcal{V}$. FIGURE 4.4 shows the message exchange in a Clique round. The consensus consists in one single phase, the *block proposal* in which the round leader (or one of the allowed validators) creates and propagates a new block b. Thus, any validator receiving b just

[6]At the time of writing this mechanism was not implemented in Parity.

verifies that the sender is one of the leaders in $\mathcal{L}$, and then instantly confirms that block; *confirmed* state.



(A) Round 1 (B) Round 2

FIGURE 4.5: *Active validators election in Clique PoA*

FIGURE 4.5 shows how Clique handles simultaneous leaders considering a network with $n_e = 8$ validators. For each round, $|\mathcal{L}| = n_e - (n_e/2 + 1) = 3$ validators allowed to propose a new block, of which one is acting as round leader (marked circle). In FIGURE 4.5a, the leader is $l = v_1$, while v_2 and v_3 are allowed to propose blocks. In FIGURE 4.5b, v_1 is removed from $\mathcal{L}$ (it was in the previous round, so it has to wait for $n_e/2 + 1$ further rounds), while v_4 joins $\mathcal{L}$ and is now authorised to propose a block. Round's 2 leader is $l = v_2$. As in one round more validators can be leader and propose a block, forks may occur. However, forks likelihood is limited by the fact that every block sent by non-leader, allowed validators, is subjected to a random delay. According to this, the block proposed by the round leader is likely to be the first received by all the validators. In the event of forks, the Ethereum GHOST protocol [175, 194] is applied, which is based on a block scoring approach: at the end of each epoch the nodes agree on the blockchain with the highest score - leaders' blocks have higher scores.

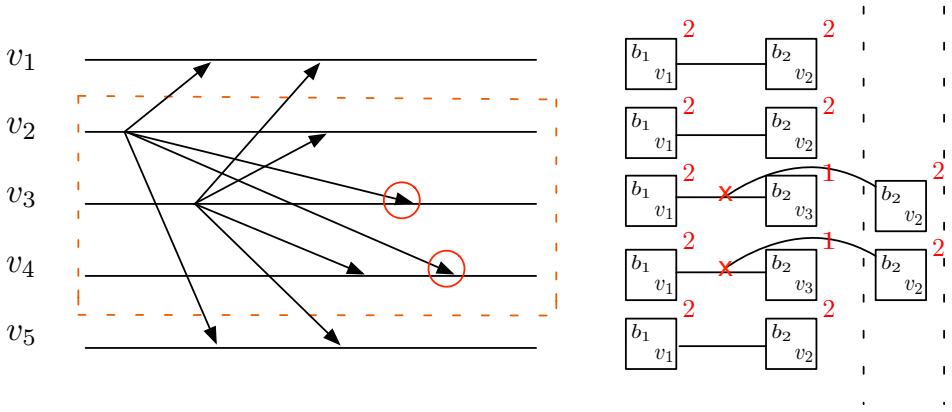


FIGURE 4.6: *A Clique fork resolution using the GHOST protocol*

FIGURE 4.6 depicts a simplified execution of the GHOST protocol, considering a round with $\mathcal{L} = \{v_2, v_3\}$, and round leader $l = v_2$. l proposes a new block, as well as the other allowed validator v_3. The former block precedes the latter in the views of v_1 and

v_5, while the opposite occurs for v_4, and v_3. The resulting fork is easily detected by each validator when the next block is received, as it references a previous block, not at disposal of the validator. By relying on the scoring mechanisms (*i.e.*, blocks proposed by principal leaders), the GHOST protocol resolves the forks. Clique protocol ensures BFT as soon as a majority of validators remains honest, *i.e.*, $n_e \geq 2f + 1$. This ensures that the validators can propose a block every $n_e/2 + 1$ rounds, avoiding that single Byzantine validator could finalise a sheer number of blocks. If the validators act maliciously, (*e.g.*, proposing a block when not allowed) they can be voted out via a smart contract based voting mechanism like in AuRa[7].

4.4.3 Istanbul BFT

The IBFT [135] is a PBFT-like algorithm implemented in GoQuorum [43]. It was firstly developed by AMIS Technology [117], then updated in its more stable version [135], fixing safety and liveness issues [167, 166]. IBFT guarantees BFT as soon as there are at most $f = \lfloor \frac{n-1}{3} \rfloor$ Byzantine validators. For each round r, there is only one validator elected as leader then entitled to propose a block. The IBFT leader election is achieved either with a *sticky proposer* or *round robin* algorithm[8]. Similarly to PoAs, in IBFT, the set of validators is dynamic and can change according to a voting mechanism based on smart contracts. Moreover, as in Clique, IBFT proceeds in epochs and at the end of each epoch the new set of active validators can be specified through a transition block.

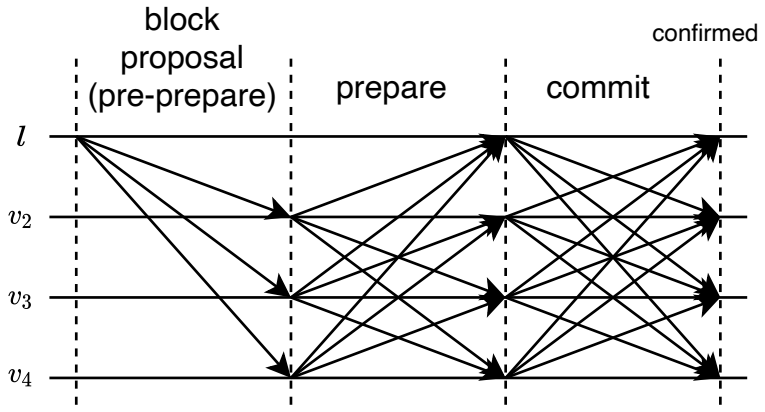


FIGURE 4.7: *IBFT message exchange in one consensus round*

IBFT achieves instant finality, forks are not allowed, hence when a new block is confirmed it is also considered finalised. Instant finality is achieved through a three-phase consensus round in which the validators exchange messages to agree on the newly proposed block. At the end of each round, all the nodes converge on the same state - which is finalised and cannot be reverted. FIGURE 4.7 shows the message exchanges of these three phases that

[7] At the time of writing, this mechanism was not implemented in GoQuorum.

[8] The way a leader is chosen does not impacts the IBFT consensus process. We leave to the reader deepen the differences between these approaches.

proceed as follows: for any round r, one node is elected as leader l from the validators. The leader l prepares a new block b and multicasts a PRE-PREPARE message to all the validators, including the block b and the current round r. When a validator $v_i \in \mathcal{V}$ receives a valid PRE-PREPARE message, it enters in a PRE-PREPARED state and accepts the proposed block. Then, any v_i sends to all the validators a PREPARE message, together with the current round number r and the block b. When v_i reaches a *quorum* of $\lfloor \frac{n+f}{2} + 1 \rfloor$ valid PREPARE messages, enters in the PREPARED state. At this point, every v_i realises that a quorum of honest validators agreed on b, and that b is ready to be confirmed. Hence, v_i sends a COMMIT message to all the validators. Similarly, the validators wait for a quorum of COMMIT messages. When a validators receives a quorum of COMMITs, then it confirms and finalises b.

The IBFT implements a *Round Change* protocol to guarantees BFT in case of misbehaving leaders. The round change allows consensus liveness by preserving safety, as soon as no more than f validators become faulty. Specifically, every consensus phase has a timeout. If the timeout expires in a certain round r, the validators send a ROUND-CHANGE message along with the new round $r' > r$. If in r a validator was in a PREPARED state, then the ROUND-CHANGE message will also contain the previously proposed block b for that round. Each validator who receives $f + 1$ valid ROUND-CHANGE messages moves to r'. Hence, the new leader l in r' starts a new PRE-PREPARE phase for round r' along with either the pending block b or with a new block b'.

4.5 Qualitative Comparison

4.5.1 CAP Theorem-based Security Analysis

We characterise the AuRa, Clique and IBFT protocols via the CAP Theorem properties of consistency (C), availability (A), and partition tolerance (P). Considering an Internet-deployed blockchain, P must be guaranteed. We analyse the behaviour of the protocols under the adversarial conditions presented in Section 4.3.1, observing their C and A properties. Hence, for the analysis we consider the Byzantine fault tolerance of each protocols assuming a blockchain system of $n = |\mathcal{V}|$ validators running the consensus. Under these conditions, the protocols claim to tolerate up to f Byzantine validators, such as $f = \lfloor \frac{n-1}{2} \rfloor$ for PoAs [148, 156], and $f = \lfloor \frac{n-1}{3} \rfloor$ for for IBFT [135]. FIGURE 4.8 summarises the results of our analysis, and showing that AuRa and Clique are AP protocols, while IBFT is a CP protocol.

AuRa Analysis Being AuRa based on strong assumptions on the UNIX time, in a real network validators' clocks can drift and become *out-of-synch*. Hence, there can be periods where validators do not agree on the current round and consequently on

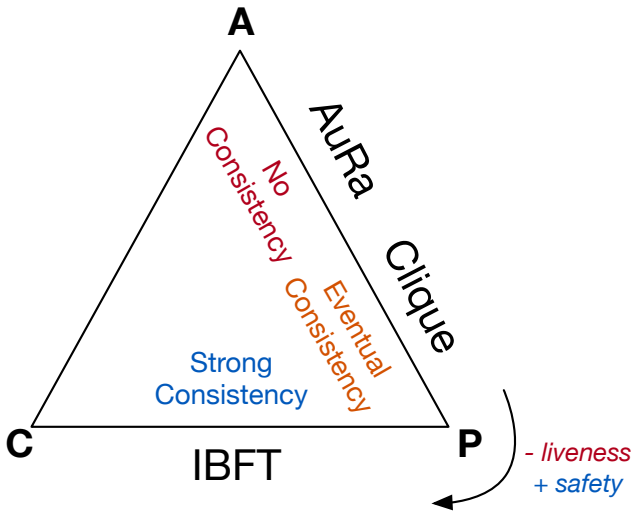


FIGURE 4.8: *AuRa, Clique and IBFT security comparison with the CAP Theorem*

the current leader in force. Clocks' skews can be reasonably assumed strictly lower than the round duration, which is in the order of seconds, thus we can have short time windows where two distinct validators are both considered as leaders by two disjoint sets of validators, say $\mathcal{V}_1$ and $\mathcal{V}_2$. This can critically affect the consistency of the whole system. Following, we describe a particular scenario in which consistency is violated. In such qualitative analysis we assume the existence of a smart-contract based voting mechanism [150] implemented with the algorithm.

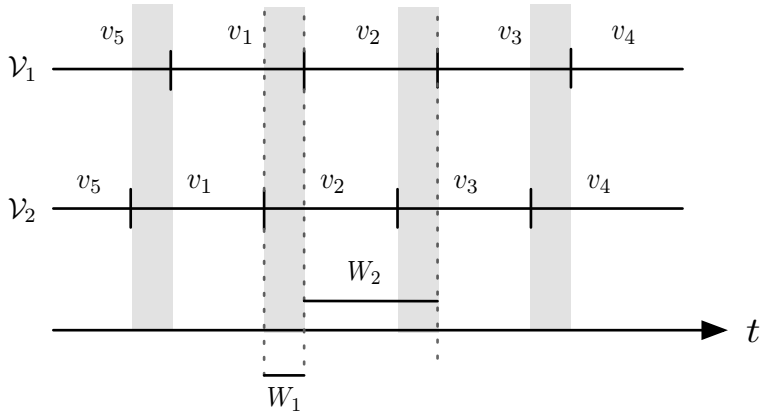


FIGURE 4.9: *Example of out-of-synch validators in AuRa (each step for a set of validators is labelled by the expected leader)*

Let $V_1 = |\mathcal{V}_1|$ and $V_2 = |\mathcal{V}_2|$ (where $V_1 + V_2 = n$ and n is an odd number) be the number of validators in the two sets, respectively. We have a majority of validators, say $\mathcal{V}_1$, agreeing on who is the current leader. This leads to a situation as depicted in FIGURE 4.9: validators in $\mathcal{V}_1 = \{v_1, v_3, v_5\}$ see rounds slightly out of phase with respect to the validators in $\mathcal{V}_2 = \{v_2, v_4\}$. Indeed, the time windows coloured in grey are those where $\mathcal{V}_1$ disagrees with $\mathcal{V}_2$ on who is the current leader. During time window W_1, v_2 considers itself the leader and sends a block to the other validators. v_2 is believed to be

the leader by the validators in $\mathcal{V}_2$ but not by those in $\mathcal{V}_1$, hence the former validators confirm its block while the latter ones reject it. During the time window W_2, validators in $\mathcal{V}_1$ expect v_2 to send a block but this does not occur because it has already sent its block for the current step: at the end of W_2 validators in $\mathcal{V}_1$ will vote v_2 as malicious and being a majority they will force v_2 to be removed. Therefore, all the remaining validators in $\mathcal{V}_2$ will be eventually voted out one by one analogously. As $\mathcal{V}_2$ is a minority, the blocks proposed by validators in $\mathcal{V}_2$ will never be finalised because they are removed from the queue of confirmed blocks, before reaching finalisation according to Equation (4.2). Summarising, this mechanism preserves consistency. However, it is to note that in this case all the validators are honest.

Differently consistency is violated when validators are Byzantine. Let us consider a scenario where there are B malicious validators, all in $\mathcal{V}_1$, that corrupt the voting mechanism used to kick out out-of-synch validators in $\mathcal{V}_2$. Hence, if $B \geq V_1 - n/2$, then a majority of votes is never reached to remove $\mathcal{V}_2$ validators, leading to the finalisation of the blocks they proposed. This event causes a fork that is never resolved: if validators are not voted out, their blocks are considered as valid and part of the chain. Summarising, in presence of desynchronisation periods AuRa is not BFT. Indeed, let us consider a set $\mathcal{S}$ with an odd number of elements $|\mathcal{V}| = 2K + 1$, and a partition of such set in two non-empty subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ with cardinality V_1 and V_2, respectively, such that $\mathcal{S}_1$ is a majority and $\mathcal{S}_2$ a minority, *i.e.*,

$$K + 1 \leq V_1 \leq 2K \tag{4.3}$$

$$1 \leq V_2 \leq K$$

$$V_1 + V_2 = |\mathcal{V}|$$

We want to prove that it suffices to remove a minority [9] of B elements from $\mathcal{S}_1$ to make it become a minority. Hence, we want to prove that

$$\exists B \mid V_1 - B \leq K \wedge B \leq K \tag{4.4}$$

Proof. Equation (4.4) can be proved by demonstrating that $V_1 - K \leq K$. This expression can be written as $V_1 \leq 2K$, which is always verified because of Equation (4.3).

[9] A minority with respect to the set $\mathcal{S}$.

Regarding availability, it is always guaranteed in AuRa regardless what a minority of Byzantine validators do. As soon as there is a majority of validators able to communicate, the protocols progresses keeping the blocks confirmed over time in that majority. Eventually, committed blocks will be finalised. Under this assumption, we can claim that AuRa preserves availability, hence can be classified as an *AP protocol*, but in specific circumstances, consistency is not guaranteed at all.

Clique Analysis By design, Clique allows more than one validator to propose blocks, assigning random delays. This permits coping with leaders that could not have sent any block due to either network asynchrony or benign/Byzantine faults. Resulting forks are anyway resolved by the Ethereum GHOST protocol. Given that, we can state that consistency is eventually guaranteed. On the other hand, as the block proposal frequency of validators is bounded by $\frac{1}{n/2+1}$, a majority of Byzantine validators is required to take over the blockchain. If the assumption of a minority of Byzantine validators persists, there will always be at least one honest leader able to propose a block, guaranteeing availability. According to these considerations, Clique can be classified as an *AP protocol*, in which consistency is *eventual*.

IBFT Analysis As long as less than one third of nodes are Byzantine, IBFT has been proven to preserve consistency due to the impossibility of having forks [117, 135]. However, in the event of a network partition the protocol may stall due to communication delays. In this period the round phases cannot proceed and blocks are not confirmed/finalised. Consequently, the validators hold in the same state until the partition is not resolved. Thanks to this behaviour, the protocol preserves consistency because it is not possible to propose, confirm, or finalise new blocks during the partition. IBFT achieves strong consistency at the cost of availability, indeed transactions may stall for periods longer than the timeout, leading to rejection. According to these considerations, IBFT can be classified as a *CP protocol*.

4.5.2 Complexity-based Performance Analysis

The performance evaluation of the AuRa, Clique, and IBFT is summarised in Table 4.4. The analysis assumes messages correctly delivered and honest validators.

	Communication Complexity	**Message hops**
AuRa	$\mathcal{O}(n)$	$2(n/2+1)$
Clique	$\mathcal{O}(n)$	1
IBFT	$\mathcal{O}(n^2)$	3

Table 4.4: *Complexity analysis of AuRa, Clique, and IBFT performance*

To calculate AuRa's complexity we analysed the message exchange in FIGURE 4.3. Each round has two phases in which (i) the leader sends a block to all the other validators, and (ii) the validators exchange the block for confirmation. The communication complexity of AuRa is determined by the second phase of a round, and it is asymptotically quadratic $\mathcal{O}(n^2)$. For latency, we consider the number of message hops required to finalise a block. A round requires 2 message hops to confirm a block. However, a block is final once a majority of further blocks have been proposed by other validators, hence the number of hops is $2(n/2+1)$. In Clique, FIGURE 4.4, we obtain better performance considering each round with one phase in which the leader sends a block to all the other validators, leading to a linear complexity of $\mathcal{O}(n)$. The block is confirmed and eventually finalised straight away - in case of absence from forks - hence the latency in terms of message hops is 1. Such a huge difference between AuRa and Clique is due to their different strategies to cope with malicious validators aiming at creating forks: AuRa waits that enough further blocks have been proposed before finalising, differently Clique copes with possible forks after they occur. Clique also outperform IBFT, which latency requires 3 message hops to finalise a block (three phases consensus showed in FIGURE 4.7) and its complexity is dictated by a quadratic number of messages $\mathcal{O}(n^2)$ [117].

4.6 Quantitative Comparison

In this section we deepen the performance and security evaluation for AuRa, Clique and IBFT, applying the METHUS offline testing technique. We firstly illustrate the deployment, design, and configuration of the blockchain system hosting the protocols; then we present the implementation details of the experiments. Finally, we illustrate the experimental evaluation discussing the measured metrics of performance and security.

4.6.1 Design and Implementation

FIGURE 4.10 illustrates the architecture we used to reproduce the *deploy*, *load* and *collect* phases of the METHUS offline-based testing technique. For the deploy, we simulated a blockchain system running into a single server via a *container-based* infrastructure using Docker [56]. The architecture is characterised by one Docker container per blockchain node. Hence, in each container we installed the softwares of Parity and GoQuorum. The containers communicate over a virtual network, called *bridge docker network*. Usually, such virtual networks make the communications between containers near to zero-latency, that is not realistic to reproduce a real scenario with nodes spread geographically over the globe. To overcome this issue we embedded into each container a network simulator, namely *NetEm* [89], that enabled us to reproduce WAN delays between containers.

For the *load* phase we used a load generator, called *Tsung* [184], able to generate distributed loads of transactions triggered from different users. We configured tsung to produce

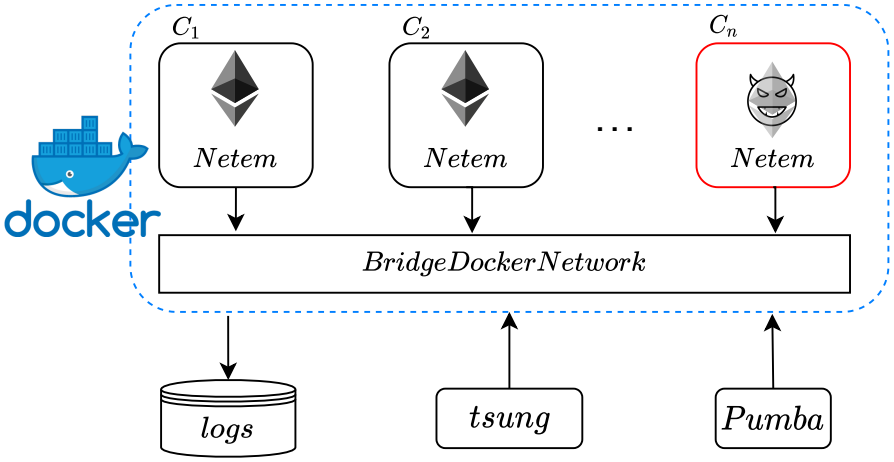


FIGURE 4.10: *Deployment of a containerised permissioned blockchain*

constant workloads of Ethereum transactions[10]. In addition, we adopted a chaos testing tool, called *Pumba* [57], for stressing the network during the load time simulating the adversarial conditions for a specific attack time duration. Pumba allows to intentionally simulate several types of failures, like network partitions and Byzantine faults, at the infrastructure level (Docker containers) and for fixed duration intervals. Finally, to accomplish the collect phase, we used a repository, external to each container, for storing the logs generated by the blockchain systems.

Ethereum parameters configuration. Notwithstanding Parity and GoQuorum differ in several functionalities and programming language, they are both founded on the main principles of the Ethereum protocol, detailed in the Ethereum yellow paper [194]. Each Ethereum client runs the EVM that is the blockchain software adopted by the Ethereum protocol to execute Turing complete smart contracts. There exist several configurable options to accomplish specific tasks and tune the EVM according to particular needs [197, 198]. In this work, we focus on those parameters that mainly concern the consensus protocols. Specifically, we focussed on the *Transaction Pool*, *Block Period*, *Validators Number*, *Server Threads*, and *Gas Limit*. A comparative study of those parameters has been proposed by the CoinFabrik community here [30].

- *Transaction Pool*: Submitted transactions are collected by the validators of the blockchain network on a local pending queue. Those transactions remain in that queue waiting to be added into a new block. This parameter determines the maximum amount of transactions holding in a queue. If the pools saturate new transactions may be rejected. This parameter must be tuned considering the available resources and the workload;

- *Block Period*: (or round duration): This parameter determines the elapsed time between two blocks. It mainly affects the throughput providing a fixed block

[10]Transaction format compatible for Parity and GoQuorum.

production. A misconfiguration of this parameter may lead to consensus failures or performance slowdowns. Indeed, short block periods may affect the rounds calculation (*e.g.* AuRa protocol) leading to desynchronisation and forks; high values produce low throughputs;

- *Validators Number*: This parameter represents the number of validators running the consensus. It straight determines the way consensus algorithms finalise new blocks. For example in AuRa this parameter affects the finalisation, while in Clique it determines the number of rounds a leader must wait before proposing a second block. Increasing the PoA validators' number lead to strong security on finalised blocks - to revert a block an attacker should corrupt the majority of validators - despite performance, affected by higher finality latencies. In PBFT-like algorithms as IBFT, more validators implies more security, but leads to higher message exchanges, thus finality slowdowns;

- *Server Threads*: This parameter indicates the number of concurrent HTTP server threads enabled by the Ethereum node. It determines the way a node can serve Ethereum's RPC requests over HTTP. Multiple RPC threads can deserialise and process the requests in parallel. The number of cores available on the server node determines the optimal number of server threads, and it is suggested by the Ethereum community to be set as $serverthreads = 2 * cores$ [145].

- *Gas Limit*: The Ethereum *Gas* is a unit that measures the computational effort required by the EVM to complete a task. The execution of an Ethereum transaction requires a certain amount of gas. The gas limit specifies the maximum amount of gas that a block can afford, determining the *block-size*. The Ethereum protocol provides a mechanism to dynamically adjust the gas limit, accordingly to the workload. However, with heavy loads this might lead to very high gas limits, hence large blocks that are difficult to propagate, affecting performance. In this case, block sizes can be bounded with a parameter called *target gas limit* that fixed the maximum gas limit allowed in a block.

Load Generation. The load generation is a crucial activity when testing a blockchain system. It determines the amount of workload to submit toward the system and it must be carefully configured to avoid imbalances that may cause unwanted overheads. A blockchain load generation must consider: *the workload source, workload distribution*, and *workload method*.

- *Workload source*: We generate the load via the Tsung distributed testing tool. This tool can simulate thousands of virtual users on a single CPU, sending concurrent requests according to ad-hoc configuration setups. The advantages of Tsung are multiple, such as high performance, multi-threading workloads and simulation of realistic traffic;

- *Workload distribution*: The workload was balanced among the client nodes. Such a balancing enabled to distribute the load and parallelise the processing of requests reducing overheads;
- *Workload method*: We configured Tsung initialising a fixed number of virtual users starting HTTP connections with the blockchain for a *load duration* period. We produced a workload of fixed rate of transactions per second within the same HTTP session. This improved the performance of the load generation reducing network overheads opening new HTTP connections.

Every request was an RPC to the JSON APIs triggering an Ethereum transaction on the blockchain[11]. The requests were sent as HTTP-RAW data with the parameter *no_ack* enabled. This parameter enabled Tsung to send requests without waiting for server responses. This was crucial to generate a constant input rate. Indeed, waiting for server responses between requests might delay the load generation due to high response times.

Chaos Testing Configuration. To simulate the adversarial scenario we used Pumba, a chaos testing tool. Pumba was configured to inject a network partition and simulate the corruption attack, lasting for a fixed time called *attack duration*. For the network partition, we used the Pumba's `netem delay` command that permits to delay the egress traffic of one or more containers. We configure specific network delays using the `target` option of the delay command, a delay variation `jitter`, and the delay `correlation` command options. Hence, for the whole duration of a partition and before its resolution, we used the `netem corrupt` command to simulate Byzantine faults by corrupting one or more container' packets. This command adds corrupted packets according to Bernoulli's probabilistic model. We configure Pumba's `netem corrupt` command with a maximum packets corruption percentage.

4.6.2 Experimental Evaluation

Testbed. We ran the private blockchains using Parity *v2.4.5-stable*, and GoQuorum *v2.6.0-updated_ibft* releases. We embedded both releases in custom container images. embedding into each image the NetEm simulator [8, 9]. The experiment was run on a PowerEdge R730xd rack server with 56 logical processors Intel(R) Xeon(R) CPU E5-2695 v3 2.30GHz, with a VMware ESXi hypervisor. We ran two virtual machines, one for the blockchain network and another for the workload generator. The first VM had 20 CPUs, 80GB RAM, 300GB hard drive, and for each container, we dedicated 2 CPUs and 16GB RAM. The second VM had 8 CPUs, 8GB RAM, 50GB hard drive. Both VMs ran Ubuntu *18.04.2 LTS*, in addition the former also ran Docker *v18.09.7*, while the latter Tsung *v1.7.0*.

[11]The Ethereum APIs provide a number of functions to interact with the blockchain.

Blockchain Setup. We initiate three blockchain networks running respectively AuRa - on Parity; Clique and IBFT - on GoQuorum. Accordingly to the available resources, we create a network of $|\Pi| = 4$ nodes $\mathcal{C} = \mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each one running in a docker container. For the sake of simplicity we deployed a network composed just by validators[12]. We emulate a realistic connection link between nodes where communication are subject to latencies. We configure Netem to delay the egress traffic of each container setting the parameters $delay = 100ms$, $jitter = 0.5ms$ (random variation) and $correlation = 20\%$. This parameters has been selected emulating the `ping` times between Rome and New York. To obtain these values we ran 100 ping iterations from Rome to New York and averaged the observed the delay, jitter and correlation. Hence, we create one blockchain account $< Pk, Prk >$ on every validator, assigning an unlimited balance of Ether. The configuration of blockchain's parameters is not an easy task and requires deep analysis and evaluations that are out of the scope of this work. Here used the *optimal* configuration setup suggested on the official documentations [43, 151] and by the Ethereum community [145]. so that: *transaction pool* set to 32768 transactions, a *block period* of 5s, 8 *server threads*, a *gas limit* and *target gas limit* set to 40M gas units. For the sake of simplicity, in these experiments, we configured the set of validators as static, and we did not integrate any smart-contract voting mechanism used to kick-off (or add-in) malicious validators. We believe that such a mechanism, as is, introduces computation and message overheads skewing the measurements. Indeed, this procedure can be achieved by adding a software layer based on smart contract on top of the consensus phase, but it is nor part of the consensus itself. We leave the evaluation of such a component and its impact to the performance and security of consensus protocols as a future work.

Workload. We produced a workload of $300req/s$ balanced over the four validators, for a load duration of 4 minutes. Each request was an HTTP call to the Ethereum's JSON RPC API `eth_sendTransaction` that implements a simple atomic swap[13] transaction. We chose such a simple transaction to evaluate the behaviour of each consensus protocol avoiding possible biases caused by more complex operation involving computation (like smart contracts). The gas of an `eth_sendTransaction` transaction was 21k gas units. The workload was tuned in accordance with the maximum amount of TPS that an Ethereum blockchain can afford, *i.e.*, TPS_{max}. This value can be computed using eq. (4.5):

$$TPS_{max} = \frac{Gas_{limit}}{Tx_{gas} * Block_{period}} \tag{4.5}$$

Considering the blockchain setup above, we obtain $Gas_{limit} = 40M$, $Block_{period} = 5s$ that combined with the transaction gas $Tx_{gas} = 21k$ produced a $TPS_{max} \approx 380,8$.

[12] Validators acting as clients do not impact the consensus protocol.

[13] Exchange of a fixed amount of Eth between two blockchain accounts.

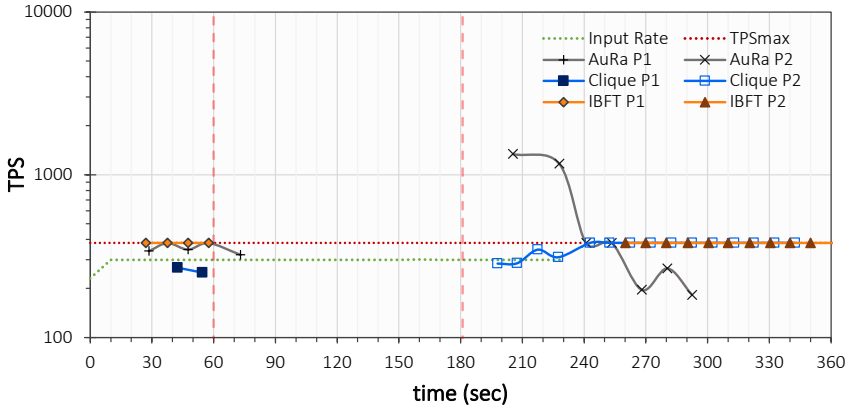
Assuming a workload of $300 req/s$ every node was expected to achieve the maximum throughput without saturating the queue. Full queues lead to transactions loss not dependent by the consensus protocol, that are out of the scope of our measurement.

Experiment Configuration. We ran one experiment per consensus protocol following the METHUS offline-based testing technique. The *deploy* and *load* phases of the technique were implemented in accordance to the above configurations. Each experiment emulated the adverse scenario that injected one network partition and one Byzantine fault during the workload generation. Specifically, for the duration of 2 minutes (attack duration lower than load duration) the network was partitioned in two groups $\mathcal{V}_1 = \{v_1\}$, and $\mathcal{V}_2 = \{v_2, v_3, v_4\}$ with $f = v_2$ acting maliciously. We observed how such an adverse condition impacted each consensus protocol in terms of performance and security. For this purpose, we divided each experiment into two macro periods named the *pre-partition* and *post-partition*. We will refer to this periods respectively as P1 and P2. To carry on with the *collect* phase, we implemented a polling service querying validator's local queues. This service was started immediately after the load duration, and terminated once all the queues resulted empty. This was used to terminate the experiment once all the transactions were processed. Afterwards, we acquired the logs of the four validators into a local repository of the server. Hence, we processed such logs using a Python script parsing data into a dataset composed by the tuples $< tx_i, b_j, b^n, ts^s, ts^c, ts^f >$. We generated three datasets D_{AuRa}, D_{Clique}, and D_{ibft}. For each of dataset we measured the performance and security metrics of METHUS' *measure* phase.

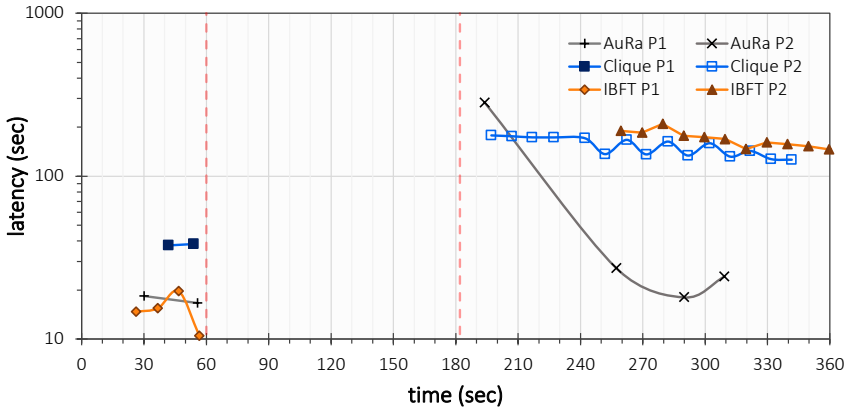
Performance Analysis

FIGURE 4.11 shows the *throughput* and *latency* measured during the load phase. Both graphs depict two vertical lines representing the macro periods P1 and P2 . The time frame between P1 and P2 indicates the attack period in which the validators are partitioned into $\mathcal{V}_1$ and $\mathcal{V}_2$, and the validator v_2 is Byzantine.

Throughput in P1. FIGURE 4.11a shows that during P1 AuRa's TPS approached the TPS_{max}. At the end of P1 the network got partitioned and v_2 subverted. At this point we observe that AuRa stopped producing TPS because transactions never get finalised neither in $\mathcal{V}_1$ nor in $\mathcal{V}_2$. Both groups lacked of a majority of validators able to finalise blocks according to the Formula eq. (4.2). However, AuRa obtained some TPS even right after P1. Indeed, within the attack period AuRa's honest validators continued confirming blocks that eventually get finalised. From the graph we can see how IBFT instantly finalised blocks until there are $3f + 1$ honest validators. Under this conditions the protocol achieved maximum throughput, TPS_{max}. After P1, there were not enough honest validators to execute the protocol which got stuck stopping finalising transactions; during the attack duration IBFT $TPS = 0$. Clique allows forks which eventually



(A) Throughput



(B) Latency

FIGURE 4.11: *Throughput and Latency of AuRa, Clique and IBFT*

get resolved with the GHOST protocol. This process caused hard blockchain resynchronisations between nodes, generating overheads. This overheads produce latencies in transactions finalisation affecting the TPS. We observe this behaviour in FIGURE 4.11a, where at the beginning of P1 Clique TPS was lower than TPS_{max} because the existence of forks. After P1, Clique TPS was interrupted because the remaining honest validators cannot cope with the condition of proposing a block every $|\mathcal{V}|/2+1=3$ rounds.

Throughput in P2. In P2 we observe the protocols recovering from the attack period. AuRa's throughput reached a peak of $TPS = 1000$ at the beginning of P2, and then collapses below the TPS_{max}. Such a peak included all the transactions confirmed by the honest validators in $\mathcal{V}_2$. After the attack, v_2 returned honest proposing a new block that caused the instant finalisation of the previous confirmed blocks. After the input rate, AuRa's approached $TPS = 0$ following a sinusoidal progression. Indeed, looking at the logs we noticed that in P2 the partitioned node v_1 never resynchronised, and all its transactions never finalised. This caused in P2 less TPS than expected. Conversely, during the attack period both Clique and IBFT got stuck. While in P2, the protocols' TPS approached the optimum TPS_{max}, nevertheless from the graph we observe different TPS progressions. Indeed, IBFT recovered slowly than Clique due to its three-phase

consensus. Notwithstanding Clique recovered faster producing TPS earlier than IBFT, the TPS took some time to reach its maximum value. Again, this behaviour was caused by the network overheads of the GHOST protocol.

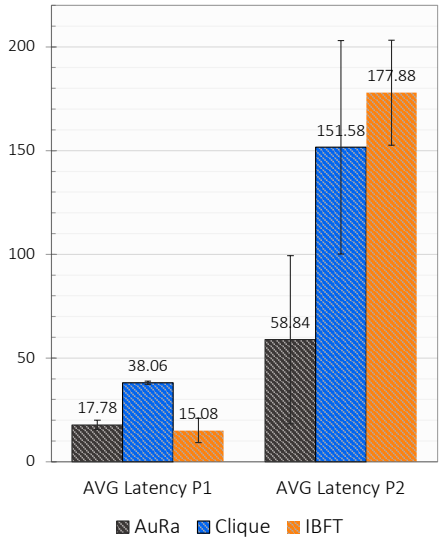


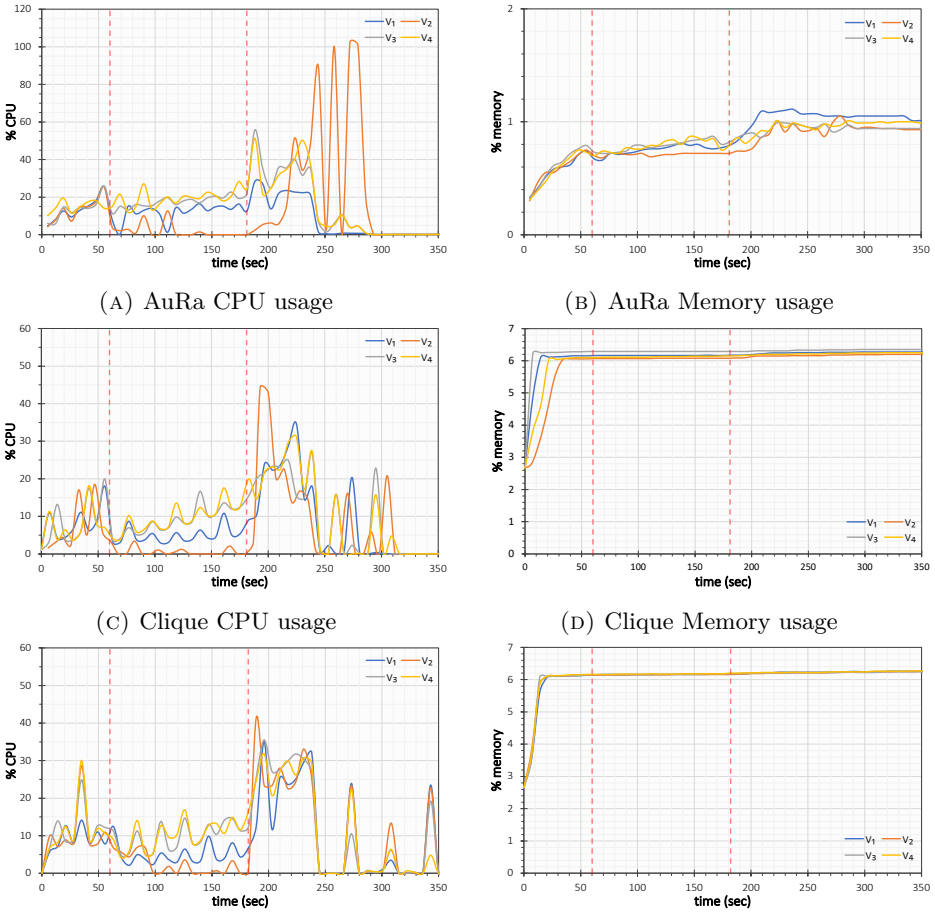
FIGURE 4.12: *AuRa, Clique and IBFT average latencies*

Latency in P1. We noticed blockchain latencies to be affected mostly by the workload and the block period. Indeed, in presence of heavy loads, blockchain systems struggled deserialising and processing requests. Requests started queueing and consequently transactions finalisation times increased. Moreover, the block period determines the frequency of transactions confirmation straightforward. In our experiment transactions got finalised every five seconds having $block\text{-}period = 5s$. What distinguishes latency performance is the way a protocol achieves finality. From FIGURE 4.11b and FIGURE 4.12 we observe AuRa's latency approaching on average 18s which is optimal considering the block period and the finalisation time driven by eq. (4.2). In IBFT the latencies ranges between 10s and 20s due to the block period plus the time needed by the three-phase commit process. Differently, due to the fork resolution overheads, Clique's transactions struggled to achieve finality, reaching latencies around 40s.

Latency in P2. When the network partition is resolved the validators start a re-synchronisation process. At this point we observed in AuRa all the confirmed blocks proposed during the attack period to be instantly finalised. This caused in graph a latency progression starting very high and gradually returning to its normal values. Such latencies included all those transactions confirmed during the attack period, together with the remaining transactions confirmed in P2. Differently, in this phase both Clique and IBFT latencies drastically increased because during the attack period, the protocols stalled.

Resources Utilisation

FIGURE 4.13 shows the CPU and memory consumptions for AuRa, Clique and IBFT. In P1 AuRa' resource consumptions remained moderate, with a CPU usage around 50% of its capacity and a very low memory consumption - around the 1% - FIGURE 4.13a and FIGURE 4.13b. It is notable that v_2's CPU consumptions strongly differ from the others. During the attack period v_2 assumed a Byzantine behaviour without following the protocol explaining a lower CPU consumption. Differently, v_2's CPU peaked three times in P2 because the validator returned honest starting proposing new blocks and finalising all the pending confirmed blocks.



(A) AuRa CPU usage

(B) AuRa Memory usage

(C) Clique CPU usage

(D) Clique Memory usage

(E) IBFT CPU usage

(F) IBFT Memory usage

FIGURE 4.13: *AuRa, Clique and IBFT resources utilisation*

On the other hand, Clique and IBFT obtained similar resource consumption because both protocols were executed under the same blockchain system[14]. Looking at FIGURE 4.13c and 4.13e, the CPU consumptions of both protocols slightly decreased right after P1, and then grew drastically reaching a peak after P2 due to resynchronisation operations. In terms of memory consumptions, FIGURE 4.13d and FIGURE 4.13e show that both protocols required a memory consumption almost constant - up to 6%.

[14]GoQuorum.

Security Analysis

Evaluation of Safety. Safety was measured validating the security properties of *persistence* and *finality* via the metrics of *integrity* and *consistency* (TABLE 4.3). Respectively the former evaluates finality looking at the state of the blockchain at the end of the experiment; the latter evaluates persistency monitoring the presence of blockchain forks during the experiment. We obtained $Integrity = True$ for both Clique and IBFT, having all the validators converging to the same blockchain state. Differently, AuRa resulted in $Integrity = False$ due to validator v_1' state diverging from the others. AuRa lacked of synchrony during the attack period, leading to a fully desynchronisation of validator v_1. v_1 never re-joined the network and started diverging on its own blockchain fork.

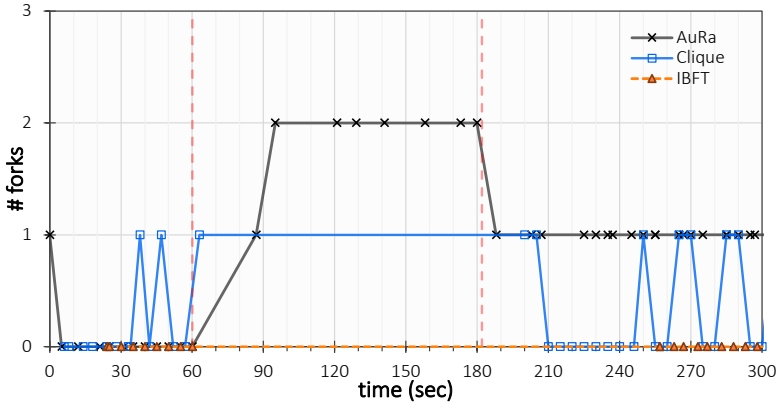


FIGURE 4.14: *Comparison of AuRa, Clique and IBFT number forks over time*

Figure 4.14 represents the number of detected forks in which we observe that AuRa achieved *no consistency* caused by the presence of one unresolved fork; Clique achieved *eventual consistency* having forks that eventually get resolved; IBFT achieved *strong consistency* thanks to the absence of forks. In TABLE 4.5, we summarised the safety analysis showing the correlation between consistency and integrity metrics.

	Consistency	**Integrity**
AuRa	No Consistency	False
Clique	Eventual	True
IBFT	Strong	True

TABLE 4.5: *Comparison of AuRa, Clique and IBFT consistency and integrity properties*

Evaluation of Liveness. Liveness was measured via the *termination* metric. This metric measures the transactions finalised within the experiment, *i.e.*, not null ts^f. Hence, given the total workload of the experiment, *i.e.*, $300 req/s$, and the load duration, *i.e.*, 4 minutes, we obtained a total of $|\Gamma| = 72000$ transactions submitted into the system for finalisation. FIGURE 4.15, shows the percentage of finalised transactions. We observe that not all transactions were finalised. Specifically, in AuRa the $18,52\%$ of transactions

were rejected due to v_1 desynchronisation in P2 - v_1 did not recover from the partition leading to termination failure, *i.e.*, transactions not finalised. Differently, Clique and IBFT rejected transactions due to a validation failure. During the attack period, v_2's transactions were considered not valid due to the corruption attack. Hence, Clique and IBFT refused respectively the $15,68\%$, and $14,61\%$ of transactions.

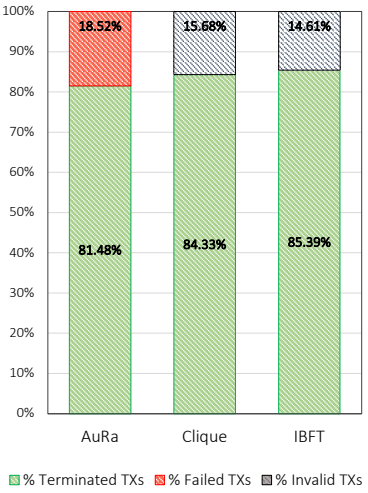


FIGURE 4.15: *Comparison of AuRa, Clique and IBFT transactions termination*

FIGURE 4.16 represents the transactions finalisation rate, *i.e.*, those transactions that (eventually) achieved finalisation. Usually, the finalisation rate of a blockchain must be equal to the input load. Observing the graph we note that AuRa's was halved during the attack period $\approx 150 req/s$. This progression suggests that in the attack phase the transactions coming from both v_1 and v_2 were rejected by the protocol. An interesting result was the AuRa's finalisation rate measured after the input. By observing the logs we realised that this behaviour was caused by validator v_2 which processed all the transaction requests submitted during the attack period afterwards. Indeed, v_2 did not processed incoming requests when subverted, however it stored them into an RPC requests pool. These mechanisms implemented by the Parity software permitted v_2 to keep requests into the pool until returning honest. Differently, Clique and IBFT finalisation rate was reduced of $1/4$ ($\approx 220 req/s$) with respect the input rate during the attack period. Indeed, looking at the logs we realised that the transactions submitted by v_2 were rejected because not valid. Therefore, differently from AuRa, in P2 both protocols were able to recover from the partition, finalising all the transactions submitted by v_1. Nevertheless, neither Clique nor IBFT finalisation rate returned to its normal value, viz. the input load. Indeed, looking at the logs of Clique and IBFT we noticed that some transactions were rejected by v_1 at P2. Specifically, v_1 entered into a `read-only` state due to network resynchronisation; in this phase the node started rejecting transactions. Moreover, for Clique the resynchronisation caused also validator v_4 to reject transactions due to

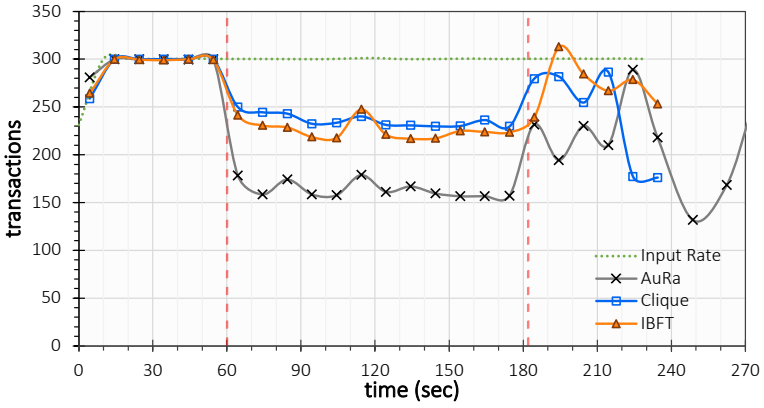


FIGURE 4.16: *Comparison of AuRa, Clique and IBFT transactions finalisation rates*

a `known transaction` error[15]. This provoked the slightly higher percentage of refused transactions with respect to IBFT, we observed in FIGURE 4.15. Such errors suggested the presence of two bugs in the version of GoQuorum leading to transactions failure in presence of heavy loads.

Overall Results. From the experiments emerged that AuRa achieves better performance than Clique and IBFT. Then, IBFT obtained overall a better performance than Clique in both P1 and P2, even if its latencies resulted the worst in P2, due to a slower recovery from the adverse conditions. However, AuRa's sacrifices security. Indeed, we showed how, in presence of network partitions, AuRa violates both safety and liveness. Conversely, Clique guarantees better security having forks eventually resolved, but at the cost of worse performance. Finally, IBFT showed a strong security ensuring both safety and liveness propertied, however this comes at the cost of highest latencies, even though in P1 IBFT's average latencies were lower than both PoAs. This surprising result undermines the most praised advantage of performance claimed by PoA algorithms.

4.7 Discussion

In this chapter, we proposed METHUS a framework and methodology to evaluate performance and security of blockchain consensus protocols. We considered a deployment over the Internet where communications are partially synchronous rather than synchronous. The proposed methodology aims at assessing consensus protocols with a lack of formal analysis and detailed descriptions, by evaluating the performance and security properties under adversarial conditions. Firstly, we define a representation of security and performance properties in the context of a blockchain system. Specifically, security in blockchain is represented with the properties of *safety* and *liveness*, while performance with the properties of *throughput*, *latency*, and *resources utilisation*. The methodology

[15]Mechanism used by GoQuorum to avoid double-spending attack.

is characterised by a qualitative and quantitative study of those properties. The former is conducted by applying the CAP Theorem and by evaluating the complexity of the message exchanges of the studied protocol. The latter quantifies the properties through experimental evaluation.

We applied METHUS to evaluate three consensus protocols used in permissioned blockchains, namely AuRa, Clique, and IBFT. Firstly, we defined a general model for permissioned blockchains so to instantiate the consensus protocols under the same conditions. Hence, we applied METHUS' qualitative and quantitative approached to evaluate and compare them. From the qualitative analysis emerged that PoA algorithms can give up consistency for availability when considering the presence of Byzantine nodes. This can prove to be unacceptable in scenarios where the integrity of the list of transactions has to be absolutely kept (which is likely to be the actual reason why a blockchain-based solution is used). On the other hand, PBFT keeps the blockchain consistent at the cost of availability, even when the network behaves temporarily asynchronously and Byzantine nodes are present; this behaviour is much more desirable when data integrity is a priority. Thereafter, we conduct a quantitative analysis to validate our claims by experimental evaluation. Hence, we ran experiments for AuRa, Clique and IBFT, and we measured their security and performance under adversarial conditions by simulating a network partition and one Byzantine node. We observed that AuRa obtains better latencies than IBFT and Clique, but it gives up data consistency. This result confirmed our claims in the qualitative analysis. In terms of security, the experiments showed how Clique favours safety despite liveness and performance. Finally, IBFT resulted in the most reliable solution when it comes to real, Internet-based, environments in which security is a paramount property. Indeed, even if during asynchronous periods the algorithm stalls and latencies increase, it always guarantees safety and liveness as soon as $3f + 1$ nodes remain active.

In conclusion, the emerged tradeoff between the security and performance of these consensus algorithms shows that AuRa obtains high performances despite security; Clique lower performance than AuRa, but with a slightly better security, and finally IBFT achieves acceptable performance, with strong security.

Chapter 5

PETHARD: Performance Evaluation of Ethereum and Algorand Consensus Protocols

In the previous chapter, we discussed the importance of consensus protocols in blockchains and how to evaluate their performance and security using both qualitative and quantitative approaches. However, the study only focussed on three protocols used in permissioned blockchains, in which there is a higher level of trust between the participants and less decentralisation. In this chapter, we move our analysis one step forward. We extend our analysis to the underlying consensus protocols designed for permissionless blockchains. These protocols operate in a fully trust-less network, bringing consensus to a new stage, *i.e.*, reaching an agreement in a fully decentralised network without relying on communication expensive, voting-based protocols. In this context, the PoW represents the backbone protocol. PoW uses *mining* to achieve consensus, an expensive operation that requires enormous computing power, leading to slow and inefficient operations for simple transactions such as payments, which require an instant confirmation for day-to-day use. An alternative to PoW, which addresses these issues, is the PPoS protocol, a novel consensus algorithm introduced with the Algorand blockchain, that claims to solve the performance issues that most blockchain platforms have. Notwithstanding the large interest in permissionless blockchains offering a secure and decentralised system to build future services and applications, the efficiency of systems based on PoW remains questionable, and conversely, most of the claims of 'superior' PoS solutions are still unverified.

Recently, a lot of effort has been devoted to analysing performance of consensus protocols [170, 199], most of these works consider mostly consensus protocols employed for permissioned blockchains or only focus on PoW performance. There is a lack of comparative studies between classic PoW algorithms against the novel, permissionless,

consensus protocols like the PPoS. To evolve from a well-established PoW solution to new generation PoS protocols like the PPoS we need new tools for evaluating and comparing their performance fairly.

In this chapter, we present *PETHARD: a PErformance benchmarking framework for Ethereum and AlgoRanD consensus protocols*. PETHARD is built following the systematic methodology proposed in the previous chapter. It enables a user to run experiments for assessing performance and scalability of PoW and PPoS consensus protocols. Specifically, it creates a standalone private network for both Ethereum and Algorand blockchains and generates custom workloads to evaluate their behaviour under various load conditions. PETHARD simulates a testing environment based on LAN networks and enables the creation of custom test cases based on various network sizes and workloads. To the best of our knowledge, PETHARD is the first tool to enable the experimental evaluation and comparison of both PoW and PPoS consensus protocols.

Contributions. The contributions of this chapter can be summarised as folllows:

- we introduce PETHARD a framework for assessing performance and scalability of the Algorand PPoS and Ethereum PoW consensus protocols; we also define a systematic benchmarking procedure run experiments on both blockchain systems;
- we evaluate PETHARD proposing as a use case, a comprehensive comparison of performance and scalability of PoW and PPoS under various configurations; we also use PETHARD to tune the block period consensus parameter employed in the Ethereum PoW.

Chapter Structure. Section 5.1 introduces the system model, so that Section 5.2 presents PETHARD, describing its underlying methodology, architecture, and benchmarking procedure. Then, Section 5.3 provides a description of the implementation, while Section 5.4 shows the experimental evaluation and comparison of the algorithms. Finally, Section 5.5 summarises the results and discuss PETHARD's limitations.

5.1 System Model

We define a System Under Test (SUT) as a blockchain *private network* representing the target system of a benchmarking *test*. A test is every operation that aims at sending a workload to the SUT and observes how the system behaves accordingly. A blockchain *private network* is characterised by a fixed set N of processes, that we call *node*s, running a *blockchain protocol*. A node of a private network can be of two types, *i.e.*, either a *network* node, or a *participation* nodes. *Network* nodes handle the communication routing between the *participation* nodes, and have a unique *network identifier*; *participation* nodes handle the blockchain protocol.

A *blockchain protocol* is a distributed computing protocol executed by the nodes of the private network. Hence, we define a *blockchain* as a distributed ledger fully replicated over the private network. The nodes collectively maintain and update the blockchain according to the execution of runtime operations. The ledger data structure consists of a list of records, called *blocks*, cryptographically linked together. Blocks are of fixed size, *i.e.*, *block-size*, and include information such as the cryptographic hash of its predecessor block, a timestamp, *i.e.*, block *finalisation time*, and a list of *transactions*. A transaction identifies an operation executed on the blockchain between two *accounts*. Blockchain *accounts* are asymmetric key pairs $< Pk, Prk >$; the public key of a blockchain account represents a unique identifier on the blockchain, and we call it *address*. Nodes organise accounts into cryptographically secured collections, *i.e.*, *wallets*. Users usually own one or more accounts and wallets and use them to interact with the protocol sending transactions requests. Transactions sent over a user requests on the blockchain are called *submitted*. Associated with a blockchain account, the protocol stores also specific data, like a *balance* of digital assets. We define a *digital asset* as a fungible token governing the blockchain core protocol. Clients can own and exchange digital assets, or use them to execute transactions, *i.e.*, transactions usually require the payment of a *fee*. The total supply of a digital asset and its initial distribution is usually defined within the genesis block. The *genesis block* is the first block of the blockchain, and it specifies parameters like the *block-size*, the list of *participation* nodes, and more.

In a correct blockchain protocol, all the nodes of the network eventually agree on the same view of the blockchain running a *consensus* protocol. The *consensus* determines the way the nodes agree on the next block to append on the blockchain and the frequency, *i.e.*, *block-period*. In some cases, the consensus protocol may cause periods in which not all the nodes have the same view of the blockchain, and we call them *forks*. In the event of a fork, the network needs to synchronise on the same blockchain, and some blocks may be dropped or reorganised; we call this procedure blockchain *reorganisation* or *reorg*. When reorgs are unlikely to happen for a certain block, that block is said *final* or *finalised*. Thus, we refer to the transactions of a *final* block as *finalised* (Finality defined in Section 2.3).

5.2 PETHARD Framework

PETHARD is a framework that allows users to evaluate and compare performance and scalability of the two most prominent consensus protocols adopted in the context of permissionless blockchain systems, namely the PoW and the PPoS. PETHARD adopts a quantitative evaluation technique in which the SUT is subjected to a workload, and its behaviour is monitored afterwards. We refer to such a technique as a PETHARD *experiment*, or *test*, throughout this chapter. An experiment takes as input a set of parameters that allow users to create various conditions under which to evaluate the SUT.

Specifically, a PETHARD experiment requires: (i) the SUT *platform* which implements the blockchain consensus protocol to be tested, (ii) the parameter N which specifies the network size (number of participation nodes), and (iii) the *workload*. PETHARD has been implemented to support two *platforms*, namely Algorand [80] and Ethereum [138]. The former is used to evaluate the PPoS, while the latter is for the PoW.

5.2.1 Methodology

PETHARD implements the quantitative methodology we introduced in Chapter 4. The methodology defines the steps to follow for evaluating a blockchain consensus protocol. PETHARD applies such a methodology to measure performance and scalability of PoW and PPoS. Broadly, a PETHARD experiment works as follows: it initialises a new SUT, either Algorand or Ethereum platforms, thus it generates a *workload* of blockchain transactions. When the SUT terminates the execution of the transactions, a separate PETHARD process to collect and analyse data begins. This process is designed according to the technique introduced in section 4.3.2, that we called *offline-based technique*. Such a technique separates the SUT computation from the benchmarking procedure and avoids overheads that cause measurement unbalances.

Workload. We design the *workload* in PETHARD as a set of blockchain transactions requests that are sent over the SUT. We refer to the workload duration as the period during which the workload is constantly generated, and we call it *load-duration*. Throughout the *load-duration*, requests are equally balanced across the participation nodes, avoiding the distribution toward a single node. Centralised load represent a bottleneck for the requests processing. Thus, the workload is divided into *batches* of fixed dimension, called *batch size*. The *batch size* indicates the number of request in a *batch*. *Batches* are iteratively distributed over the nodes at a constant rate, called *input rate*.

Data collection. PETHARD collects data from the participation nodes who run the consensus protocol and generates a dataset called, *PETHARD Dataset*. This dataset contains the information about transactions and blocks like the timestamp of transactions submission or blocks finalisation. Data collected from the logs must be (i) parsed, to extract only relevant data on transactions and blocks, and (ii) merged, to create a unique dataset that considers all the data measured from the nodes of the network. Hence the *PETHARD Dataset* is characterised by a 6-tuple representing the information of a single transaction, such as $\langle node_{id}, tx_h, tx^s, tx^f, block_n, batch_n \rangle$, where:

- $node_{id}$: is the identifier of a node of the SUT; this value is used to monitor and verify the number of transactions processed by each node;
- tx_h: is the hash of the transaction submitted to the network;
- tx^s: is the transaction submission timestamp;

- tx^f: is the transaction finalisation timestamp, *i.e.*, timestamp when the block containing the transaction t_h is considered final;
- $block_n$: is the number of the final block containing t_h; the *genesis* block has $block_n = 0$;
- $batch_n$: is the numeric identifier of the workload *batch* with which t_h was submitted.

Evaluation Metrics. PETHARD uses the following metrics:

- *Throughput*: measured as the number of transactions finalised by the protocol from t_1 to t_2, it is usually measured in *TPS* (transactions-per-second)

$$TPS = \frac{\#txs(t_1, t_2)}{t_2 - t_1} \tag{5.1}$$

where:
$\#txs$: number of final transactions;
(t_1, t_2): transactions finalisation period;
$t_2 - t_1$: duration (seconds) of the finalisation period.

- *Latency*: measured as the difference between the transaction finalisation time and the transaction submission time, such as for any transaction tx we have:

$$tx_{latency} = tx^f - tx^s \tag{5.2}$$

where:
tx^f: transaction finalisation time;
tx^s: transaction submission time.

- *Scalability*: measured as the variation of throughput and latency altering the number of nodes and the input rate.

Summarising, to design PETHARD we followed the methodology proposed in section 4.3.2 defining the four phases of the *offline-based technique* as follows:

1. *Deploy*: this phase determines the SUT; it receives as input the network size $|N|$, the nodes *type*, the blockchain *platform* and its relative *consensus*, thus prepares the environment and initiates the SUT;
2. *Test*: this phase specifies the workload for stress-testing the SUT; it determines the *input rate* and the *duration* of a workload, thus it generates the load; the workload is balanced across the participation nodes of the SUT to avoid bottlenecks;

3. *Collect*: this phase determines the data collection procedure; it starts once the SUT terminates to process the workload; a process is used to collect the logs from the participation nodes and to store them externally from the SUT;

4. *Measure*: this phase determines the experiment outcome; a parsing process takes as input the collected logs and merges them into a unique dataset, *i.e.*, *PETHRD Dataset*; the dataset is then used to compute performance and scalability metrics.

5.2.2 PETHARD Architecture

PETHARD's architecture is composed of two independent components, namely *Orchestrator* and *Evaluator*, that operate on a single-host server. These components have been designed for deploying the SUT and running the experiments afterwards. The SUT is itself embedded into the PETHARD architecture and supports the deployment of a private network of either Algorand or Ethereum blockchain platforms. Figure 5.1 depicts an overview of the architecture.

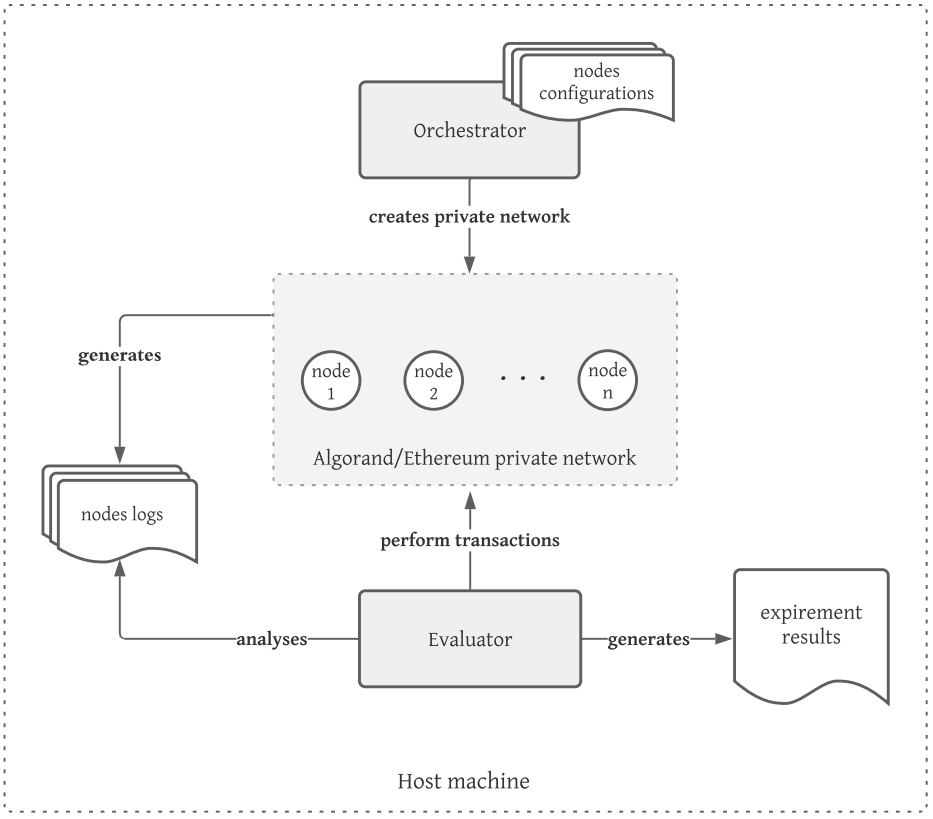


Figure 5.1: *PETHARD architecture. The arrows indicate the interaction between each component, whereas at the centre there is the blockchain private network*

Orchestrator. This component executes the *deploy* phase of PETHARD. It takes as input a configuration file containing the parameters of a SUT. Specifically, given a *platform*

and *consensus*, the number of nodes N, the *type* of nodes, and a set of node-specific configuration options, the Orchestrator produces the SUT environment and deploys the network. Algorand and Ethereum require specific settings in which the nodes must be properly configured and connected to operate properly. The Orchestrator embeds two independent software modules that support the requirements of both Algorand and Ethereum platforms.

Evaluator. This component executes the *test*, *collect*, and *measure* phases of PETHARD. It is responsible for generating the workload, collecting and parsing the logs, computing the *PETHARD Dataset* and the metrics. FIGURE 5.2 shows in detail the components of the Evaluator. There are two sub-components, that we design as independent processes running within the Evaluator itself, namely the *Transactions Generator* and the *Analyser*. The Transaction Generator receives as input a set of parameters that details the workload to be created within the experiment. Thus, it generates the set of transactions requests, namely *TX entities*, needed to accomplish the workload requested. TX entities are nothing more than HTTP requests that invoke the execution of transactions on both Algorand and Ethereum. To comply with the formats and specifications of both platforms, the Transactions Generator must integrate a Software Development Kits (SDKs), and build the transactions requests according to the platform's specifications. Thus it divides the *TX entities* into batches and sends them towards the nodes of the SUT network, load balancing the workload. Afterwards, the Analyser parses and analyses data from the logs, producing the *PETHARD Dataset* and computing the metrics.

5.2.3 Benchmarking Procedure

Symbol	Description
$platform$	Blockchain platform (Algorand/Ethereum)
N	Private network size (number of nodes)
γ	Batch size
τ	Time to wait after submitting a batch
β	Number of batches submitted to each node

TABLE 5.1: *PETHARD experiment parameters*

We define a systematic procedure to execute benchmark experiments of both PoW and PPoS consensus protocols running respectively Ethereum and Algorand private networks. First, we summarise in TABLE 5.1 the input parameters of an experiment. These parameters can be altered to run different experiments varying the consensus parameters. When a new experiment begins, a private network starts from scratch with new parameters (like wallets, accounts, and balances) and a new genesis block.

Summarising, a PETHARD experiment can be interpreted as follows:

The transactions generator submits concurrently γ transactions to every node with a timeout of τ second (γ/τ TPS). This process is repeated β times, amounting to a total of $\gamma * \beta$ transactions processed by the network.

We now introduce the systematic benchmarking procedure of an experiment. The order of the steps must be properly followed as shown, in order to ensure a correct execution. The *input* indicates the parameters or actions required to execute a precise step; the *output* indicates the logs to be observed before moving forward.

1. Start the SUT. Invoke the Orchestrator to initialise a new private network of the blockchain to be tested
   input: $(platform, N)$;
   output: (i) logs showing that all nodes have been created, configured and connected to the network, (ii) consensus logs displayed;

2. Observe the consensus logs of the nodes and wait until the consensus protocol starts its execution across the nodes
   input: n/a;
   output (Algorand): logs showing round's termination and the corresponding votes for its respective block;
   output (Ethereum): logs showing node's mining details and the information of committed blocks received from the other miners;

3. Start the Evaluator and initialise the Transactions Generator and the Analyser components
   input: $(platform, N, \gamma, \tau, \beta)$;
   output: logs showing the nodes' accounts and wallets that are (eventually) signing and processing submitted transactions;

4. Wait until the Transactions Generator terminates the workload and collects the relative blocks
   input: n/a;
   output: log from orchestrator's console asking to stop the network manually;

5. Stop the SUT
   input: `SIGINT` signal to the Orchestrator. (This action triggers the collection of the logs);
   output: logs showing that all nodes have been stopped and the consensus logs aggregated into a single file;

6. the Analyser automatically parses the consensus logs and terminates the experiment
   input: n/a;

output: log confirming that the results of the experiment have been generated and PETHARD has terminated properly.

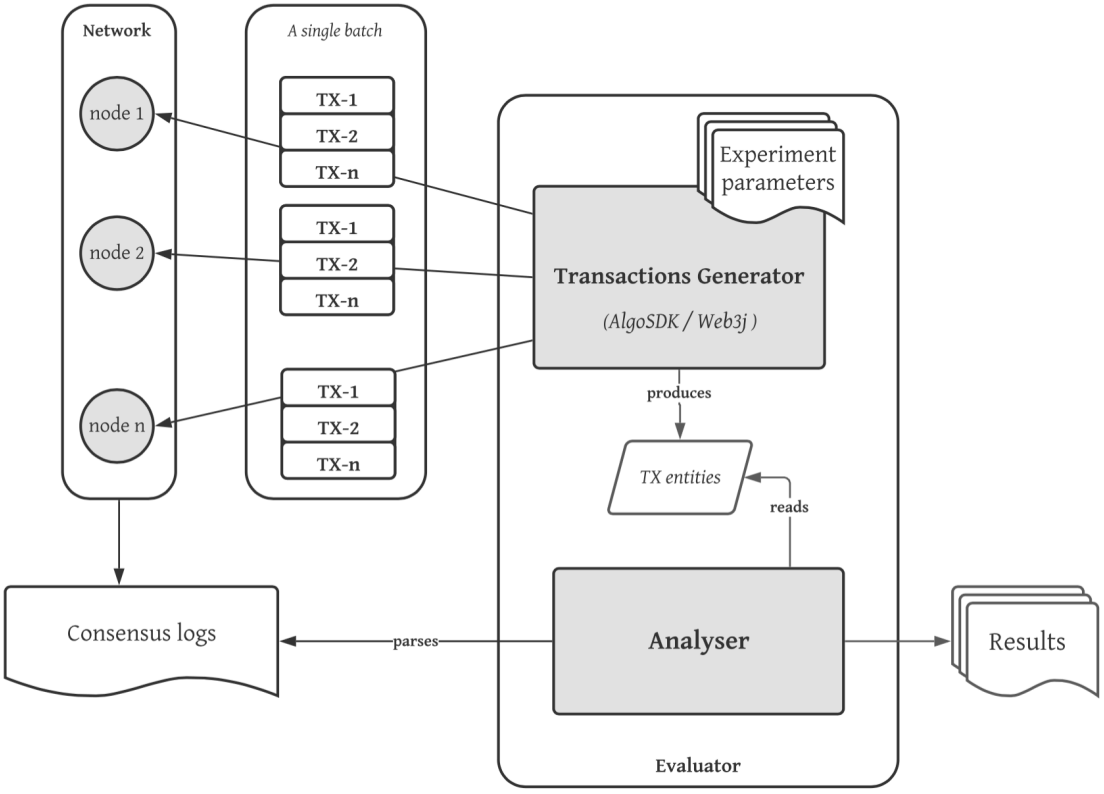


FIGURE 5.2: *A close-up view of the PETHARD evaluator. The transactions generator creates a distributed workload while the analyser parses the logs*

5.3 Implementation

In this section, we describe how PETHARD has been implemented for supporting both Algorand and Ethereum platforms. We discuss the tools and techniques used to build each component of PETHARD and how we implement the interaction with the platforms. PETHARD runs within a virtualised environment deployed over a single machine. Within the same environment we implement three sub-systems, namely the *SUT*, *Orchestrator*, and *Evaluator*. The SUT is deployed using Docker [56], as a container-based virtual network, in which each node of the SUT runs into a single Docker container. Containers are provided with the source code for running either Algorand or Ethereum software. This technique enables the easy deployment of a SUT even in small testbeds and brings great portability and accuracy when carrying out experiments on a local machine. Conversely, the Orchestrator is implemented as a standalone UNIX process via Bash scripting. It is composed of three scripts that implement the functionalities to

configure a start the SUT. Finally, the Evaluator is built as a standalone Java application that creates the methods to initiate and execute both the Transactions Generator and Analyser. Both the Orchestrator and Evaluator interact with the SUT to carry on the experiment phases. This is achieved by implementing ad-hoc software components that interact with the SUT nodes. Specifically, both the blockchain platforms expose a command-line interface (CLI) to interact with the nodes via console, and a set of APIs to access nodes' functionalities programmatically.

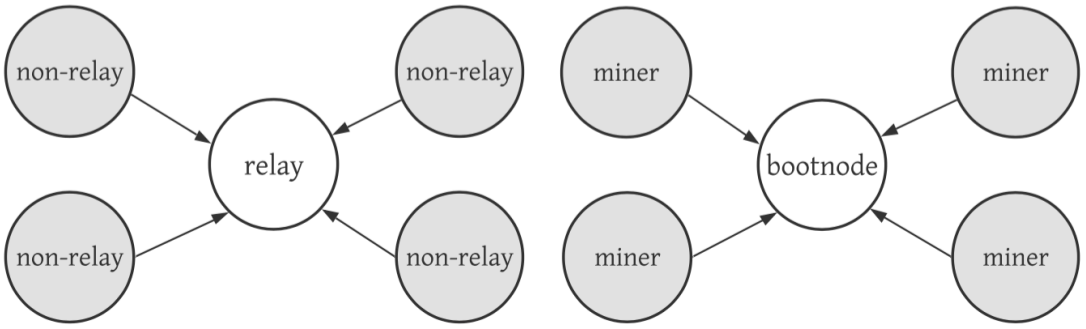
5.3.1 Orchestrator

The Orchestrator executes Bash commands that interact with the Algorand and Ethereum CLIs, respectively `goal` [83] and `geth` [79] interfaces. It is characterised by three different shell scripts which implement the functionalities for creating, starting, and stopping a blockchain network, hence the SUT. The Orchestrator Bash scripts are:

- `createNetwork.sh`: it reads from the Orchestrator configuration file the platform, consensus, network size and type, and the nodes options, thus creates the environment to run the SUT. Specifically, it configures the blockchain nodes, the network topology, and the genesis file to start a new blockchain;
- `startNetwork.sh`: it accesses the environment in which the SUT has been created, thus it starts the network by running the startup command on every node;
- `stopNetwork.sh`: it accesses the environment in which the SUT has been created and therefore stops the network. Afterwards, it stores the logs of the nodes in a repository external to the SUT environment.

Networks Topologies. PETHARD adopts the same network topology for both Algorand and Ethereum SUTs. Given the number N of nodes provided with the configuration file of a new experiment, PETHARD deploys a network with one network node and N participation nodes. With this configuration, the network node is responsible for connecting the participation nodes and managing the communication routings. FIGURE 5.3 depicts an example of such topology for both Algorand and Ethereum. The network nodes are called *relay* and *bootnode* respectively by Algorand and Ethereum, whereas participation nodes are called *non-relay* (or participation) in Algorand, and *miner* in Ethereum. Miner nodes join the PoW consensus and mine new blocks, whereas *non-relay* nodes host stakeholders' accounts that participate in the PPoS consensus.

The `createNetwork.sh` script, therefore, interacts with a Python standalone program that deals with the creation of the SUT network. Specifically, this program creates a YAML configuration file, called *docker-compose.yml*, which represents the containerised Docker network. This file is used to run the SUT through the Docker Compose [58]



(A) Algorand (B) Ethereum

FIGURE 5.3: *Example of Algorand and Ethereum private networks topologies composed by 1 network node and 4 participation nodes. The Ethereum bootnode and the Algorand relay are responsible for communication routing to the participation nodes*

daemon, a Docker tool that offers a set of commands for executing and managing multiple containers at once, reading the YAML file. We configure the Docker Compose to allocate a space of the host's filesystem for every container, *i.e.*, docker *volume*. Hence, the Orchestrator uses *volume*s to store node's configuration files and their logs. Configuration files are a set of files used by a node to set up its settings, while the logs are console outputs containing information on transactions and blocks.

Nodes Configuration. `createNetwork.sh` generates into each container's *volume* the configuration files and invokes the CLI commands to start up the Algorand and Ethereum nodes. The implementation of the script follows the requirements and specifications of the official documentation of both platforms. It includes the methods to generate (i) the configuration data necessary for a node to participate in the consensus protocol, including the *wallet*s, blockchain *account*s, *balance*, and (ii) the blockchain *genesis file*. In particular, the genesis file includes the type of *consensus* protocol, a network identifier called *network id*, the *balance* distribution across the wallets. The genesis file represents the initial state of the blockchain and it is the same for every node.

Algorand Configuration. The Algorand CLI `goal` provides a command to deploy an Algorand private network using a template file. The command is `goal network`., It prepares the configuration files and the genesis file of each Algorand node. This command takes in input a templated JSON file, called *algorand-template.json*, which represents the template of the private network to create, and generates the Algorand nodes configurations and *genesis file* automatically. The template contains the following information:

- *Network name*: a simple string representing the unique network identifier. We used the parameter N to assign the network name, for instance, a network with $N = 7$ would have the name *7nodes-net*;
- *Total supply*: total amount of *Algo*, *i.e.*, Algorand's digital *asset*, circulating within the network;

- *Wallets*: set of wallets and accounts to use at genesis. For each account, it requires the percentage of *Algo*s owned at genesis, and a boolean status indicating the account's registration for PPoS consensus. We create one wallet per node, and one account per wallet. Thus, We assigned to each wallet $\lfloor \frac{100}{N} \rfloor \%$ of the total supply, where N is the number of *non-relay* nodes used for the experiments;

- *Network topology*: the set of *relay* and *non-relay* nodes. For *non-relay* nodes it also requires the wallets owned (defined above). We create a network with one *relay*, and N *non-relay* hosting one wallet each. *Relay* nodes do not host wallets and do not participate in the consensus.

According to the parameters detailed above, the `goal network` creates the required configuration files for each node. Specifically, Algorand is characterised by two main processes, called *algod* and *KMD* (Key Management Daemon). The former is the main Algorand process for handling the blockchain like message exchanges and transactions processing, the latter handles the cryptographic operations for generating wallets and accounts private keys, and for signing transactions. These processes are configured using a JSON file located in the nodes directory. Hence we used the command `goal network` to create them. *Relay* and *non-relay* nodes use two different *algod* configurations, while the *kmd* configuration is only used by *non-relay* nodes. Indeed, Algorand *relay* nodes do not host wallets and therefore the KMD is not needed. Finally, the `startNetwork.sh` script interacts with the Algorand's CLI starting all the *algod* processes of the nodes, and the respective *kmd* processes of the *non-relay* nodes. Similarly, the `stopNetwork.sh` kills those processes and stops the SUT.

Ethereum Configuration. Differently from Algorand, Ethereum's CLI `geth` does not provide a unique command to create a private network. Therefore given a network size N, Ethereum's `createNetwork.sh` handles all the operations to create the deployment environment, such as creating the nodes configurations files, wallets, accounts, and genesis file. For each node, it creates one account through the `geth account new` command. Finally, the private key of the account is stored in a file called *keystore*. To generate the genesis file, the `createNetwork.sh` invokes the Ethereum's `puppeth` command that is used to create and customise a genesis file manually. Ethereum nodes have various parameters that can be tuned according to specific needs. It has been proved that *gasLimit* and *difficulty* parameters mostly impact performance [116, 170]. `puppeth` provides default values for both params, hence we changed those values as follows:

- `gasLimit`: it determines the *block-size* in Ethereum. Hence, to obtain realistic measurements, we reflect the current *gasLimit* of the Ethereum's MainNet network; the default value of `puppeth` was modified from 524,288 to 12,500,000; we used the Ethereum's MainNet average gas limit at the time of writing [68];

- `difficulty`: it determines the *block-period* in Ethereum's PoW. It defines the difficulty of the PoW puzzle that the miners must solve to create a new block; we fixed the default `difficulty` generated by `puppeth` to reflect Algorand's block period average of 4-5 seconds.

Differently from Algorand, we configure the Ethereum nodes using the *geth* CLI options. Specifically, `createNetwork.sh` starts both the *bootnode* and *miners* running the `geth` command. In addition, to start the *miners* we used the options listed in TABLE 5.2 and TABLE 5.3. TABLE 5.2 shows the commands used to specify the *miner* type and its fundamental parameters such as (i) the number of threads used by the node to process transactions, (ii) the transactions fees, called *gasPrice*, (iii) the *gasLimit*, and (iv) the *target* which must be equal to *gasLimit* to maintain a constant *block-size*. TABLE 5.3 shows the commands used to specify the name of the network defined in the genesis file, the location to be used for storing the logs and to retrieve the *keystore*, and the network address of the *bootnode*.

Option	Value	Description
`--mine`	–	Mining enabled
`--miner.threads`	1	Number of CPU threads to use for mining
`--miner.gasprice`	0	Minimum gas price for mining a transaction
`--miner.gaslimit`	12500000	Maximum gas ceiling for mined blocks
`--miner.target`	12500000	Target gas for mined blocks

TABLE 5.2: `geth` *options used to deploy a PoW miner node in Ethereum*

Option	Description
`--networkid`	Identifier of the private network
`--datadir`	Data directory for the databases and keystore
`--bootnode`	Network identifier of the bootnode

TABLE 5.3: `geth` *options used to connect a miner to the bootnode and expose the port required by the Transactions Generator*

5.3.2 Evaluator

We implement the Evaluator as a standalone Java application implementing both the Transactions Generator and Analyser components. To interact with both the Algorand and Ethereum blockchains, the Evaluator imports the Algorand and Ethereum Java SDKs, respectively *Java AlgoSDK* [3] and *web3j* [66].

Transactions Generator. The Transactions Generator creates the workload specified within an experiment. The workload is splitted into *batche*s of Algorand and Ethereum

transactions requests. Batches are then stored as lists of *TX entities*, and therefore balanced on each node of the SUT. We implement the workload such as all the transactions are validated and signed programmatically through the SDKs. With this approach, we delegate the cryptographic operations needed for creating valid transactions to the Java process. In this way, the blockchain nodes only have to execute the transaction without engaging with computation consuming cryptographic operations to sign transactions.

The Algorand implementation is characterised by the Java classes `AlgodClient.java` and `AlgorandNode.java`. The former provides the functionalities to connect and communicate with the Algorand nodes. It accesses the nodes using the IP addresses and ports assigned to the containers by Docker. The latter imports the *Java AlgoSDK*to create Algorand TX entities, and to sign them with the account's private keys. To implement the signature functionalities we built an additional class, called `KmdApi.java`, which imports the wallets private keys used to sign transactions through the `POST /v1/key/export` method exposed by the Algorand APIs. In this way, we achieved the signature of transactions programmatically on the `AlgorandNode.java` class. In presence of high loads, concurrent transactions need to be distinguished to avoid transactions rejection. For this reason, we use the optional `note` field of the Algorand transaction. Hence, the `AlgorandNode.java` makes use of the `UUID.randomUUID()` method from the `java.util` package to create the unique transactions with custom `note` fields.

Similarly, we implement the Ethereum Transactions Generator. A Java class called `EthClient.java` connects to the nodes through the containers' IP addresses and network ports and implements the functionalities to communicate with the Ethereum APIs. Thus, a class called `EthereumNode.java` imports the the *web3j* SDK for constructing Ethereum *TX entities*, and the node's *keystore* for collecting the accounts' private keys. Private keys are then used to sign transactions. Therefore, for each *TX entity*, the Transactions Generator starts a Java `Threads` and subsequently each thread issues all transactions concurrently. For each *TX entity* the Java thread starts a timeout that waits for the nodes' acknowledges. However, in the case of large workloads, transactions are queued and the nodes' replies got delayed. Hence, to avoid the Java threads being killed before submitting all transactions, we delayed the timeout using the `CountDownLatch` from the `java.util.concurrent` package. This method enables the synchronisation of all the *TX entities* waiting for all transactions to be submitted. Similarly to Algorand, in Ethereum concurrent transactions need to be distinguished to avoid errors. The `EthereumNode.java` creates unique transactions fixing the `nonce` optional field of the Ethereum transactions. The `nonce` is used in Ethereum to numerate the transactions issued by a certain account and update its balance accordingly [194]. Hence, the `EthereumNode.java` creates unique `nonce`s using the parameter *TX entities* list size γ. Specifically, for each transaction, it computes the `nonce` as $nonce = \gamma + cNonce$, where $cNonce$ indicates the value of the previous `nonce`.

The process of workload generation used with the Transaction Generator is summarised with the Algorithm 1. The parameters and the functions used in the algorithm are described below:

- $N := \{n_1, n_2, \ldots, n_n\}$ is the finite set of participation nodes;
- $W := \{w_1, w_2, \ldots, w_n\}$ is the finite set of *wallets* addresses;
- $\Omega : N \twoheadrightarrow W$, is a one-to-one function that accepts a node and returns its wallet; given a node n_1, $\Omega(n_1)$ returns the wallet of n_1, that is w_1;
- β represents the number of workload *batch*es;
- γ represents the size of a workload *batch*, *i.e.*, the number of *TX entities* in a *batch*;
- $\Pi := \{\pi_1, \pi_2, \ldots, \pi_i\}$ is a finite set of transactions ready to be submitted - *TX entities* (a single batch). The size of Π depends on γ and the size of the network ($|N|$), thus $|\Pi| = |N| \times \gamma$.
- τ defines *batch*es issuance rate.
- $R_n := \{r_1, r_2, \ldots, r_\gamma\}$ defines the set of *receivers* (wallets) for a given node $n, n \in N$; for a node n_1 and its wallet address w_1 derived from $\Omega(n_1)$, then $R_n = \{W - \{w_1\}\}$;
- $\Delta : (N \times W \times R) \rightarrow \Pi$, is a function executed by each node to create a *TX entity*; it takes as input a 3-tuple $\langle n_1, \Omega(n_1), r_1 \rangle$ (a node, a sender and a receiver) and it returns a *TX entity* π_1;
- $TX := \{tx_1, tx_2, \ldots, tx_n\}$: it represents a list of transactions receipt in which tx_i corresponds to the transaction identifier on the blockchain *TX entities*;
- $\Phi : \Pi \rightarrow TX$, is a function executed by the nodes to process submitted *TX entities*; it takes as input a *TX entity* and returns the receipt of processed transaction *tx*.

Analyser. The Analyser is implemented as a single Java process responsible for collecting the logs of the blockchain nodes and afterwards processing them to generate the *PETHARD Dataset*. Specifically, once the Transactions Generator terminates, it retrieves the list of *TX entities* and the list of transactions receipts TX which identify the transactions processed but not yet executed. Therefore, it starts a new process with the method `waitTransactionsToBeProcessed()`, that waits the execution of every transactions in TX. This method checks the nodes' *mempools* and terminates when all the *mempolls* are empty, *i.e.*, when all the submitted transactions have been executed. When `waitTransactionsToBeProcessed()` terminates, the Analyser iterates over the list of transactions receipts TX and. For each tx_h, the Analyser collects from the blockchain

Algorithm 1 Pseudocode of Transactions Generator Workload

Require: N, Ω, R_n, Δ, β, τ, γ
Ensure: TX, the set of all transactions processed by network.

$TX \leftarrow \emptyset$
for β batches **do**
  $\Pi \leftarrow \emptyset$
  for all $node \in N$ **do**
    for γ iterations **do**
      $sender \leftarrow \Omega(node)$
      $receiver \leftarrow SelectRandomReceiver(R_{node})$ //selects any r from R_{node}
      $TXentity_\gamma \leftarrow \Delta$ (node, sender, receiver)
      $\Pi \leftarrow \Pi \cup \{TXentity_\gamma\}$
    end for
  end for
  for all $TXentity \in \Pi$ **do in parallel**
    $tx \leftarrow \Phi(TXentity)$
    $TX \leftarrow TX \cup \{tx\}$
  end for
  $timeout(\tau)$
end for

$node_{id}$	tx_h	t^s	$block_n$	$batch_n$
1	0x54a5aa1862...	1618940343328	23	1
2	0xc6c4a43dd3...	1618940345193	21	1
...	...	...	...	...

TABLE 5.4: *Ethereum transactions receipt dataset*

the values for $node_{id}$, tx_h, t^s, $block_n$, and the batch number $batch_n$. Hence, it creates a dataset as shown in TABLE 5.4. We call this dataset `transactionsResults.csv`.

Afterwards, the Analyser collects the finalisation time t^f of transactions parsing the nodes' logs. The log parser collects the blocks finalisation timestamps and generates a 2-tuple dataset called `blockResults.csv`, such as $< block, t^c >$. TABLE 5.5 shows an example of the 2-tuple dataset.

In Ethereum, the logs contain information about the PoW consensus like the nodes mining phases. Hence, we implement the log parser so that it stores the timestamp when the logs show the finalisation of a block, which is represented with the log string:

```
INFO [04-04|10:41:46.601] Block reached canonical chain number=517 ...
```

Multiple nodes of the network generate this log for a specific block number, and it informs a developer that a particular block can be considered part of the main chain of the entire network, *i.e.*, the block is finalised, and it cannot be reverted. The most recent log time is considered to be the log confirming a specific block.

$block_n$	finalisation time
1	1618939903046
2	1618939903263
...	...

TABLE 5.5: *Ethereum blocks confirmation dataset*

On the other hand, Algorand's PPoS proceeds in rounds and for each round, one block is proposed. PPoS also provides instant finality, hence blocks are finalised when the round terminates. In PPoS, the round is concluded when the leader, selected amongst a committee, certifies a block and broadcasts it to the network [80]. We parsed the logs identifying the rounds termination times, *i.e.*, when the entry `Type` is equal to `RoundConcluded`. The log parser collects N logs and therefore delects the log with the most recent timestamp as the most reliable finalisation time of a block. Finally, we consider the round number the same as the block number as the protocol generates one block per round [80]. The block numbers and their t^f are then stored in a single dataset.

At the end of the parsing phase, the Analyser merges both datasets into the *PETHARD Dataset*. This operation is achieved using a simple Python script with the library `pandas3` [147]; given the `transactionsResults.csv` and `blockResults.csv` datasets, it runs an inner join operation on the $block_n$ column and produces the final dataset. An example of the *PETHARD Dataset* is shown in TABLE 5.6.

$node_{id}$	tx_h	tx^s	$block_n$	$batch_n$	tx^f
1	0x54a5aa1862...	1618940343328	23	1	1618939903046
2	0xc6c4a43dd3...	1618940345193	21	1	1618245653328
...	...	...	...	...	...

TABLE 5.6: *PETHARD dataset*

5.4 Experimental Evaluation

5.4.1 Environment and Deployment

The environment used to deploy and evaluate PETHARD was composed of a PowerEdge R730xd rack server with 56 logical processors Intel(R) Xeon(R) CPU E5-2695 v3 2.30GHz running the VMware ESXi hypervisor. We ran one virtual machine with *68-Core Intel Core i7 2.6 GHz* with *132 GB 2667 MHz DDR4 RAM* and 1T storage. We deploy the SUT running $N + 1$ containers (N participation nodes plus 1 bootnode)

with 2 CPUs and 4GB RAM each. The SUT ran with Docker *v18.09.7* and Docker Compose v1.24.0. We used *Algorand 2.8.0 stable* and the Ethereum *Geth v1.10.6*.

5.4.2 PETHARD Evaluation

PoW Difficulty Configuration

We used PETHARD to tune the `difficulty` parameter of the Ethereum PoW. The main purpose was to make Ethereum's experiments consistent with Algorand's block period. We ran an experiment for both platforms deploying private network of $N = 5$ nodes, with a workload of $\beta = 20$ *batches* at the *input rate* of 40 tx/s γ/τ. Hence, we used the *PETHARD Dataset* to compute the average block *block-period* (BP) with the eq. (5.3), such that:

$$BP_b = finalised_b - finalised_{b-1} \qquad (ms) \tag{5.3}$$

where:
b: The block number
$finalised_b$: finalisation timestamp of block b.

The first block is equal for every node and it is computed straightforwardly from the genesis file, hence $BP_0 = 0s$. To calculate the average BP of a blockchain, we compute $\overline{BP}$ with eq. (5.4), such that:

$$\overline{BP} = \frac{\sum_{n=1}^{N} BP_n}{N} \qquad (ms) \tag{5.4}$$

where:
N: quantity of all blocks generated during an experiment.

In Algorand, the $\overline{BP}$ is $\approx 4.5s$ according to the official MainNet metrics[16]. This value is fixed and depends on how PPoS achieves consensus; there is not a parameter in Algorand to change it. Conversely, in Ethereum's PoW, the BP value depends on the `difficulty` params set at genesis, as we already discussed in section 5.3. We tune the PoW's `difficulty` such that $BP_{PoW} \approx BP_{PPoS}$. The default difficulty set by the Ethereum's command `puppeth` is `0x80000`, *i.e.*, `524288` in decimal format. FIGURE 5.4 shows that the experiment measured for Algorand a $\overline{BP} = 4.24s$. Then, we ran the same experiment for Ethereum. We first tested the platform using the default `difficulty` value and we obtained $\overline{BP} = 2s$. The default `difficulty`, used as a starting point, was then increased by 50, 100 and 125% to find the closest $\overline{BP}$ to Algorand. The chart shows that increasing the default value by 100% gives us a $\overline{BP} = 4.08s$.

[16]Algorand MainNet metric dashboard: `https://metrics.algorand.org`.

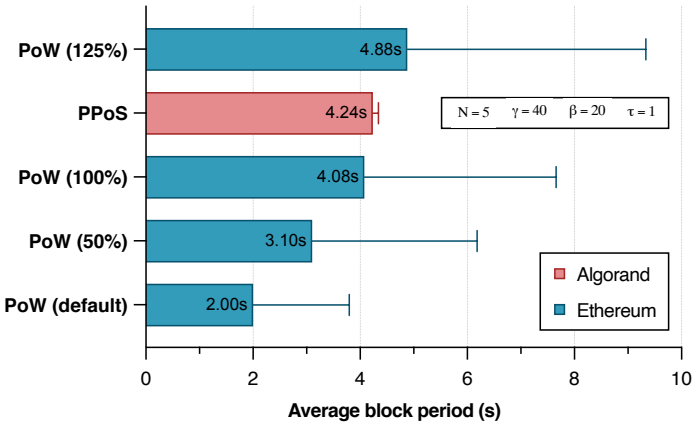


FIGURE 5.4: *Average block period of Algorand's PPoS and Ethereum's PoW when submitting 100 ($N \times \gamma$) transactions for 40 (β) times every second (τ). The default difficulty of PoW, defined by puppeth, is increased by 50, 100 and 125%*

FIGURE 5.4 also reveals to us that $\overline{BP}$'s PoW can be variable and not stable as for Algorand, reaching a maximum of 7.6s. This is because the PoW forks are common, and if they occur, the process to confirm a block is delayed, whereas, in PPoS, the probability of forking is 10^{-18} [80]. To sum up, the difficulty chosen for the PoW problem is (`1048576`), which was found by increasing the `puppeth`'s default value by 100%. This difficulty will allow us to have a similar $\overline{BP}$ to Algorand during the experiments where the protocols are compared.

Performance: PoW versus PPoS

We compared PPoS and PoW performance measuring their *throughput* and *latency* metrics over time, with a fixed input rate and network size. As shown in TABLE 5.7, we deployed a network of 5 nodes and we tested it with a workload of 500 tx/s for 2 minutes.

Parameter	Value
N	5
γ	100
τ	1
β	120

TABLE 5.7: *Experiment setting for performance evaluation.*

Throughput Evaluation. The chart in FIGURE 5.5 illustrates the throughputs of both PPoS and PoW protocols over the experiment time, and their respective averages. FIGURE 5.6 shows the first 200 seconds of the experiment. We measured the throughput by counting the number of transactions finalised in a block b and its BP_b. To allow

better visualisation of the performance over the experiment time, the y-axis represents the average throughput every 5 seconds. PPoS achieves a constant throughput equal to the input rate, whereas PoW shows a sinusoidal pattern that after the 225^{th} second reaches random peaks. This result provides evidence that PPoS generates blocks at a constant rate, whereas PoW blocks generation is variable due to mining. To further understand this throughput rate, we measured the average block period and the number of transactions per block. Figure 5.7(a) shows that PPoS maintains its block period stable, finalising an average of 1859.5 transactions per block, which means that the input rate is immediately processed and finalised.

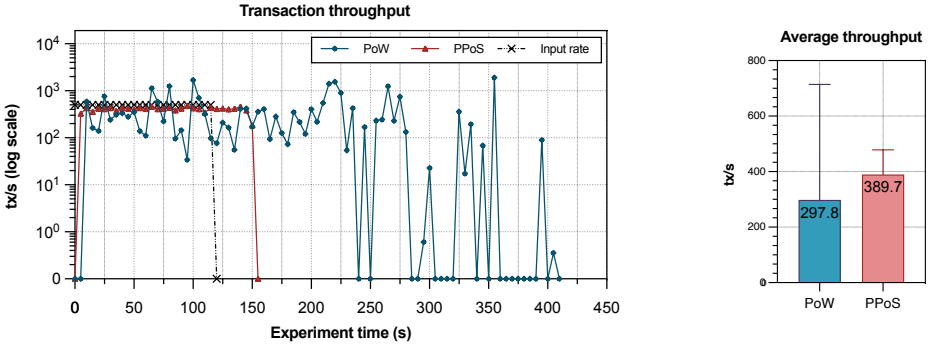


FIGURE 5.5: *PPoS and PoW TPS over time, and average throughput*

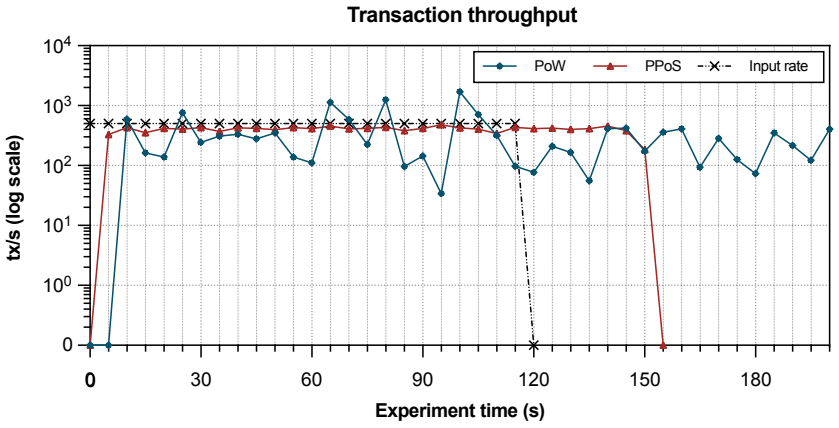


FIGURE 5.6: *PPoS and PoW TPS 200s experiment*

On the other hand, PoW's blocks were generated with different block periods ranging from less than 1 second up to 18 seconds resulting in even higher peak throughput than PPoS due to low block periods. This suggests that blocks were generated by multiple miners, which increased the probability of forking. Looking at the number of transactions in Figure 5.7(b), PoW produced empty blocks (with no transactions) between block 75 and block 150, delaying the entire experiment time. This provides evidence that a fork occurred, and miners struggled to synchronise on the longest chain using the GHOST protocol [175, 194]. To prove this claim, we examined the logs of the nodes to find out

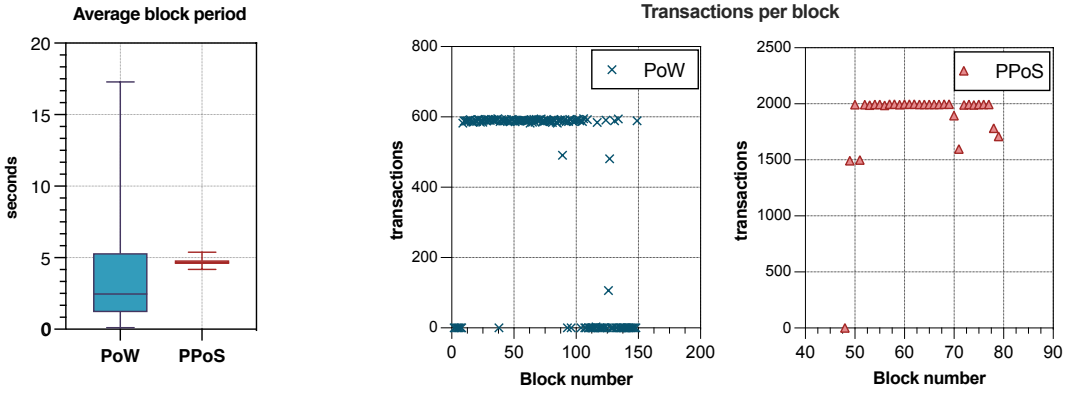


FIGURE 5.7: *Comparison of PPoS and PoW average block period and number of transactions per block*

whether some blocks got refused, *i.e.*, *uncle blocks* [194]. We found that five blocks were classified as *uncle blocks* during the experiment, which means the transactions contained inside were reverted to the transaction pool and as FIGURE 5.5 shows. These transactions were finalised over the last 100 seconds of the experiment.

Latency Evaluation. We measured the average transaction latency per batch. FIGURE 5.8(a) illustrates the latency of 120 batches submitted sequentially with a rate of 500 tx/s. In PoW, the latency linearly increased as the batches were delivered, reaching an average latency of 137s, while PPoS presents a constant behaviour with an average transaction latency of 7.2s. Hence, a constant input rate does not impact PPoS throughput and latency, whereas the PoW suffers from unstable throughput that leads to a linear increase of latency over time.

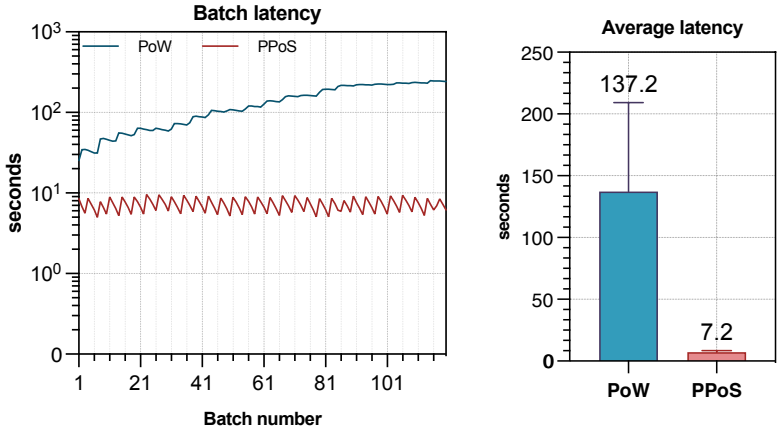


FIGURE 5.8: *Comparison of PPoS and PoW transactions latency and averages*

Scalability: PoW versus PPoS

Input Rate Variation. The following experiment shows how the variation of input rate may affect scalability, hence the average latencies and throughputs, in both protocols. TABLE 5.8 summarises the experiments we conducted.

Experiment name	N	γ	β	τ
8 nodes - 100tx/s	8	13	60	1
8 nodes - 200tx/s	8	25	60	1
8 nodes - 400tx/s	8	50	60	1
8 nodes - 600tx/s	8	75	60	1
8 nodes - 800tx/s	8	100	60	1

TABLE 5.8: *Experiments setups to measure PPoS and PoW scalability varying input rate and fixed network size*

For each experiment, we estimate the overall throughput by considering the number of transactions confirmed during the experiment time, that is, the period from the first transaction submitted to the last transactions finalised by the network. The transaction latency is measured by taking the average of all transaction latencies. FIGURE 5.9 illustrates the transaction latency and transaction throughput versus the input rate ranging from 100tx/s to 800tx/s. The graph in Figure 5.9(a) shows that the average transaction latency in PPoS always stays under 10s no matter how large the input rate is. By contrast, increasing the input rate from 400 tx/s to 600tx/s doubled PoW's latency from around 50s to just over 110s; however, when increased the input rate from 600tx/s to 800tx/s, PoW's high latency declined steadily, whereas PPoS's latency slightly increased from 6.5s to 9.8s. In comparison, however, PoW's latency is about 10× higher when delivering 800 tx/s.

Looking at the transaction throughput in FIGURE 5.9(b), PPoS achieves the best performance, improving its throughput linearly while maintaining the same latency. Conversely, in PoW, the throughput slightly increased but always remained under 200 TPS for all input rates, whereas PPoS achieved a maximum throughput of 500 TPS with an input rate of 800 tx/s. The low throughput of PoW is due to the `gasLimit` that we assigned to Ethereum nodes and genesis file; with our configuration, each block can theoretically have at most 595 [17] transactions per block. When considering the overall latency and throughput of the experiment, the performance is capped due to the fixed block size set by the `gasLimit`, *i.e.*, we expect higher latencies, and constant throughputs even in case of higher input rates. Indeed, we observe in FIGURE 5.9(a) a sharply increases in transactions latency when delivering 600 tx/s, which means that the network must generate

[17]The number of transactions was found by dividing the `gasLimit` value set in our configuration by the standard Ethereum transaction's gas value: 12500000/21000 = 595.2 section 4.6.2.

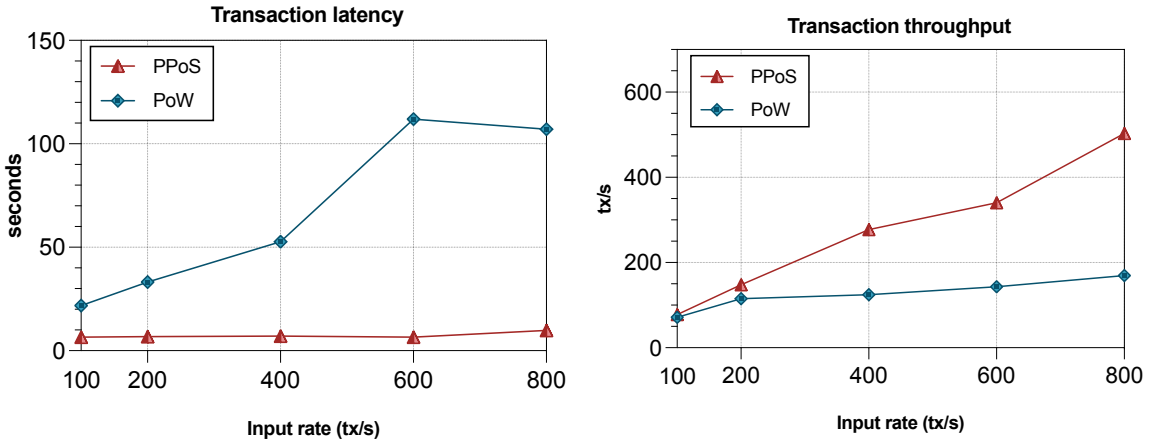
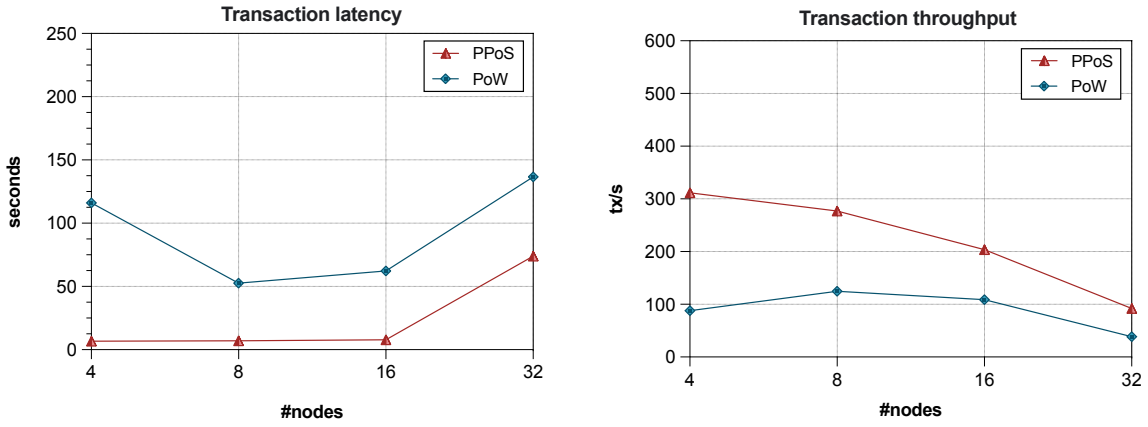


FIGURE 5.9: *Comparison of PPoS and PoW average throughputs and latencies with input rates ranging from 100tx/s to 800tx/s, and a network size of 8 nodes*

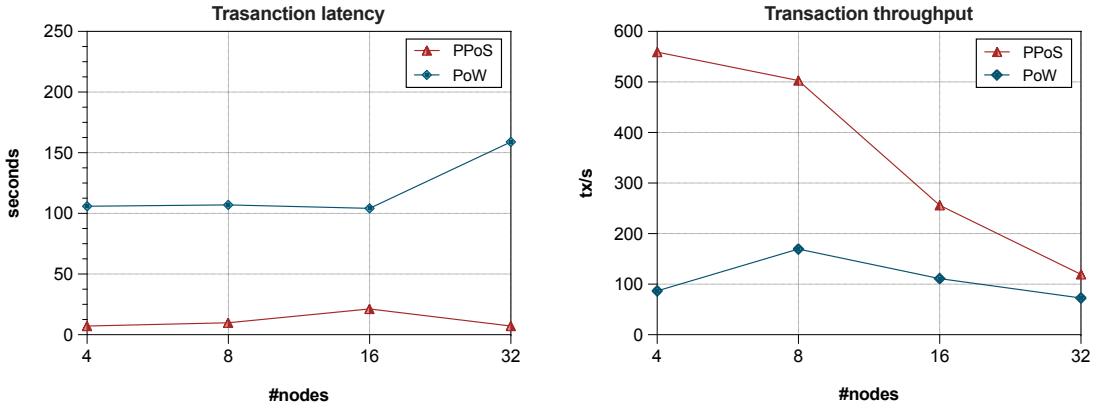
two blocks to fit 600 transactions, instead of just one; therefore, this also explains the latency 2× higher drifting the input rates from 400 tx/s, to 600 tx/s and 800 tx/s.

Input Rate and Network Size Variation. To analyse the performance scalability of the protocols, we varied both the network size and input rate. FIGURE 5.10 illustrates a comparison of the protocols under two different input rates, such as 400 tx/s and 800 tx/s. We ran four experiments varying the number of nodes from 4 up to 32 per input rate. Overall, the two graphs (FIGURE 5.10a-(b) and FIGURE 5.10b-(b)) show that the horizontal scaling of the system lowered the throughput of both protocols, reaching roughly the same value moving from 16 to 32 nodes. This was caused by the type of network topology used, in which the network node, *i.e.*, *relay* for Algorand and *bootnode* for Ethereum, represented a bottleneck of the blockchain network delaying the propagation of blocks and transactions to all nodes. We leave as future work the evaluation of the same experiment in a more complex topology without centralised network nodes. Looking closely at FIGURE 5.10a-(a), the transaction latency of both protocols sharply increases when raising the number of nodes from 16 to 32. FIGURE 5.10b-(a), however, shows that with a higher input rate, only PoW's latency increased, whereas PPoS's latency remained under 10s. This behaviour was caused by the fact that in PoW more miners generate more blocks simultaneously hence the probability of forks increases. As a result, the PoW had to run more iterations of the GHOST protocol to synchronise the blockchain, bringing additional delays to transactions finalisation. Differently, FIGURE 5.10a-(a) shows that switching from 16 to 32 nodes, with an input rate of 400tx/s, caused in PPoS a drastic increase of latency. This result was caused by the relay node failing to verify messages. Specifically, these issues occurred during the *Block Proposal* phase where nodes broadcast the blocks using the *Gossip protocol* [80], and at the same time, they wait for other blocks. In this phase, only one block is selected. However, in



(a) (b)

(A) Input rate 400 tx/s



(a) (b)

(B) Input rate 800 tx/s

FIGURE 5.10: *Comparison of PPoS and PoW average throughputs and latencies with input rates of 400 tx/s and 800 tx/s, and network sizes of 4, 8, 16, and 32 nodes*

the case of *relay* overloading, some messages may fail and some rounds skipped with zero transactions finalised - no block finalised.

5.5 Discussion

In this chapter, we presented PETHARD, a benchmarking framework for measuring performance and scalability of Proof-of-Work (PoW) and Pure Proof-of-Stake (PPoS) protocols. PETHARD can be used to run experiments with fixed workloads and measure metrics like transactions throughput, latency, and scalability. Thus, we tested PETHARD running an experimental evaluation of the protocols. First, we demonstrate how the PoW difficulty impacts performance, thus we provide a tuning procedure in which we fixed the parameter according to the average Algorand's block period. Then, we tested performance and scalability of both protocols running experiments with the same

configuration. The results have shown that the PPoS protocol can solve the scalability issues of PoW-based protocols. In particular, we showed that varying the input rate, the transaction throughput of PPoS linearly increases, and the transaction latency always remains constant. On the other hand, even though PoW achieves a slight increase in throughput, this comes at the cost of higher latencies. Moreover, by increasing the number of nodes, PoW experienced an increment of latency and a linear decrease of throughput, whereas PPoS kept a constant latency and a decrease in throughput. Summarising, PETHARD enabled us to demonstrate that PPoS outperforms PoW both in terms of performance and scalability. However, PETHARD only enables testing experiments in LAN environments. We believe that, a realistic comparison should consider a WAN network deployed over the Internet. Moreover, the experiments highlighted the following limits of PETHARD: (i) the network topology used with a centralised network node might cause bottlenecks that unbalance the measurements, (ii) the configuration parameters of a blockchain also affect performance regardless of its consensus protocol. We leave as future work the extension of PETHARD that overcomes those limits.

Chapter 6

From PETHARD to PERSECUS: A Dependability Benchmark for Blockchains Systems and Consensus Protocols

In the previous chapter, we introduced a framework for benchmarking two consensus protocols used in permissionless blockchains, namely PoW [138] and PPoS [80]. The proposed framework enabled us to evaluate performance and scalability of both protocols outlining their different behaviours. However, the results showed that consensus is not the only component affecting performance measurements. Indeed, we observed that systems' misconfigurations may create hidden bottlenecks that impact systems' performance regardless of the consensus protocol. For instance, the experiments showed that Ethereum block sizes drastically impacted the throughput of PoW beyond its effective performance. In addition, the framework only worked for private testbeds where nodes communicate through a LAN environment, free from delays and network faults. However, blockchains are usually deployed over untrusted environments like the Internet, hence to evaluate them under realistic scenarios there is a need of a more comprehensive benchmarking framework.

Although in recent years some effort has been devoted to proposing benchmarking frameworks to measure performance of blockchains, most of these solutions tailor pre-existing tools designed for traditional distributed systems, like YCSB [46] or TPC-W [129], to specific blockchain use cases [88]. Besides them, nowadays only few works propose customised benchmarking tools designed specifically for blockchain systems [55, 73]. Even though these benchmarks are useful tools to assess performance, the number of supported blockchains is limited and it results usually difficult to extend them to other systems. Performance studies are nowadays limited to the few platforms available with

these tools. Moreover, the solutions proposed so far follow ad-hoc benchmarking procedures and do not rely on systematic standards. Consequently, different tools may lead to divergent results that cause confusion in the common knowledge between researchers and scientists. To overcome this issue, blockchain benchmarks should follow structured and standardised approaches, which can be reproducible and extensible to any system.

Furthermore, all the benchmarking solutions emerged to date focus on performance, with the aim to fulfil a primary market need: compare blockchains against classical transactional and payment systems. However, the blockchain was born to offer a secure alternative to centralised systems for securing interactions between untrusted parties without intermediaries. In this context, security and dependability are paramount properties besides performance. A blockchain system that cannot guarantee them, it is not a blockchain system at all. Thus, nowadays benchmarks must shift the focus from measuring performance to the measurement of both performance, security and dependability. To this extent, *dependability benchmarks* [5, 186] represent a standardised specification. These particular benchmarks define a set of rules following performance benchmarking techniques and common dependability assessment procedures, to measure and compare performance, security, and dependability of computer systems.

In this chapter, we present *PERSECUS*, a comprehensive dependability benchmark to evaluate *PERformance and SECurity of blockchain systems and consensUS protocols*. PERSECUS adapts the standard procedures defined by traditional dependability benchmarks, to modern blockchain systems. Specifically, it aims at resolving the limitations of state-of-the-art benchmarks, proposing a novel benchmarking system based on standards. Specifically, PERSECUS identifies the following standard requirements that a blockchain dependability benchmark must have:

1. *Optimised experimental setup*: Most of the blockchain benchmarks run the experiments within test environments in which the system under test is configured with default parameters that might not be optimal, and the nodes communicate over LAN networks that do not represent a realistic environment; PERSECUS determines an optimised experimental setup in which (i) blockchain parameters can be tuned to avoid misconfigurations, and (ii) the system under test is deployed under a realistic environment simulating a large WAN network;

2. *Benchmark procedure and rules*: definition of a systematic benchmarking procedure and rules that must be followed to run the experiments;

3. *Faultload*: a set of faults and adverse conditions that simulate realistic faults that can affect a blockchain, like network partitions or malicious behaviour of nodes;

4. *Efficient workloads generation*: For the sake of simplicity, current benchmarks generate workloads using a centralised approach where: (i) a single server machine acts as workload generator, (ii) the workload generator runs on the same server

of the blockchain, (iii) the workload is not balanced over the blockchain. It has been proved that such a centralised approach creates overheads that may affect the performance of the blockchain, and also produces unwanted bottlenecks [172]. PERSECUS introduces an alternative approach to workload generation, in which the load is produced externally from the blockchain by a set of distributed clients, and load-balanced over the entire network;

5. *Efficient metrics computation*: Most of the benchmarks adopt HTTP runtime procedures to collect data from the blockchain overloading the system with unwanted computation overheads that affect the measurements [199]. PERSECUS embeds an efficient method to metrics computation using a log-based approach that does not stress the blockchain with computing and networking runtime data collection services.

PERSECUS is also intended to be a *flexible* and *extensible* tool, that facilitates the experimenting of various blockchain systems varying configurations parameters, network conditions, and workloads according to specific needs. Furthermore, it is built with a modular architecture that allows the integration of any blockchain platform through the implementation of adapters. PERSECUS also ensures *scalability*, enabling benchmarking tests that can scale up to different blockchain networks and workloads.

We evaluate PERSECUS comparing three Ethereum blockchains running different consensus protocols, namely *Parity* [151] with *AuRa* [148], and GoQuorum [43] with Clique [156] and IBFT [117]. PERSECUS enabled us to identify three components of these blockchains that caused performance bottlenecks and security flaws beyond the efficiency of different consensus protocols. Specifically, we observed that (i) a misconfiguration of parameters directly impacts performance and security of the system, and (ii) the serialisation component of a blockchain may turn into a vulnerability.

Contributions. The contributions of this chapter can be summarised as folllows:

- we define four fundamental characteristics that a blockchain benchmark must have to fulfil accurate measurements, and limit bottlenecks and computations overheads that can generate misleading results;
- we provide a systematic benchmarking procedure and rules that characterise the experiment of a blockchain benchmark;
- we introduce PERSECUS, a comprehensive dependability benchmark to evaluate performance and security of blockchain systems and consensus protocols under various configurations; PERSECUS monitors the behaviour of a blockchain and simulates realistic network conditions in which communications are delayed, and

the system is subject to faults. It generates two datasets, namely (i) the *PERSECUS Dataset*, which contains data collected for metrics computation like transactions throughput and latency, (ii) *Forks Dataset*, which contains the number of forks detected during the experiment. In addition PERSECUS provides a *Security Report* which contains a security analysis of the blockchain;

- we provide an evaluation of PERSECUS through a comparative study of three blockchain systems. First, we tune the configuration parameters of the blockchains running various tests using different combinations of parameters. Then, we evaluate and compare performance and security of those platforms under their optimum configuration. Finally, we evaluate scalability varying the number of nodes of the blockchains and the workloads.

Chapter Structure. Section 6.1 discusses the related works and how PERSECUS differs, while Section 6.2 defines the system model and introduces the novel concept of blockchain system under test. Section 6.3 presents PERSECUS detailing the benchmark procedure and the architecture, thus Section 6.4 introduces an implementation of PERSECUS with two Ethereum clients. The experimental evaluation is detailed and discussed in Section 6.5, whereas Section 6.6 outlines the results and summarises the chapter contributions.

6.1 Related Works

The benchmarking of complex computer systems is a well established research field. There exists several works defining standard procedures and tools to evaluate and compare performance of different systems [47, 48]. However, with the advent of blockchains, most of the existing frameworks became unpractical [185]. Recently, several alternatives to classic benchmarks arose with the aim of assessing blockchain systems.

Blockbench. Blockbench [55] is a benchmarking framework to evaluate performance of private blockchains. It monitors runtime statistics of a blockchain under test, and measures metrics like throughput, latency, and scalability with various workloads. To date, it provides workloads ranging from smart contract use-cases, to traditional database-oriented loads, such as YCSB [46] and Smallbank [84]. Despite numerous workloads, Blockbench cannot achieve efficient workload generation due to a centralised design. Differently, PERSECUS enables the deployment of high efficient workloads employing distributed clients. Blockbench provides metrics for fault tolerance and security. However, it can only simulate crash faults and network partitions. Currently, Blockbench only supports default deployment configurations for Hyperledger Fabric [96], and two Ethereum platforms, namely Parity [151] and Geth [82].

Caliper. Hyperledger Caliper [73] is a blockchain benchmarking framework, which allows performance tests of different blockchain platforms with custom use-cases. It provides metrics for transactions throughput and latency, thus transactions success rate and resource consumptions like CPU and Memory usage. Metrics are processed at runtime using an HTTP method, using the same workers that generate the workloads. A recent study from Baliga et al. [14] demonstrated that Caliper's metrics collection creates overheads that affect measurements. The authors propose a patch for Caliper, introducing a new metrics collection componente. Caliper's workloads support custom smart contract applications and custom loads configurations like the number of clients and the load rate, however, it can only be generated from a single, multi-threaded, cluster. Caliper supports Hyperledger Fabric [96], Hyperledger Besu (Ethereum) [95], and Geth (Ethereum) [82], but it is extendible to any Ethereum-based platform. However, it does not provide the possibility of tuning configuration parameters. Differently, PERSECUS enables custom blockchain configurations defining parameters like the consensus, block size, block period, and more. Caliper relies on container virtualisation for the deployment and testing of blockchain platforms. However, it lacks network emulation functionalities, considering only LAN deployments.

Chainhammer. Chainhammer [105] is a benchmarking framework for testing performance of Ethereum-based platforms. Currently, it supports the following platforms and consensus protocols: Parity-AuRa [148], Geth-Clique [156], and GoQuorum with Raft [42] and IBFT [117]. Although Chainhammer enables testing with various network sizes and input rates, it only provides throughput metrics, transactions per second, and do not measure scalability. Moreover, Chainhammer's blockchain deployment is based on container virtualisation starting a containerised network on a single host. However, it has also been used to benchmark a remote Microsoft Azure blockchain network. Chainhammerx implements workloads through Ethereum's `web3` Python SDK [67], as single batches of transactions generated from one client, and forwarded to one node of the blockchain under test.

BTCMark. BTCMark [164] is a framework to assess different blockchains through various application scenarios and different infrastructures. It provides a general framework to deploy in a single cluster the blockchain under test, the monitoring system, and the load generator without depending on any particular environment. BTCMark can simulate different network conditions via a network emulator, and create random workloads. Thus, it provides metrics to evaluate performance and resources consumptions using runtime HTTP APIs. Currently, BTCMark supports Geth (Ethereum) [82] and Hyperledger Fabric [96]. It has been used to evaluate performance and resources consumptions of these platforms with custom workloads in two independent testbeds.

Table 6.1 outlines five aspects that we identified to be crucial requirements for a blockchain benchmark, and we show how PERSECUS fulfils them against state-of-the-art benchmarking tools presented above.

	Optimised setup	Faultload	Efficient workload generation	Efficient metrics computation
***BLOCKBENCH** [55]*	No	Partial (crash, partitions)	No (centralised design)	No (computation overheads)
***Caliper** [73]*	No	No	No (multi-thread, single host)	No (computation overheads)
***Chainhammer** [105]*	No	No	No (centralised, not balanced)	No (computation overheads)
***BTCMark** [164]*	Partial (real network, no params tuning)	Partial (partitions)	No (single host, random loads)	No (computation overheads)
PERSECUS	Yes (real network, params tuning)	Yes (partitions, crash, Byzantine)	Yes (multi-thread, balanced)	Yes offline log-based

TABLE 6.1: *Comparison of PERSECUS properties against state-of-the-art benchmarks*

6.2 System Model

We define a *benchmark* as a set of m parallel processes $\Sigma = \{c_1, .., c_m\}$, stress-testing a target system, called *System Under Test* (SUT) [157]. A process is also called *client*, and we indicate with j the *index* of the client c_j. *Parallel* means that the processes are independent and run simultaneously. We refer to a *stress-test* as a period of fixed duration, *i.e.*, *test-duration*, characterised by two phases; in the first phase the clients load the SUT with a controlled amount of requests, *i.e.*, *workload*, in the second phase they observe the SUT behaviour accordingly. *Controlled* workload means that the clients can specify the duration of a load phase, *i.e.*, *load-duration*, the type of requests, *i.e.*, *req_type*, and the frequency, *i.e.*, *req_rate* or *input rate*. During the load phase, the SUT starts processing the workload according to the *input rate* and eventually returns a response to the clients. *Eventually* means that some requests might fail due to communication failures or SUT unavailability. A SUT is any system provided with *hardware*, *software* and *connectivity* components [157]. We model the SUT as a private blockchain running a distributed computing protocol across a structured *peer-to-peer* (P2P) network [50], and we call it *Blockchain System Under Test* (BSUT). *Structured* means that the nodes of the network are connected with a predefined topology. Broadly, a BSUT is composed by a set of servers, *i.e.*, *BSUT hardware component*, that communicate over a P2P network, *i.e.*, *BSUT Network Component*, and execute the same distributed protocol, *i.e.*, *BSUT Software Component*.

BSUT Hardware Component. The BSUT is composed by a fixed set of n servers (either physical or virtual) $\Pi = \{p_1, .., p_n\}$; we indicate with i the *index* of the server p_i. These servers, called *nodes* or *peers*, are provided with connectivity, storage, and computing capabilities.

BSUT Network Component. The BSUT network is a P2P decentralised network of interconnected nodes spread across different geographic locations. Connection links between nodes can be faulty and messages delayed, lost or duplicated. In the absence of faults, a delay $\Theta = (delay, jitter, correlation, distribution)$ affects communications

due to node distances. *Delay* represents the delay in milliseconds, *jitter* the random delay variation in milliseconds, *correlation* the delay correlation in percentage, and *distribution* the delay distribution in terms of probability distribution. Additionally, we assume the BSUT network to be *partially synchronous* [60], in which communications are asynchronous, messages delayed, and network partition may occur. In this model, communications eventually become synchronous and messages delivered within some unknown finite time-bound Δ.

Trust Model. We refer to the threat and attacker models presented in Section 4.2.2. In this model we refer to unexpected communication error like message delays or loss as *timing faults*. Therefore, we assume an upper bound f of nodes in Π to be faulty. A node is said *faulty* if it does not operates as expected by the protocol. A faulty node is unreachable due to a crash, *i.e.*, *crash fault*, or because subverted by an adversary, *i.e.*, *Byzantine fault*. Non-faulty nodes are said *correct* or *honest*.

BSUT Software Component. We refer to the BSUT software component as a distributed computing protocol, *i.e.*, the *blockchain core*, executed among the nodes. Such protocol includes the fundamental components of *(i) blockchain data structure*, *(ii) transactions scheduler*, and *(iii) consensus protocol*. To describe the interactions with such components, we refine the nodes in Π as: *network nodes*, *participation nodes*, and *validator nodes*. The network nodes deal with the communication routings and maintain a list of *nodes addresses*, *i.e.*, unique network identifiers, the participation nodes deal with the protocol computation tasks and execute runtime operations, and the validator nodes run the consensus protocol and update the blockchain data structure accordingly.

Blockchain Data Structure. We define a *blockchain* as a distributed ledger fully replicated over a P2P network. The nodes collectively maintain and update the blockchain according to the execution of runtime operations. The ledger data structure consists of a list of records, called *blocks*, cryptographically linked together. Blocks are of fixed size, *i.e.*, *block-size*, and include information such as the cryptographic hash of its predecessor block, a timestamp, *i.e.*, *block-timestamp*, and a list of *transactions*. A transaction represents an operation executed on the blockchain. The blockchain data structure is of type read-only/write-once, such that the blocks cannot be removed once appended on the blockchain. The first block of the blockchain is called *genesis block* and specifies important parameters like the block-size, the list of validator nodes, and more. For each block b, appended on the blockchain, the distance between b and the genesis block is called *height*. For a certain height h and a block b, we define the tuple *(h,b)* and the blockchain *state*. In a correct blockchain core protocol, all the nodes of the network eventually agree on the same state; there may be periods in which nodes have inconsistent views of the state, and we call them *forks*.

Transactions Scheduler. Clients interact with the blockchain core protocol invoking the execution of transactions via cryptographic identities, called *accounts*, stored in a participation node. Blockchain accounts are asymmetric key pairs $< Pk, Prk >$; the public key of a blockchain account represents a unique identifier on the blockchain, and we call it *address*. Nodes organise accounts into cryptographically secured collections, *i.e.*, *wallets*. Associated with a blockchain account, the protocol stores also specific data, like a *balance* of digital assets. We define a *digital asset* as a fungible token governing the blockchain core protocol. Clients can own and exchange digital assets, or use them to execute transactions, *i.e.*, transactions usually require the payment of a *fee*. The total supply of a digital asset and its initial distribution is usually defined within the genesis block. Afterwards, we define the transaction scheduler as the software component dealing with transactions. A *schedule* describes the execution of a transaction issued to the system [18]. The transaction scheduler of the BSUT comprises the execution of several schedules, namely the *transactions acceptance schedule (TxAS)*, *transactions serialisation schedule (TxSS)*, and *transactions pooling schedule (TxPoolS)*. The TxAS includes a runtime API that exposes the APIs to interact with the protocol, and a queue to store pending requests. Pending requests represent unprocessed transactions of the BSUT, referred to as *submitted*. The TxSS interprets those pending requests, and we call this procedure *serialisation*. Requests can be serialised either sequentially or concurrently to optimise performance [18, 193]; with concurrent serialisation, the schedule starts several instances, called *txss-thread*. Concurrent schedules must guarantee *serialisability*, *i.e.*, a property that guarantees the equivalence in the outcome of concurrent and sequential schedules [18]. At the time of serialisation, the TxSS checks transactions signatures. Similarly to blockchain core protocols like Bitcoin and Ethereum [138, 194], we define a transaction as *signed* if it contains a digital signature of the issuer account entitled to pay the transaction fee. In the case of an unsigned transaction, the TxSS computes the digital signature with the private key of the issuer; we define a transaction as *rejected* if the computation of its signature fails. After serialisation, the TxPoolS validates serialised transactions and stores them in a data structure also referred to as *mempool*. The validation is achieved by a deterministic function γ, such that for each transaction tx, $\gamma(tx) \rightarrow \{True, False\}$[18]. Even though γ can be implemented with a wide range of validation checks, for the sake of simplicity, and without loss of generality, we assume a transaction tx to be valid if the following checks succeed:

- the transaction includes a correct and unique index, *i.e.*, a unique number also called *nonce*;
- the account that issued the transaction has enough digital assets in its balance to pay the associated fees;
- the transaction includes a valid signature of the issuer.

[18]Validity was firstly introduced by Cachin et al. with the notion of *external validity* in [33].

We refer to valid transactions stored on the mempool as *accepted*.

Blockchain Consensus Protocol. The blockchain core acts as a secure and trustworthy protocol executed across untrusted nodes. To ensure the correct execution, this protocol must ensure BFT. BFT ensures that a set of nodes agree on a common state (or output), despite the presence disruptive events like Byzantine faults [76]. We assume a consensus protocol that tolerates up to f Byzantine nodes. Such BFT consensus periodically elects validators as block proposers, also called *leaders*. Leaders create blocks at a certain rate, called *block-period*. For each *block-period*, the leader creates a new block with some transactions collected from the mempoool, then proposes such block to the network. We refer to a transaction recently added to a new block as *confirmed*. Afterwards, validators agree on such a new block and propagate it to the rest of the network. Thus, every node appends the block to the blockchain and remove the confirmed transactions from the mempool accordingly.

In some cases, a block may take longer to be accepted by the network due to the possibility of blockchain forks. In the event of a fork, the network needs to synchronise on the same blockchain, called *canonical*, and some blocks may be dropped or reorganised; we call this procedure blockchain *reorganisation* or *reorg*. When a block joins the canonical chain it cannot be altered or removed, hence it is considered *final* or *finalised*. *Finality* is an essential feature in blockchain, and it represents the termination of consensus on blocks and transactions. A blockchain consensus protocol ensures finality when the following properties are meet:

> *Persistency.* If an honest node appends a block b to its local blockchain at a certain height h, such that $state = (b, h)$, then b will be eventually appended at heigh h of any other honest node despite the possibility of forks;
>
> *Termination.* If a valid transaction tx is accepted by a node, eventually tx will be inserted into a block b and successively b appended to the blockchain of every honest node, and finalised.

6.3 PERSECUS

6.3.1 Benchmarking Procedure

PERSECUS builds on the following idea: propose a *benchmark* for assessing complex blockchain systems through stress-testing experiments. PERSECUS enhances measurements accuracy mitigating unwanted bottlenecks and overheads that afflicted recent benchmark studies. To accomplish this goal, we define with four requirements:

1. *Optimised BSUT setup.* There exist several configuration parameters of a BSUT that can be tuned for performance optimisation, such as *consensus* protocol, *block-period*, *block-size* and more. A misconfiguration of those parameters may cause bottlenecks and security flaws that can obfuscate the real behaviour of a BSUT [14, 180]. PERSECUS fosters the deployment of optimised BSUT providing configurable settings which can be tested with various workloads and operating conditions;

2. *Faultloads.* Most of the existing benchmarks evaluate the BSUT performance in a fault-free operating environment assuming robust networks and the absence of faults. However, in a real deployment, the BSUT is intended to operate over the Internet where communications may fail, messages may be delayed or lost, and nodes may crash or act maliciously. PERSECUS allows the design and implementation of custom experiments considering both fault-free and adverse deployments;

3. *Efficient workloads generation.* The workload generation is crucial for a benchmark. Recent benchmarking studies undertake centralised workloads produced directly from the BSUT, and propagated toward one or few nodes of the BSUT itself [105, 107, 126]. This approach hides several drawbacks [172]: (i) workload capped by the server resources; (ii) workload produced within the BSUT overloads the systems of unwanted computation and creates overheads; (iii) sending requests to specific nodes of the BSUT generates bottlenecks in the BSUT requests processing. PERSECUS overcomes with these limitations fostering balanced workloads generation, where users connect from distributed external servers and equally balance the load between all the participation nodes of the BSUT;

4. *Efficient metrics computation.* The way a benchmark collects and processes data is crucial. Most of the benchmarks collect data using the so called RPC-method, by interacting with the blockchain peers via API endpoints exposing blockchain data to the external. However, this method has been proved to be inefficient, introducing computation overheads within the system [199]. PESERCUS uses a log-based data collection method which collects data from the logs of the BSUT peers at the end of each experiment and analyses such data externally. Such a log-based method has several advantages: (i) it does not introduces overheads, (ii) it produces more detailed results since the logs contain more information than the API endpoints, (iii) it scales to any type of BSUT regardless the APIs.

To accomplish the above requirements we designed a general benchmarking procedure that can be applied to any blockchain systems. Such a procedure defines systematically the fundamental steps and tasks to execute during a benchmarking *experiment*. FIGURE 6.1 depicts the procedure workflow and its phases. The first phase is called *deploy*, and it defines and executes the BSUT. In this phase we configure and deploy the target system, defining its network topology, configuration parameters, consensus protocol, and initial state (e.g. genesis block, accounts, etc.). To identify the optimum configuration,

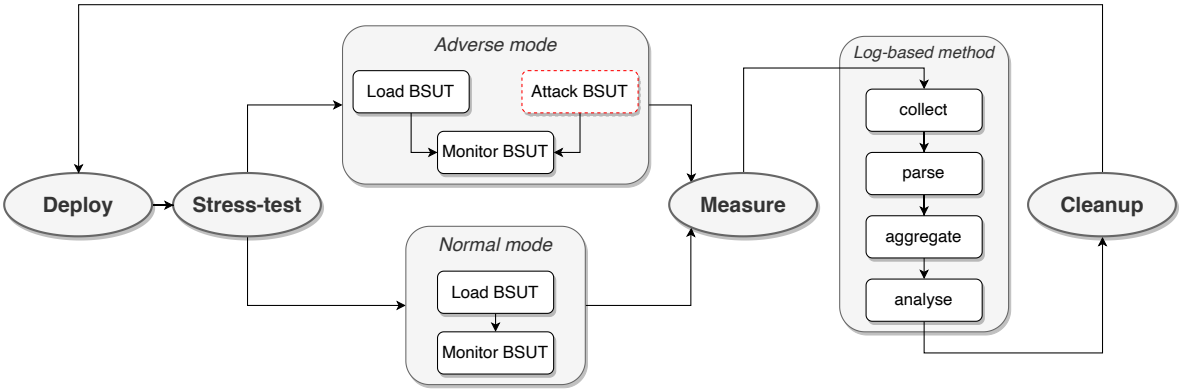


FIGURE 6.1: *Benchmarking Procedure Workflow. The circles indicate the phases of a benchmarking experiment whereas the boxes describe phase specific sub-tasks.*

we run the experiment varying the BSUT settings. After the deploy phase, the BSUT initiates the blockchain core protocol.

The second phase is the *stress-test*, in which we define the set of BSUT clients and the workload. In this phase we model two execution modes, namely the *normal mode* and *adverse mode*. The former reproduces a BSUT running under a fault-free environment, while the latter considers faultloads like timing faults and Byzantine faults. For both modes, we split the stress-test in two sub-task: (i) *load BSUT*, and (ii) monitor BSUT. The load task generates the workload, while the monitor task observes the BSUT reaction and traces the workload processing; a third sub-task is defined within the adverse mode, *i.e.*, the *(iii) attack BSUT* task, which enables reproducing adverse conditions in the BSUT. Although different workloads can be defined, it is crucial to comply with the efficient workload generation. Hence we define the following workload design constraints:

1. the workload must be generated externally from the BSUT;
2. the clients must be distributed across different machines;
3. the workload must be equally distributed across the participation nodes of a BSUT.

A stress-test terminates once the BSUT has processed all the transactions of the workload (transactions finalised or rejected), then the *measure* phase begins afterwards. Within the measure phase we define a log-based method for collecting and processing significant data from the BSUT. This method extracts data directly from the logs produced by the blockchain core protocol of any validator node of the network. To avoid overheads, data needs to be processed externally from the BSUT. The proposed method undertakes four sub-tasks, *i.e.*, *collect*, *parse*, *aggregate*, and *analyse*. The collect task retrieves the logs from the nodes; the parse task extracts from the logs of each node relevant data on transactions and blocks; the aggregate task merges the results into a unique dataset;

the analyse task processes the dataset and computes the metrics. The last phase of the approach is the *cleanup*, which quits the BSUT and clears any configuration setup. After the cleanup, a new experiment can be executed.

6.3.2 PERSECUS Architecture

PERSECUS has been designed to be an *extensible* and *flexible* benchmarking tool able to support any type of blockchain platform through a modular architecture, enabling experiments with different BSUT configurations. Besides extensibility and flexibility, PERSECUS is intended to ensure *scalability*, providing distributed workloads that can scale up together with the BSUT network, so that to produce adequate and efficient loads for stress-testing experiments under various network dimensions.

PERSECUS' architecture is characterised by an orchestration of different systems controlled by a unique interface. Broadly, users access the interface to initialise and start a new experiment. The initialisation consists in configuring the target BSUT, the workload, and the execution mode - there are two execution modes in PERSECUS *normal mode* and *adverse mode*. At the end of the experiment, PERSECUS generates two datasets: *PERSECUS Dataset*, which contains relevant data on transactions and blocks, and *Forks Dataset*, which includes information on blockchain forks. Additionally, PERSECUS generates two reports, namely *PERSECUS report* and *Security report*. The former details the experiment configuration including the network topology, blockchain params, faults, Byzantine nodes, and workload; the latter summarises the BSUT resiliency measured via forks analysis of the blockchain. FIGURE 6.2 illustrates the architecture of PERSECUS, which is composed by three sub-systems: (i) Management System, including the interface used to interact with PERSECUS and run new experiments, (ii) Operating System, containing the BSUT deployment and data collection components, and the monitoring and adverse mode agents, (iii) BSUT representing the target system.

Blockchain System Under Test (BSUT). The BSUT is the target system of a benchmarking experiment. It is composed by a set of independent nodes running the software of a blockchain core protocol. Nodes store the logs produced by the protocol into a local database and expose an API to retrieve such logs externally and to access blockchain-related information. Therefore, PERSECUS uses such API to acquire the logs for further analysis (measure phase).

PERSECUS Operating System. The Operating Systems includes a number of components running independent processes that interact with the BSUT to accomplish with the experiment tasks. The first component is the *Blockchain Deployment Manager* (BDM), which executes phases 1 and 4 of the experiment. It is intended to be extendible via custom *adapters* supporting different BSUT types. Although the BDM can have an adapter for almost any blockchain, to ensure maximum extensibility and flexibility, the

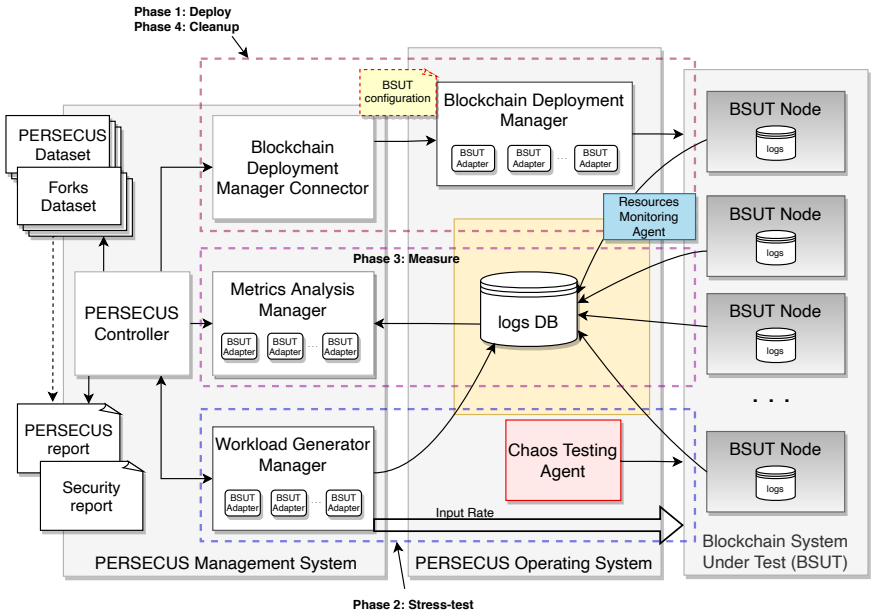


FIGURE 6.2: *PERSECUS Architecture. The dotted lines delimitate the four phases of a benchmark experiment, whereas the area with a yellow background represents the storage component shared among each phase. The Chaos Testing Agent, depicted with red background, is only used for experiment running adverse mode of a stress-test.*

entire BDM component can be replaced with customised versions. Custom BDMs allow PERSECUS to be compatible with any kind of blockchain systems, coping with every type of deployment structure. Broadly, a BDM accepts as input a *configuration* file which specifies the target BSUT and therefore invokes its relative adapter to start the system. The configuration defines a set of parameters such as the *platform*, *network*, *genesis file*, *scheduler*, and *consensus*. *Platform* identifies a specific blockchain platform, hence the BDM adapter to be used; *network* details the number of BSUT nodes and their type; *genesis file* specifies the initial state of a BSUT, detailing the total supply of the *digital asset*, the initial *balance* of each *address*, and the *block-size*; *scheduler* details the *mempool* capacity and the number of *txss-threads* (**if** *txss-threads* == *1* **then** *scheduler* = *sequential*); *consensus* details the type of consensus, and the *block-period* parameter. The BDM component exposes the following methods:

- `computeNodeParams()` – Executes the cryptographic operations required to compute the node addresses of each node of the BSUT, and also generates the asymmetric keys for the accounts of any participation node;
- `configNetworkTopology()` – Configures the network topology, hence generates the of node addresses for the network nodes of the BSUT;
- `configNode()` – Specifies the configuration of a BSUT node. Although adapters may have pre-configured parameters due to compulsory configuration requirements

of BSUTs[19], this method enables the configuration of mempool size and txss-threads which are recurrent params on different BSUTs;

- `buildGenesis()` – Creates the genesis file of the blockchain according to the structure required by the platform under test. Broadly, genesis files determine the protocol configuration at the initial state of the blockchain, *i.e.*, the total supply of assets and distribution over the blockchain accounts, the block-size, the consensus protocol, and the block-period;
- `start()` – Start the BSUT with the configured environment;
- `stop()` – Stop the BSUT.

Besides the BDM, PERSECUS Operating System comprises of two additional software agents, namely (i) the Resource Monitoring Agent, and (ii) the Chaos Testing Agent, and therefore a *logs DB* where data is stored. The *Resource Monitoring Agent* firstly monitors the consumption of CPU and Memory of each BSUT node, and then stores the observed results into the logs DB. The *Chaos Testing Agent* only activates with the adverse mode, and it implements the attack sub-task of the stress-test phase. This agent combines chaos engineering techniques, *i.e.*, methods that simulate real-world adverse scenarios, to disrupt the BSUT with timing, crash, or Byzantine faults.

PERSECUS Management System. The Management System consists of a set of interfaces called *managers* that enable the configuration and initialisation of a new benchmark experiment. The *Blockchain Deployment Manager Connector* (BDM Connecto) is an interface that connects with the BDM component; it specifies the set of required parameters to generate a BDM configuration. Such a connector is independent and can refer to a different BDM. It fosters the integration with custom BDMs. PERSECUS modularity allows benchmarking experiments without requiring users to undertake complex deployment procedures. The BDM connector is composed of the following methods:

- `configureBDM()` – Generates the configuration file required by the BDM to prepare the setup environment of the BSUT;
- `startBDM()` – Initialises the BSUT by starting the BDM's `start()` method;
- `clearBDM()` – Clears the configuration environment and stops the BSUT through the BDM's `stop()` method;

While the BDM connector implements PERSECUS' phase 1, phase 2 is handled by the *Workload Generator Manager* (WGM). The WGM is the interface to the workload generator, and it specifies the type of workload and the number of clients to initiate.

[19]Adapters developers must refer the official documentation of the target blockchain platform for guidelines on node pre-configuration requirements.

The WGM is extendible through a set of adapters that implement different types and formats of requests accepted by the BSUTs. To create a workload, the WGM needs a generator, which can be either an external tool installed into the WGM itself or an ad-hoc software script. However, besides the type of generator, the workload must be designed according to the guidelines presented in section 6.3.1 and integrated with a WGM adapter implementing the following methods:

- `configureWorkload()` – Configures the workload defining the number of clients, and the parameters *req_type* and *req_rate*; it also determines the duration of a stress-test specifying both the *test-duration* and *load-duration* parameters.
- `startWorkload()` – Initiates the *input rate* direct from the clients toward the BSUT nodes. It terminates when the *test-duration* expires.

The workload is then stored into the logs DB with the tuples *(timestamp, input rate)*. Afterwards, PERSECUS starts the *Metrics Analysis Manager* (MAM) which initialises phase 3. The MAM encompasses BSUT adapters to collect the logs of the nodes and parse them into a standardised data structure. Although the logs from different BSUTs may be of various types and structures, the MAM defines common attributes that can be usually found in the logs of any blockchain system. Specifically, the MAM parses transactions attributes considering the following events: *submission*, *confirmation*, and *finalisation* of a transaction. Hence, the MAM aggregates those data and generates a unique dataset, *i.e.*, the PERSECUS dataset, which indicates the following tuple: $(tx, b, b^n, ts^s, ts^c, ts^f)$, where tx is the transaction identifier, b is the identifier of the block containing tx, b^n is the block number[20], ts^s is the timestamp of tx submission, ts^c is the timestamp of tx confirmation, and ts^f is the timestamp of tx finalisation. Together with the PRESECUS dataset, the aggregation task also generates the Forks dataset. This dataset counts the number blockchain forks detected through the experiment by evaluating at runtime the states of the BSUT nodes. A Fork dataset is defined by the tuple $(timestamp, \#forks)$. The MAM adapters implement the following methods:

- `collectLogs()` – Retrieves the logs of each validator node from the logs DB. This method must integrate a function observing the state of transactions and the mempools and afterwards collect the logs if and only if all the submitted transactions have been processed;
- `generateDataset()` – Takes as input the logs and executes the parsing and aggregation procedures to generate the PERSECUS and Forks datasets;
- `computeSecurityEvaluation()` – Generates the Security Report which shows an analysis of the state of the BSUT nodes and an evaluation of forks detected during the experiment.

[20]The block number also represents the blockchain height.

Finally, *PERSECUS Controller* represents the software component that orchestrates the BDM connector, WGM, and MAM through each experiment phase. This component executes the methods of each manager and then produces a final report, *i.e.*, *PERSECU report*, which summarises (i) the BSUT platform, configuration parameters, consensus, and network topology, (ii) the workload duration, type, frequency, and clients, (iii) the type of faults (if adverse mode) and timestamps.

6.4 Implementation

In this section, we describe how we implemented PERSECUS and how we integrated it with two blockchain platforms that implement the Ethereum core protocol [194] namely Parity [151], and GoQuorum [43]. Parity is built in Rust programming language and it aims at being the more efficient and secure Ethereum platform; GoQuorum is designed for enterprises, and it builds on top of the original Ethereum implementation written in Golang, *i.e.*, Geth [82].

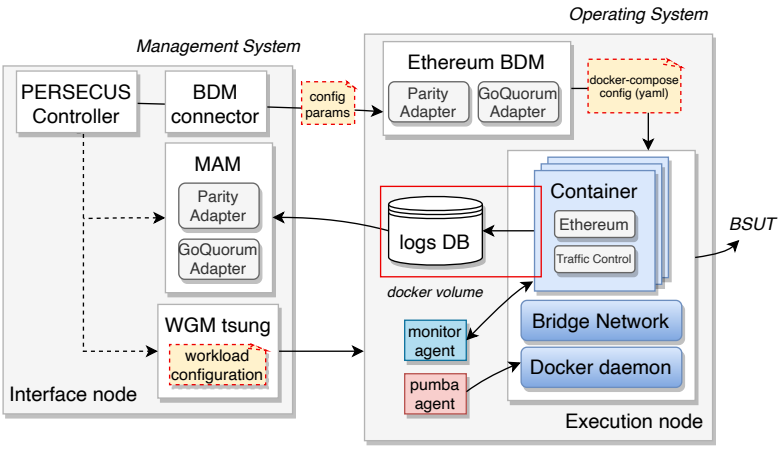


Figure 6.3: *PERSECUS implementation with two Ethereum clients.*

Figure 6.3 illustrates the implementation of PERSECUS, which is composed by two independent nodes called *interface* and *execution* node. The former represents the interface that is exposed to the users. It determines the functionalities to set up a new experiment, and it hosts the Management System. The latter executes the processes and tasks of a benchmark, and it hosts the Operating System and the BSUT. The execution node takes advantage of container virtualisation and network emulation techniques to run a BSUT and simulate a real network into a single environment. It must be provisioned with an adequate amount of resources that must be partitioned through the containers of the virtualised network according to the user requirements. Container virtualisation makes PERSECUS a *flexible* and *portable* tool: (i) it avoids large testbeds which are expensive and hard to manage, (ii) it fosters the execution of complex blockchain networks within a single machine facilitating the execution of benchmarking

activities under several configurations, and (iii) it can be easily deployed in a different environment. Network emulation allows us to reproduce realistic deployment conditions simulating network delays, partitions, and faults, within the virtualised environment. In the following sections, we describe the implementation of PERSECUS' sub-systems.

Blockchain System Under Test (BSUT). We implement the BSUT as a virtual network deployed through container virtualisation with *Docker* [56]. The BSUT hardware component is built on top of the Docker daemon, *i.e.*, a service to deploy containerised applications, that initiates a network of n isolated containers that represent the nodes of the BSUT. Thus, containers receive connectivity, computation, and storage capabilities. Computation is given by partitioning the CPUs and Memory of the execution host among the containers. Storage is provided through the so-called *Docker volume*, a portion of the execution node filesystem dedicated to Docker containers to persist data into the *logs DB*. Containers run as isolated processes that execute the blockchain core software through a docker image, *i.e.*, a special file used to start programs in a Docker container. Images include application code, libraries, tools, dependencies and other files needed to make a BSUT software component run. We built two custom docker images, respectively one for Parity [8] and one for GoQuorum [9]. Both images include a Linux-based distribution embedded with the Ethereum software (Parity or GoQuorum) and the Linux utility called *traffic control* (TC), used to configure the kernel packet scheduler in Linux. Containers communicate through the BSUT network component, which we implement with the Docker *bridge network*. We configure it with a fixed *subnet* and *IP address range*, and for each container, we assign a static IP within that range. Although communications on bridge networks are deployed over a single host have zero latency, PERSECUS enables users to configure the TC utility provided with the images and delay containers' packets. Hence simulate a realistic WAN network in which communications are delayed due to physical distances. Despite containers being isolated processes, with our configuration, each container is accessible from the external via the Ethereum APIs running on a JSON-RPC HTTP server. We provide a different network port of the execution node to each container exposing the APIs.

The BSUT configuration detailed above is entirely managed by the *Docker Compose* tool [58] which enables the definition and execution of multi-container applications within a single YAML file, *i.e.*, *docker-compose.yml*. The BDM component of the PERSECUS Operating System is entitled to generate the compose file, which is used to initialise and start the BSUT with a single command using a specific configuration.

PERSECUS Operating System. The Operating System is deployed under the execution node. We implement all its components as standalone *Python* scripts. The BDM includes the adapters for both Parity and GoQuorum platforms, and it is called *Ethereum BDM*. The adapters use predefined Ethereum templates to configure the topology of the BSUT and the configuration parameters of the BSUT nodes, *i.e.*, the *genesis.json* and *node-config.toml*. Therefore, the Ethereum BDM automatically populates those templates

using the *genefis file* parameters of the configuration file received as input from the interface node. In particular, the *genesis.json* template defines the genesis block of a new blockchain, and it requires the following parameters: the total supply of *ETH*, *i.e.*, name of Ethereum's *digital asset*, the initial balance of *addresses*, and the *block-size*. The *node-config.toml* template is a TOML file used for node-specific configurations. It specifies the list of *nodes addresses* for communication routings (only network nodes) and the parameters of the *scheduler* like the size of the *mempool* and the number of *txss-threads*. Notwithstanding PERSECUS fosters flexible BSUT settings, the Ethereum BDM provides default values for two fundamental parameters, such as the *mempool* and *txss-threads* params.

The *mempool* value is set by default with the Equation (6.1), in which we call τ the *load-duration*, λ the *input rate*, s the *block-size*, and ρ the *block-period* parameters. The formula generates the optimum *mempool* size according to a given workload.

$$mempool = \frac{\tau(\lambda * \rho - s)}{\rho} \tag{6.1}$$

Indeed, if we consider ρ and s as constant values, we can determine at any point of τ the amount of transactions left in a queue.

Proof. We know that a blockchain core protocol produces a new block every ρ seconds. Hence, we can divide the time into τ/ρ steps, where the fist step is $t_1 = \rho$. We assume that for each step a new block is created and for any new block s transactions are collected from the *mempool*.

Base case ($t_1 = \rho$) First block crated. We have $\lambda * \rho$ transactions submitted, of which s are collected from the pool to create the first block, thus:

$$mempool_{t_1} = (\lambda * \rho) - s \tag{6.2}$$

Case ($t_2 = 2\rho$) We have a total of $\lambda * t_2 = \lambda * 2\rho$ transactions submitted, and $s * t_2/\rho = 2s$ transactions collected from the pool, thus:

$$\begin{aligned} mempool_{t_2} &= (\lambda * t_2) - (s * \frac{t_2}{\rho}) \\ &= \lambda * 2\rho - 2s \end{aligned} \tag{6.3}$$

Inductive Case ($t_n = \tau$) At the last step we have a total of $\lambda * \tau$ transactions submitted, and $s * \tau/\rho$ transactions collected from the pool, thus:

$$
\begin{aligned}
mempool &= (\lambda * \tau) - (s * \frac{\tau}{\rho}) \\
&= \frac{\rho(\lambda * \tau) - s * \tau}{\rho} \\
&= \frac{\tau(\lambda * \rho - s)}{\rho}
\end{aligned}
\tag{6.4}
$$

The *txss-threads* value is instead computed according with the Ethereum best practices [145] which suggest to configure the server threads in relation to the number of cores provided to the node, *i.e.*, *txss-threads*$= \#CPUs$.

Similarly to the Ethereum BDM, both the Resource Monitoring Agent and Chaos Testing Agent have been implemented as standalone Python scripts. The *Resource Monitoring Agent* makes use of the Docker SDK for Python [59] to connect with the Docker containers of the BSUT and invoke the `stats` command of the Docker daemon to obtain a live data stream of containers resources usage. Collected data is then stored into the *logs DB* at the end of the experiment. The *Chaos Testing Agent* integrates *Pumba* [57], a chaos testing tool for Docker containers. Pumba takes advantage of a network emulator, *i.e.*, *NetEm* [89], to reproduce adverse conditions in containerised networks. It can be used to simulate crashes of containers and emulate network failures. We use Pumba to create network partitions and Byzantine faults with the PERSECUS adverse mode. Specifically, we implement a command-line interface that enables PERSECUS' users to create network partitions of fixed duration. Under the hood, the Chaos Testing Agent invokes the Pumba's `delay` command which permits to delay the egress traffic of one or more containers, together with the `target` and `duration` options which specify respectively the targeted containers and the network emulation duration. Hence to create a partition the Pumba's `delay` command is run with a fixed configuration such that the delay `time` $>$ *`test-duration`*. Beyond partitions, the Chaos Testing Agent includes a special command, called `byzantine`, which enables the corruption of certain containers by triggering a Byzantine behaviour that corrupts packets. It makes use of Pumba's `corrupt` command to force a container to produce corrupted packets according to Bernoulli's probabilistic model.

PERSECUS Management System. The Management System is deployed under the interface node, and it is implemented as a single Python script, composed of different components. The first component is the *PERSECUS Controller*. It exposes a set of configuration parameters required to configure and start a new experiment. Then, there is the *BDM Connector* component that embeds a Python SSH utility to connect with the remote BDM. It provides the parameters required to generate a new *docker-compose.yml* file with the Ethereum BDM. Thus, the BDM Connector implements two methods to call via SSH the commands of the compose file to start and stop the BSUT.

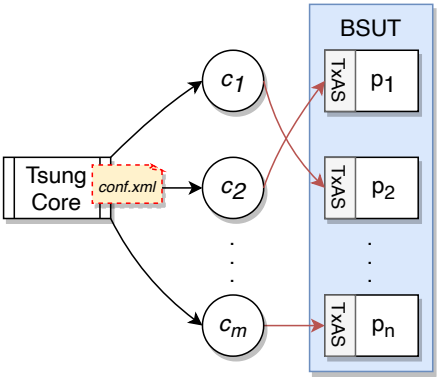


FIGURE 6.4: *PERSECUS workload generation with Tsung.*

The *WGM* is implemented as a Python module into the Management System, and it is built on top of *Tsung* [184], a load testing tool that can simulate distributed loads by running several virtual clients concurrently. We installed an instance of *Tsung core* into the interface node, then we used the Tsung XML configuration file to generate custom workloads. Broadly, we configured Tsung to produce a constant load of Ethereum transactions during a fixed period, i.e the *load-duration*. To accomplish this goal, we instructed Tsung to initiate a set of concurrent clients which started an HTTP connection with the TxAS of the BSUT nodes and then invoked the Ethereum method `eth_sendTransaction` through the nodes JSON-RPC API. This method executes a simple transaction on the Ethereum blockchain swapping a certain amount of digital assets between two accounts. In FIGURE 6.4 we give an overview of how we produced the workload with Tsung. To comply with the workload requirements detailed in Section 6.3.1, we implemented a custom configuration XML that enabled Tsung clients to start new HTTP connections with the nodes of the BSUT and generate a workload of equally balanced requests. Because in presence of large workloads, the HTTP handshakes generate network overheads that may affect performance, we reduced the number of HTTP connections by using a limited number of clients. Specifically, for a given *input rate*, we create an equivalent number of clients starting a new HTTP connection. Each connection remains open for the entire *load-duration*. Thus, clients send requests at a constant rate within the same connection. For instance, to produce a workload with *input rate* of 300 req/s we need $m = 300$ clients starting a single connection and sending one request per second, rather than starting 300 new HTTP connections per second.

Similarly to the WGM, we implemented the *MAM* as a Python module built into the Management System. The MAM integrates the adapters for both Parity and GoQuorum platforms. Although we required two different adapters to collect data from both platforms, Ethereum-based platforms have similar structures thus we used almost the same approach for both implementations. The logs collection triggers a polling service that checks the unprocessed transactions querying every 5 seconds the nodes' *mempool*. To limit overheads, each iteration of the polling service queries a different node. The polling

terminates when the *mempool* is empty, hence the logs are stored in a local directory of the interface node. Then, we create a *log parser* that processes the logs and extracts information on blocks and transactions used to create the *PERSECUS Dataset* and *Forks Dataset*. Similarly, we use a second data parser that takes as input the Forks Dataset and the final states of the blockchain and generates the *Security Report*.

6.5 Experimental Evaluation

6.5.1 Environment and Deployment

Testbed. The environment used to deploy and evaluate PERSECUS was composed of a PowerEdge R730xd rack server with 56 logical processors Intel(R) Xeon(R) CPU E5-2695 v3 2.30GHz running the VMware ESXi hypervisor. We ran two virtual machines, one dedicated to the interface node and another to the execution node. The former was provisioned with 8 CPUs, 8GB RAM and 50GB hard drive, the latter with 36 CPUs, 144GB RAM, and 300GB hard drive. Both VMs ran Ubuntu *18.04.2 LTS*. Therefore, on the execution node we used Docker *v18.09.7* and Docker Compose v1.24.0 for container virtualisation, whereas we used Tsung *v1.7.0* for the WGM.

Experiments Setups. To evaluate PERSECUS, we ran a benchmark of the blockchain platforms Parity and GoQuorum. We used PERSECUS to assess *performance*, *scalability*, and *security* of those platforms employing different consensus protocols, namely *AuRa*, *Clique*, and *IBFT*. *AuRa* [148] is Parity's PoA [51] protocol which promises to offer high performance and fault tolerance assuming a synchronous network. *Clique* [156] is the GoQuorum's PoA which offers high performance at the cost of relaxed consistency [51], while IBFT is the first BFT consensus built for blockchains, inspired by the classical PBFT [37], also implemented in GoQuorum. We developed two customised docker images [8, 9] containing the Parity *v2.4.5-stable*, and GoQuorum *v2.6.0-updated_ibft* core softwares. In this experimental evaluation we deploy three instances of BSUTs, distinguished as *platform-consensus*, *i.e.*, (i) Parity-AuRa, (ii) GoQuorum-Clique, (iii) GoQuorum-IBFT. Then, we stress-test those platforms under various configurations and network sizes; for the sake of simplicity, we consider BSUTs nodes with network, participation, and validator roles simultaneously. The Docker containers are provisioned with 4 CPUs and 16GB Memory each; the BSUT nodes host single blockchain accounts with unlimited *ETH* balance - ETH balance is required to process workloads. Communications between nodes are delayed by a common factor $\Theta = (200ms, 40ms, 25\%, normal)$. Each stress-test is composed by 6 minutes *test-duration*, and 4 minutes *load-duration*, with a workload characterised by `eth_sendTransaction` *req_type*s, and $|\Sigma| = \{300, 600\}$ clients producing 300-600 req/s *input rate*s.

Evaluation Metrics. We compute performance, security, and scalability metrics using data from the *PERSECUS dataset*, *Forks dataset*, and the *Security Report*. To measure performance, we use metrics of *throughput* and *latency*, whereas to measure security we use metrics of *persistency* and *termination*:

- *Throughput*: Measured as the number of transactions finalised per second. It is referred to as *transactions per second* (TPS);
- *Latency*: Finalisation time of a transaction tx. It is measured as the average latencies of a block, such as $block\_latency = \frac{1}{bs}\sum_{i=1}^{bs} latency_{tx_i}$, with bs = *block-size*, and $latency_{tx_i} = ts^f_{tx_i} - ts^s_{tx_i}$.
- *Scalability*: Measured as the variation of throughput and latency when altering the number of nodes and the *input rate*;
- *Persistency*: Measured with a boolean value (T, F). It is T, if the BSUT nodes converge to the same state of the blockchain at the end of the experiment. Hence no forks remain unresolved. Otherwise it is F;
- *Termination*: A transaction tx terminates if and only if it reaches finality, hence the ts^f value is not null.

6.5.2 PERSECUS Evaluation

We divide the evaluation of PERSECUS into four benchmarking activities. First, we use PERSECUS to find the best parameters setup of the three BSUT instances running eight different configurations. Then, we use the outcome to assess and compare *performance*, *scalability*, and *security*.

Parameters Tuning

Among several configuration parameters that characterise the Ethereum protocol, it has been proved that the transaction *scheduler* and *consensus* components have a direct effect on Ethereum's performance and security [116, 170]. Hence, to assess Ethereum blockchains, those components must be perfectly tuned. Indeed, a misconfiguration may cause bottlenecks and protocol failures that may generate measurement unbalances.

PERSECUS' flexibility enabled us to tune the params of each BSUT. Specifically, we tested the behaviour of the BSUTs deploying a network of $|\Pi| = 4$ nodes, using different combinations of parameters, and we stress-tested them with a workload of 300 req/s. The *scheduler* was configured using PERSECUS's optimal configuration presented in section 6.4. Differently, the *consensus* component was tested varying *block-size*

Test	*block-size (wei)*	*block-period (s)*
T1	40M	5
T2	40M	2
T3	20M	5
T4	20M	2
T5	10M	5
T6	10M	2
T7	5M	5
T8	5M	2

TABLE 6.2: *Tests T1-T8 varying block size and block period*

and *block-period* params. Indeed, it has been proved that those params affect the performance and security of a blockchain impacting the consensus [14, 170]. Ethereum platforms measure the *block-size* in millions of wei, *i.e.*, the smallest unit of ETH, whereas *block-period* is expressed in seconds. The amount of *wei* of a block indicates the total amount of transactions fees collected in a block. As already discussed in section 4.6.2, Ethereum's *block-size* and *block-period* parameters can be used in combination with the transaction fee, *i.e.*, Tx_{fee} to compute the maximum throughput, *i.e.*, *i.e.*, TPS_{max}, given a workload of same *req_type*s [194]. TPS_{max} can be calculated with eq. (6.5).

$$TPS_{max} = \frac{bloksize}{Tx_{fee} * blockperiod} \tag{6.5}$$

To maximise the TPS, a developer might be tempted to use large block sizes and short block periods. However, this configuration generates heavy blocks, hard to propagate over the network, and as a consequence consensus errors may happen [30]. To find the best configuration we ran 8 tests, T1-T8, with various *block-size*s and *block-period*s. Then, we observed the behaviour of the blockchains measuring their throughput, latency, and the percentage of terminated transactions. TABLE 6.2 shows the parameters we used, while TABLE 6.3 shows the maximum TPS per test we calculate with eq. (6.5), assuming $Tx_{fee} = 21k$ *wei*, *i.e.*, the fee Ethereum assigns to transactions generated with the method `eth_sendTransaction`.

Parity-AuRa Consensus Tuning. FIGURE 6.5 shows Parity's transactions termination. We observe from the diagram that T1, T3, T5, and T7 ensured termination achieving the 100% of terminated transactions. Differently, T2, T4, T6, and T8 violated termination, obtaining all transactions rejected. All those tests had *block-period*$= 2s$. This result shows that AuRa cannot guarantee termination when configured with such a block period. AuRa divides time into equal steps of *block-period* duration; for each step, a new block is proposed by the leader and verified by the majority of validators [51]. Too short block periods generate consensus failures due to missing blocks: if we assume two steps S_1 and S_2, and two respective blocks b_1, b_2; during S_1 some validators

Test	*TPS MAX*
T1	380
T2	952
T3	190
T4	476
T5	95
T6	238
T7	47
T8	119

TABLE 6.3: *Maximum Ethereum TPS of T1-T8 configurations*

fail to verify b_1 before S_2 begins. Hence, there is always a majority of validators that do not receive b_1 within S_1. Consequently, some blocks never get verified, and their transactions remain in the *mempool* until timing-out.

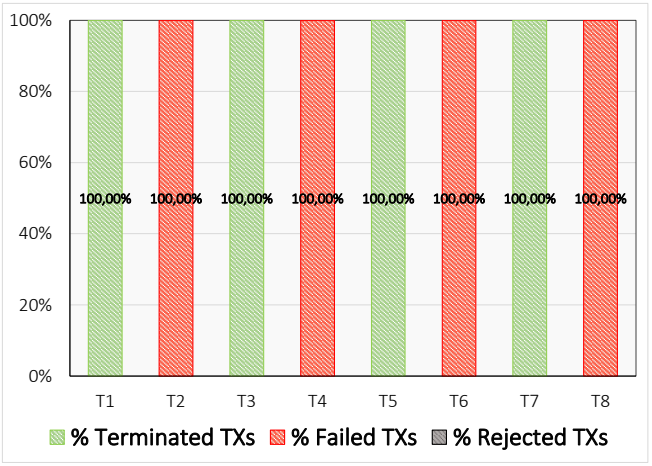
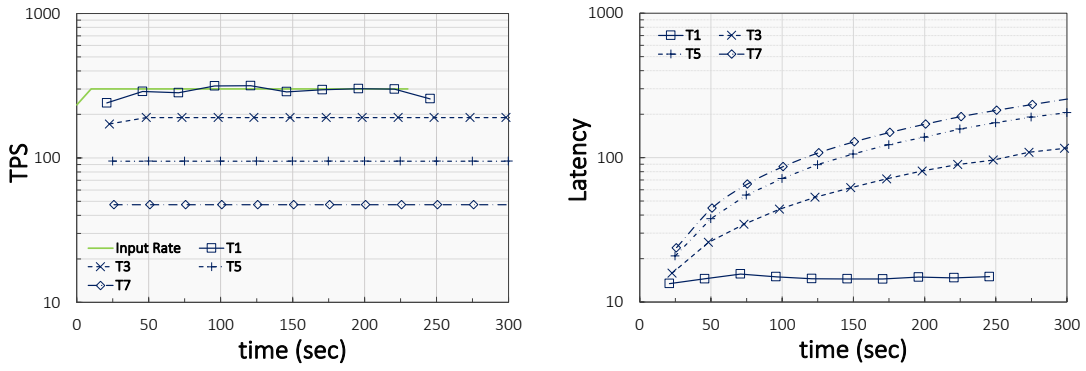


FIGURE 6.5: *Parity-AuRa transactions termination varying block size and block period*

FIGURE 6.6 illustrates the TPS and latency measured for T1, T3, T5, and T7. We observe that, given an *input rate* of 300 req/s, T1 was the best configuration, achieving TPS equal to the input, while maintaining low latency. Differently, we observe other configurations' TPS decreasing according to lower block sizes. Then, from FIGURE 6.6b, we observe that latencies with T3, T5, and T7 grew exponentially in time. The TPS of those configurations saturates according to their maximum values TABLE 6.3. As a result, accepted transactions remained stuck into the *mempool*, and their finalisation times grew together with their latencies. On the other hand, T1 achieved linear latency, which indicates that the TPS did not saturate and transactions executed at constant latency without queueing in the *mempool*.

GoQuorum-Clique Consensus Tuning FIGURE 6.7 shows GoQuorum-Clique's transactions termination. We observe from the diagram that T1, T2, T3, and T4 ensured termination achieving the 100% of terminated transactions. Differently, T5, T6, T7,



(A) TPS (B) Latency

FIGURE 6.6: *Parity-AuRa TPS and Latency varying block size and block period*

and T8, rejected some transactions in the TxSS. This issue revealed a bug in the TxSS serialisation of the GoQuorum core software. Indeed, the TxSS failed to serialise and sign parallel `eth_sendTransactions`. Specifically, at the time of signature, the TxSS assigned the same *nonce*s to transactions. However, Ethereum's *nonce*s must be unique to avoid double-spending [40, 194]. Observing the logs of experiments T5-T8, we realised that some concurrent transactions received the same *nonce* by the TxSS. In that case, only one transaction was signed and subsequently accepted into the TxPoolS, while the others got rejected with a `known transaction` error message. It is worth to note that T6 resulted in the configuration with the lowest number of rejected transactions. T6 achieved the higher TPS (FIGURE 6.8a), hence with fewer transactions in the *mempool*. In that case, the probability of having two or more transactions with the same *nonce* was lower for T6 than T5, T7, and T8.

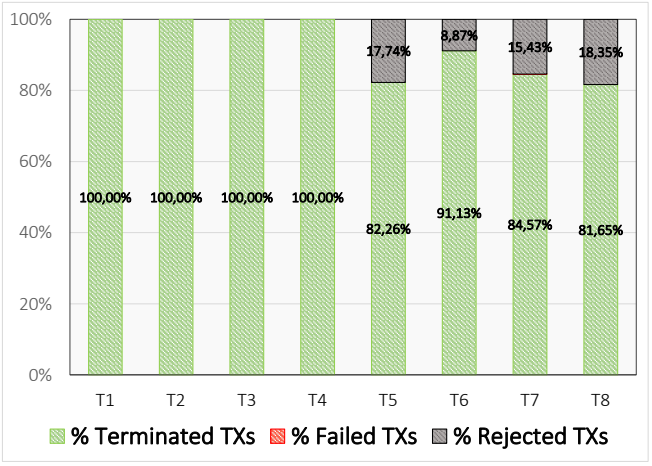
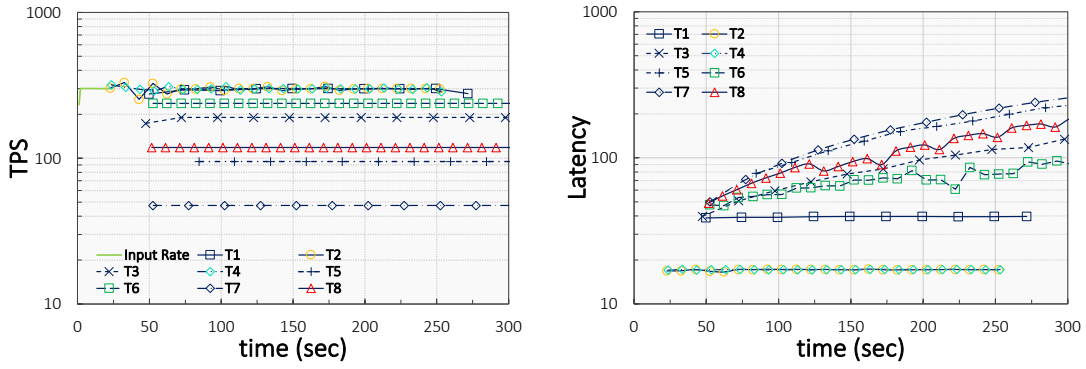


FIGURE 6.7: *GoQuorum-Clique transactions termination varying block size and block period*

FIGURE 6.6 shows the TPS and latency measured with PERSECUS. The graphs show that T2 and T4 are the best settings, achieving a TPS equal to the input rate and the lower latencies. Therefore, the *block-period = 2s*, enabled both configurations to

anticipate the transactions finalisation of $\approx 25s$ respect the others. Beyond T2 and T4, T1 was the third-best configuration in terms of throughput and latency, achieving similar TPS while preserving a linear latency equal to 40s. Differently, the TPS of the other configurations saturated, hence transactions queued in the *mempool* and latencies grew exponentially. Finally, we observed how configurations with *block-period* of 5s achieved the worst performance.



(A) TPS (B) Latency

FIGURE 6.8: *GoQuorum-Clique TPS and Latency varying block size and block period*

GoQuorum-IBFT Consensus Tuning FIGURE 6.9 shows GoQuorum-IBFT's transactions termination. Similarly to GoQuorum-Clique, we observed T1, T3, T5, and T7 with 100% of terminated transactions, while T5, T6, T7, and T8 experiencing rejected transactions due to the `known transaction` error.

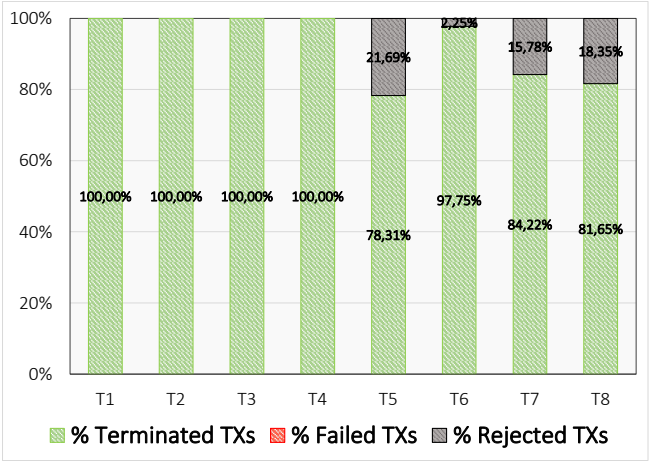


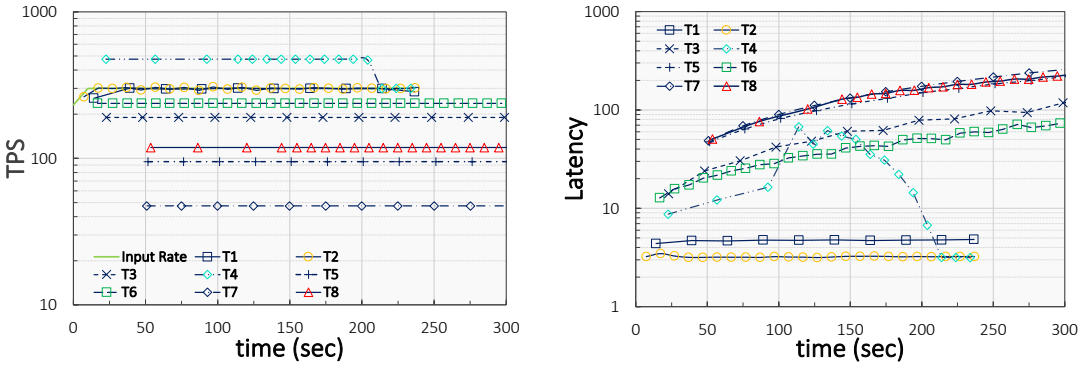
FIGURE 6.9: *GoQuorum-IBFT transactions termination varying block size and block period*

FIGURE 6.10 depicts the GoQuorum-IBFT's TPS and latencies. The graphs show that T1 and T2 obtained the best performance with maximum throughput and linear latencies. Beyond that, FIGURE 6.10b shows T4's TPS outperforming T1 and T2, despite an irregular latency. Indeed, T4's latency indicates a protocol slowdown. By inspecting the logs, we figured out that the block period was too short to enable IBFT nodes to

complete the protocol with a *block-size* of 20M *wei*. IBFT does not allow forks; for each protocol step (which is determined by the *block-period*) a new block must be appended on the blockchain. However, we observed that large blocks require too long propagation times, which need more time than the *block-period* itself. In that case, the steps miss their blocks, and consequently, the nodes are forced to commit empty blocks and move on to the next step. Finally, T3, T5, T6, T7, and T8 behaved as expected achieving maximum TPS and exponential latencies with a progression proportional to block sizes and periods.

Performance Evaluation

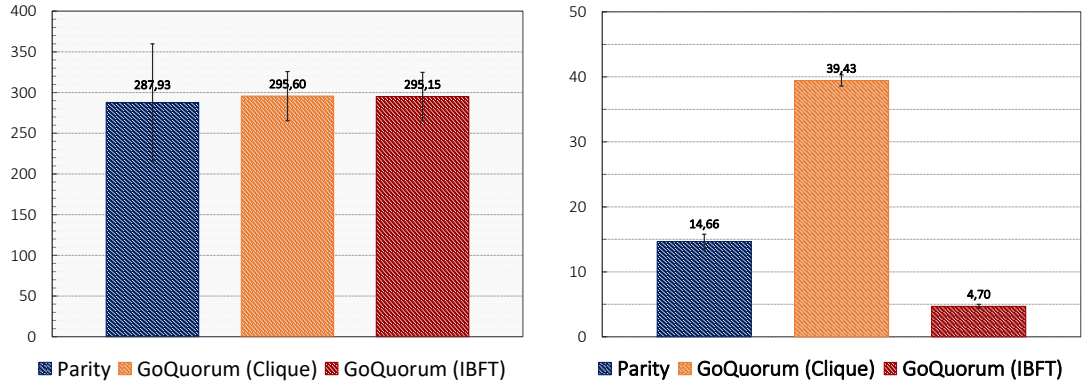
The parameters tuning showed that, with a workload of 300 req/s, T1 was the best configuration for all three instances under test. Hence, we used T1 to compare performance of those instances. FIGURE 6.12 and FIGURE 6.11 show the TPSs and Latencies of Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT. From FIGURE 6.12a and FIGURE 6.11a, we observe that all three instances achieved maximum TPS, equal to the input, with an average TPS very close to 300. Indeed, we observed that T1 perfectly handled 300 req/s without saturating. On the other hand, in FIGURE 6.12b we can observe that, although all the three have linear latencies, GoQuorum-IBFT outperforms the others with latencies between 4-5s.



(A) TPS (B) Latency

FIGURE 6.10: *GoQuorum-IBFT TPS and Latency varying block size and block period*

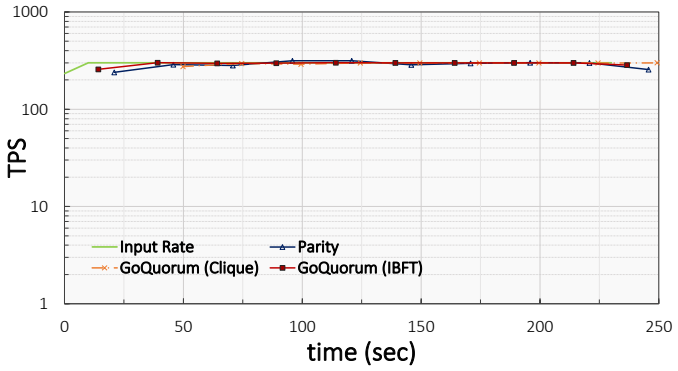
The IBFT protocol does not allow forks and guarantees transactions instant finality when a new block is committed. Differently, AuRa and Clique transactions get finalised after a certain period called *finalisation time* which is proportional to the number of validators [51]. We measured the finalisation time using the PERSECUS Dataset using the transaction's finality and submission timestamps, *i.e.*, $finalisation\_time = ts^f - ts^s$. Moreover, Clique resulted in the slowest protocol in terms of latencies because it allows forks. During its execution, Clique needs *reorgs* to resolve forks affecting the finalisation of transactions hence their latencies. This is even more clear in FIGURE 6.11b, in which we observe the average latencies of the three instances. From the diagram, we



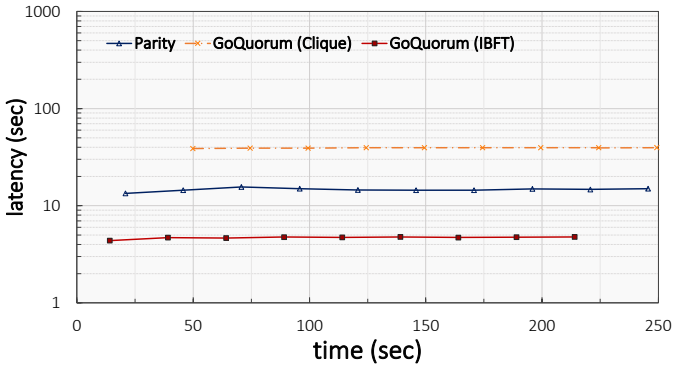
(A) AVG TPS (B) AVG Latency

FIGURE 6.11: *Comparison of Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT average performance*

observe that on average, IBFT requires 4,70s to finalise transactions, which is the time of IBFT consensus, while AuRa and Clique need respectively 14,66s and 39,43s. In FIGURE 6.14 we depicted the average finalisation times of each instance, confirming that IBFT achieved instant finality, while Clique required more time due to blockchain *reorgs*.



(A) TPS



(B) Latency

FIGURE 6.12: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT performance*

Finally, in FIGURE 6.13, we compare the block sizes measured over time. GoQuorum-IBFT outperformed the others obtaining blocks of equal dimensions during the whole

experiment. Differently, both Parity-AuRa and GoQuorum-Clique produced blocks of various sizes. AuRa's validators periodically produced blocks peaking in a range between 700 and 1900 transactions per block, while Clique produced blocks of sizes ranging from 1200 and 1900 transactions per block.

Scalability Evaluation

We measured TPSs and latencies with 300 and 600 req/s *input rate*s, and networks of $|\Pi| = \{4, 8, 16\}$ nodes deployed under configuration T1. The scalability evaluation embraced the following experiments: (i) compare transaction termination and transaction loss, (ii) measure scalability of single BSUTs, (iii) compare scalability of BSUTs.

Evaluation of transactions termination. As depicted in figures 6.5, 6.7, and 6.9, with a workload of 300 req/s and a network of 4 nodes, the BSUTs deployed under configuration T1 finalised 100% of transactions. However, this value was altered by increasing the workload and the network size parameters. FIGURE 6.15 illustrates the percentage of transactions finalised and transactions loss measured by increasing the number of nodes and the input. With 4 nodes, Parity-AuRa finalised 100% of transactions. Differently, both GoQuorum's instances suffered of $\approx 30\%$ transactions loss (FIGURE 6.15b).

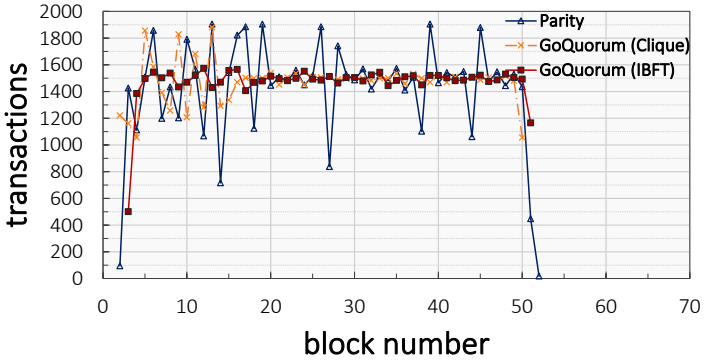


FIGURE 6.13: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT block sizes*

In this case, transactions got rejected by the `known transaction` error - see GoQuorum's parameters tuning sections above. With 8 nodes, all Parity-AuRa's transactions terminated, while for both GoQuorum's, the transactions-loss decreased to 20%. Indeed, by increasing the number of nodes, we observed a reduction in rejected transactions. By distributing the load among more nodes, we reduced the pressure over the TxSSs. Hence, the scheduler processed transactions without generating the serialisation error. A different behaviour occurred with 16 nodes, in which we obtained higher transaction loss in all three BSUTs. In that case, we realised that the blocks created with T1's configuration struggled to be propagated within larger networks. Consequently, Parity-AuRa experienced 4% of transactions loss due to consensus failures, while in both GoQuorum instances we observed an increase of serialisation errors. To understand this behaviour

we need to look at how both protocols work. Clique and IBFT proceed in steps, and at each step, only one node can generate a new block. With 16 validators, each node drains the *mempool* once every 16 steps. Nodes' pools result fuller, hence the probability of reproducing the serialisation bug increases - more frequently the TxSS assigned existing *nonce*s to concurrent transactions.

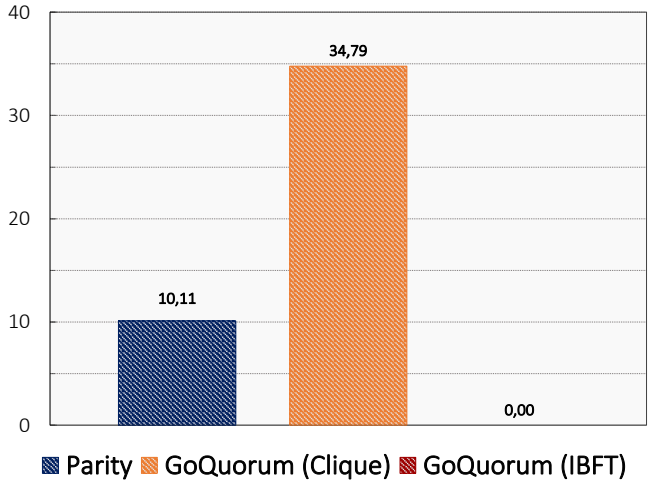
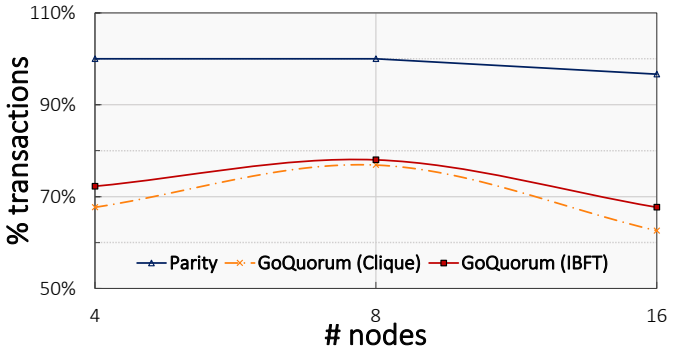
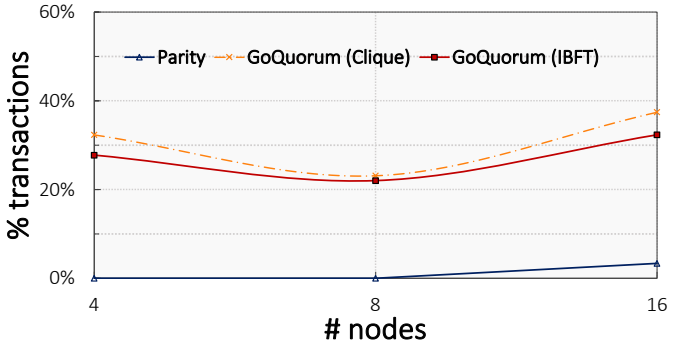


FIGURE 6.14: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT average block finalisation times*



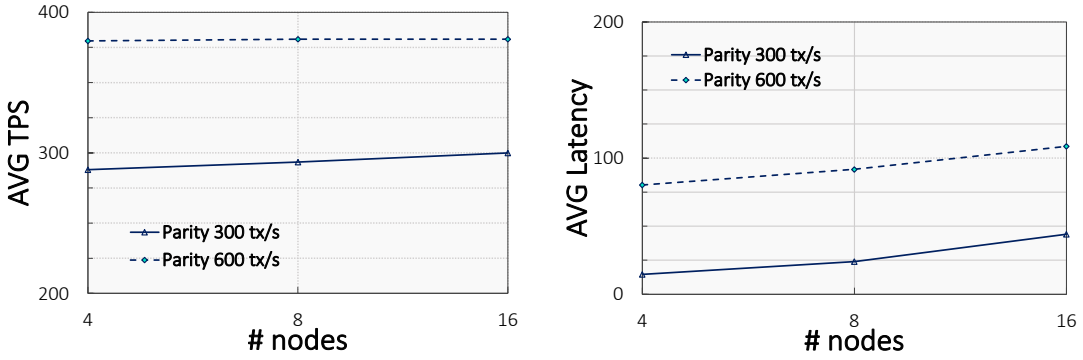
(A) Terminated transactions



(B) Transactions loss

FIGURE 6.15: *Evaluation of terminated transactions and transactions loss with 600 req/s input rate*

Parity-AuRa scalability. FIGURE 6.16 illustrates the average TPSs and latencies measured for *input rates* of 300 and 600 req/s, and networks of 4, 8, and 16 nodes. From FIGURE 6.16a we observe that in terms of average TPS, a larger network offered slightly better performance. Despite that, average latency increased either with larger networks or loads. Indeed, from FIGURE 6.16a we can observe that with 600 req/s the nodes achieved maximum TPS saturating the throughput (from 6.3: $TPS_{T1max} = 380$). As a result, transactions queued in the *mempool*s and their finalisation time grew exponentially leading to latencies on average 50s slower than the ones measured with a load of 300 req/s. Therefore, average latencies grew also by increasing the network size. AuRa's *finalisation time* is related to the number of validators [51], hence by increasing the number of nodes we obtained worst latencies - FIGURE 6.16b. To summarise, by increasing the number of nodes Parity-AuRa TPS scales to TPS_{max}, at the cost of worst latencies.

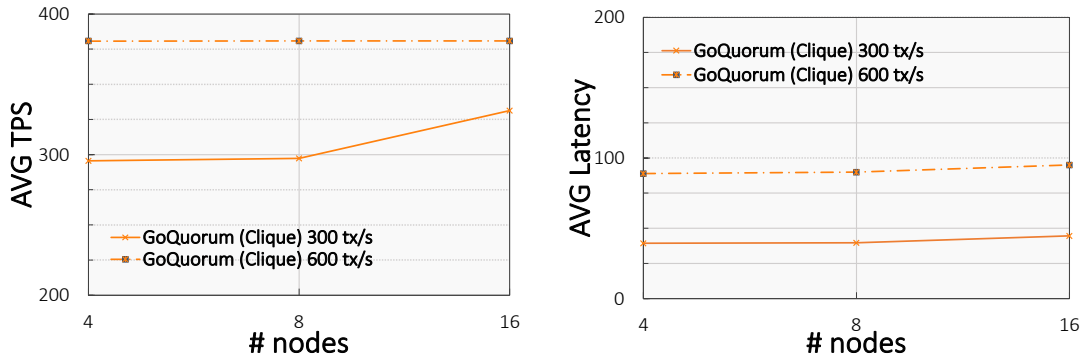


(A) Throughput (B) Latency

FIGURE 6.16: *Parity-AuRa scalability with 300 and 600 req/s input rates, and networks with 4, 8, and 16 nodes*

GoQuorum-Clique scalability. FIGURE 6.17 illustrates the average TPSs and latencies measured for *input rates* of 300 and 600 req/s, and networks of 4, 8, and 16 nodes. From FIGURE 6.17a we observe that, with an input of 300 req/s, GoQuorum-Clique achieved the same average TPS either with 4 and 8 nodes, whereas with 16 nodes the throughput improved by 25 TPS. By increasing the input to 600 req/s, the throughput is saturated at TPS_{T1max} regardless of the network size. Differently, latencies graphs showed the same trend with both *input rate*s, as depicted in FIGURE 6.17b, with a slight increase of average latencies in larger networks. Indeed, as well as in AuRa, Clique's finality strongly depends on the network size [51]: more validators imply higher finalisation times. Finally, like for Parity-AuRa, GoQuorum-Clique's latencies increased of $\approx 50s$ with higher *input rate*s due to transactions queued into the *mempool*.

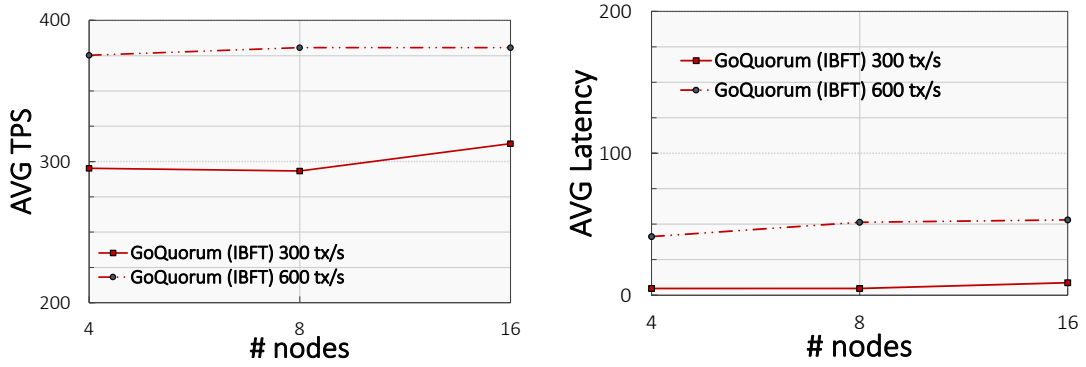
GoQuorum-IBFT scalability. FIGURE 6.18 illustrates the average TPSs and latencies measured for *input rate*s of 300 and 600 req/s, and networks of 4, 8, and 16 nodes. As shows FIGURE 6.18a, IBFT throughput revealed the same trend as Clique's. Specifically, with 300 req/s the TPS increased with a 16 nodes network, whereas with 600 req/s IBFT's throughput saturated reaching the TPS_{T1max} regardless of the network size.



(A) Throughput (B) Latency

FIGURE 6.17: *GoQuorum-Clique scalability with 300 and 600 req/s input rates, and networks with 4, 8, and 16 nodes*

As well as Clique, GoQuorum-IBFT's latencies slightly increased in larger networks. Although it is proven that PBFT-like protocols, such as IBFT, do not scale [37], we showed that GoQuorum-IBFT can afford networks up to 16 nodes without sacrificing performance. We leave as future work scalability evaluation in larger networks.

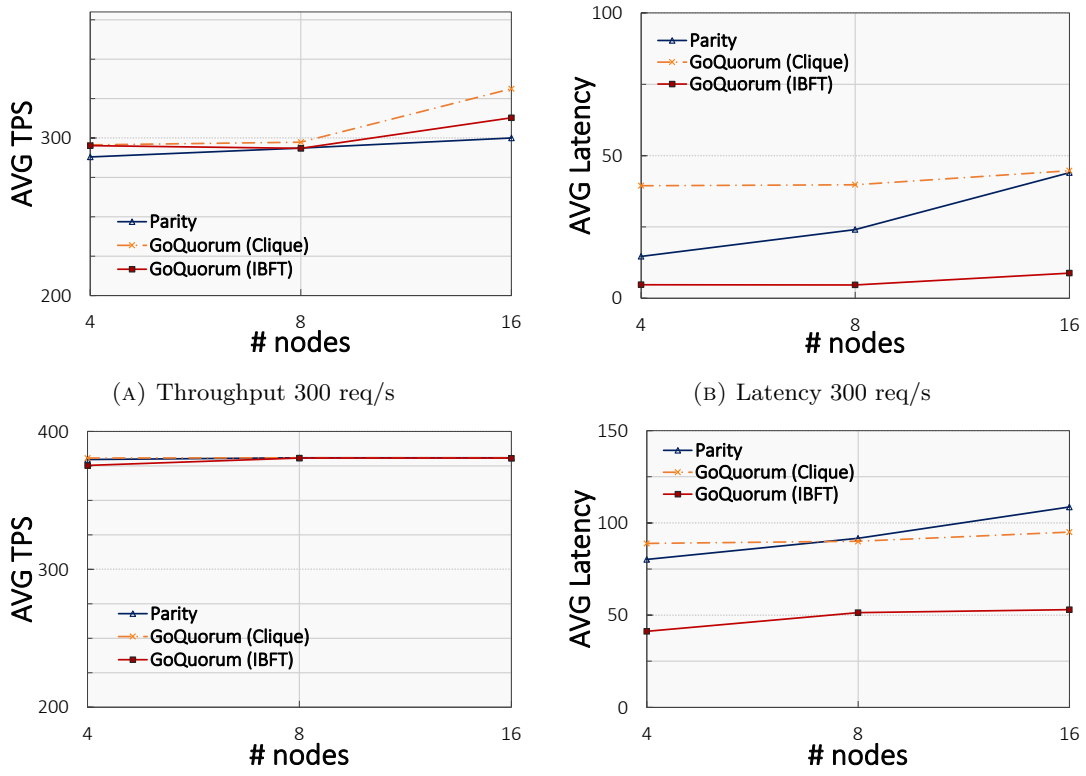


(A) Throughput (B) Latency

FIGURE 6.18: *GoQuorum-IBFT scalability with 300 and 600 req/s input rates, and networks with 4, 8, and 16 nodes*

Scalability comparison. We used the results described above to compare the scalability of Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT. FIGURE 6.19 summarises the results with both 300 and 600 req/s *input rate*s. From FIGURE 6.19a and FIGURE 6.19b we observe that, with a load of 300 req/s, a network of 16 nodes improved the throughput of each BSUT but simultaneously caused higher latencies. In this scenario, GoQuorum-Clique obtained the best scalability result: increasing the nodes resulted in higher TPS without burdening latency. Similarly, GoQuorum-IBFT showed good scalability, improving TPS with a minimum increment of latency. Finally, Parity-AuRa obtained the worst results, with a minimum increment of TPS despite an exponential growth of latencies. The BSUTs obtained a similar behaviour also with 600 req/s *input rate*. Although all platforms achieved maximum throughput as showed in FIGURE 6.19c, scaling

the network size provoked a drastic growth of Parity-AuRa's latency. Indeed, from FIGURE 6.19d we observe how Parity-AuRa obtained the worst latency with a network of 16 nodes, exceeding GoQuorum-Clique. This result suggests that Parity-AuRa cannot scale well in large networks.



(A) Throughput 300 req/s

(B) Latency 300 req/s

(C) Throughput 600 req/s

(D) Latency 600 req/s

FIGURE 6.19: *Scalability comparison with 300 and 600 req/s input rates, and networks with 4, 8, and 16 nodes*

Security Evaluation

We evaluate the security of the BSUTs through PERSECUS' *adverse* mode. We stress-tested a network of $|\Pi| = 4$ nodes deployed under T1 configuration with a workload of 300 req/s *input rate*. Within each test, we started the *Chaos Testing Agent* to simulate an adverse scenario with a duration of 2 minutes. Such an adverse scenario produced both a network partition and a Byzantine fault. Specifically, we configured the agent to create two isolated groups unable to communicate with each other. Thus, we corrupted the outgoing packets of one node through the `byzantine` command of the agent. In this scenario, we measured security by evaluating the metrics of persistency and termination. To evaluate persistency we referred to PERSECUS' *Security Report* produced by the *PERSECUS Controller* component.TABLE 6.4 summarises the results obtained for each BSUT instance. Specifically, both GoQuorum instances ensured persistency, whereas Parity-AuRa violated persistency due to unresolved forks. Parity-AuRa's persistency

BSUT	Node	Final state	Persistency
Parity-AuRa	1	5175...2782	F
	2	5376...5260	
	3	5376...5260	
	4	5376...5260	
GoQuorum-Clique	1	1655...2020	T
	2	1655...2020	
	3	1655...2020	
	4	1655...2020	
GoQuorum-IBFT	1	2503...4714	T
	2	2503...4714	
	3	2503...4714	
	4	2503...4714	

TABLE 6.4: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT persistency. The final states indicate the hash of both the blocks identifiers and numbers*

failure is also detailed in FIGURE 6.20, which illustrates the number of forks detected within the experiments. The graph shows how Parity-AuRa started forking during the adverse scenario (area in between the two vertical dotted lines). Thereafter, one fork remained unresolved - *node1* never synchronised. Differently, GoQuorum-Clique produced forks that eventually got resolved, while GoQuorum-IBFT never forked.

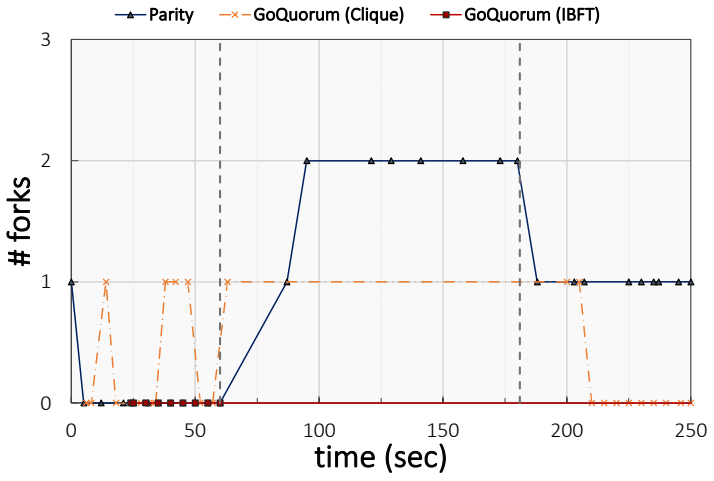


FIGURE 6.20: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT forks measured with PERSECUS' adverse mode. The vertical dotted lines delimitate the adverse period*

Despite persistency, we observed that adverse deployment scenarios also impact the termination property of the BSUTs. In FIGURE 6.21 and FIGURE 6.22 we outlined respectively the transactions termination rate and the percentage of terminated/failed/rejected transactions measured during the experiments. Specifically, the graph of FIGURE 6.21 depicts the number of submitted transactions that correctly terminated (eventually). During the adverse period, the BSUT started refusing some transactions. In that period,

the transactions termination rate of Parity-AuRa was halved, whereas both GoQuorum's reduced by 1/4. In Parity-AuRa, all transactions submitted toward the partitioned node failed because that node never recovered from the partition. Therefore, all the transactions submitted to the subverted node remained unprocessed into the TxAS. However, those transactions got processed once the node returned honest, as shown in FIGURE 6.21 with the blue line persisting after the input rate.

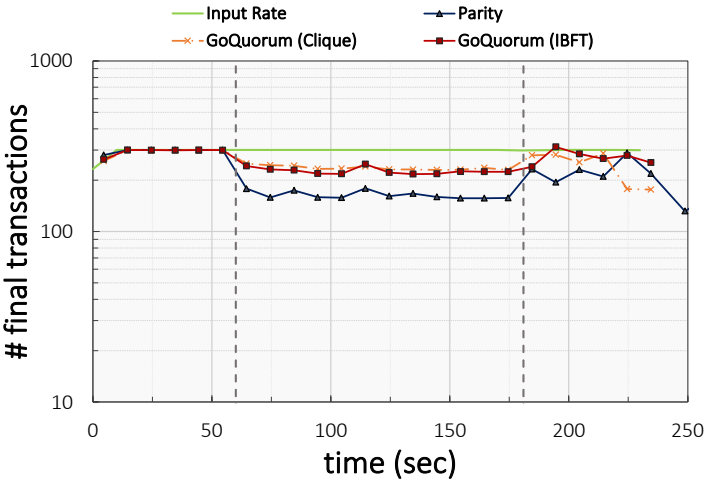


FIGURE 6.21: *Comparison of Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT termination rate with PERSECUS' adverse mode. The vertical dotted lines delimitate the adverse period*

On the other hand, the adverse mode caused in both GoQuorum platforms rejected transactions. GoQuorum considered *invalid* and thus rejected, all the transactions submitted through a Byzantine node. However, looking at FIGURE 6.21, we note that both GoQuorum platforms rejected some transactions even after the adverse mode. In this case, transactions were rejected due to a new bug spotted in the GoQuorum software. Specifically, after a network partition, the faulty nodes started a *read-only* mode to favour reorgs. Such nodes never recovered from that mode and therefore they started rejecting transactions. Although this behaviour prevented forks, it violated termination[21].

Finally, in FIGURE 6.22 we outlined the percentage of terminated, failed, and rejected transactions. The graphs confirm the results introduced above, showing for Parity-AuRa $28,41\%$ of failed transactions, then for GoQuorum-Clique and GoQuorum-IBFT respectively $16,76\%$ and $14,64\%$ rejected transactions.

[21]We discussed this bug with the GoQuorum developers, and we collaborated to fix it with a GitHub pull request [177].

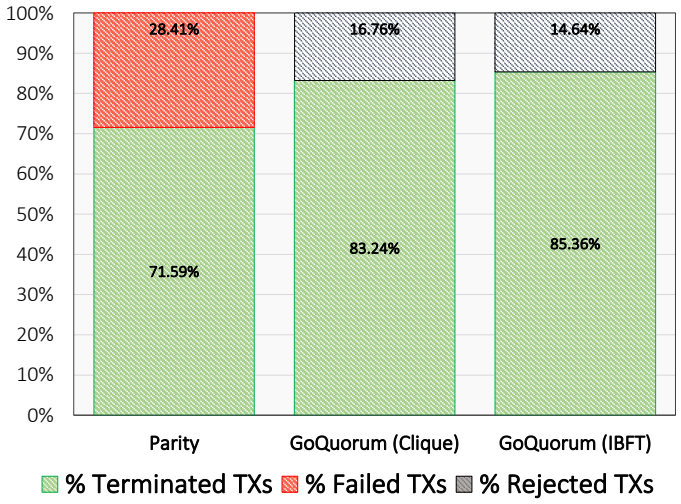


FIGURE 6.22: *Parity-AuRa, GoQuorum-Clique, and GoQuorum-IBFT terminated failed, and rejected transactions measured with the PERSECUS adverse mode*

6.6 Discussion

In this chapter, we presented PERSECUS, a dependability benchmark for blockchain systems. PERSECUS provides a flexible, extensible, and scalable tool for testing different blockchain systems under various configurations and simulating real deployment scenarios. PERSECUS aims at enhancing critical aspects that afflicted current blockchain benchmarks. Specifically, it fosters (i) optimised experimental setups free of misconfigurations that affect measurements, (ii) faultloads to evaluate systems under adverse conditions like network partitions and malicious actors, (iii) efficient workload generation, (iv) efficient and precise metrics computation. We also provided systematic benchmarking procedures which define the fundamental rules and steps that a benchmarking experiment must follow. Through experimental evaluation, we validated the implementation of PERSECUS based on two Ethereum blockchain platforms, namely Parity and GoQuorum. With PERSECUS, we analysed and compared performance, security, and scalability of three blockchain instances running different consensus protocols, such as AuRa, Clique, and IBFT. First, we identified an optimum configuration of parameters for all three blockchains under test, using a fixed workload and network size. Then we used this configuration to compare performance and scalability of all three platforms, measuring throughputs and latencies. Finally, we assessed security simulating a faultload with a network partition and one subverted node acting maliciously. Through experimental evaluation, PERSECUS revealed performance bottlenecks affecting Parity and GoQuorum platforms, besides their consensus component. Specifically, we observed that misconfigured parameters led Parity to violate termination, whereas GoQuorum rejected some transactions. In addition, we spotted a bug in GoQuorum's serialisation scheduler that led to unsigned transactions being refused in case of full *mempool*s. In terms of

performance, we observed that all three platforms achieve maximum throughput under their optimal configuration, whereas GoQuorum-IBFT latencies outperform the others thanks to IBFT's instant finality. We, therefore, measured the variation of both throughput and latency altering the network size and the number of nodes to assess scalability. The experiments showed us that Parity-AuRa cannot scale well, while both GoQuorum's performances remained unaltered. However, we believe that experiments with a larger network would be more indicative to evaluate GoQuorum's scalability. We leave further experiments for future works. Finally, we measured security assessing persistency and termination metrics. We observed that in the case of faultloads, Parity-AuRa breaks both security properties, while GoQuorums favours persistency over termination. The security evaluation outlined a second bug in the GoQuorum software that caused transactions rejection when network partitions occur.

Chapter 7

Conclusions

The aim of this thesis is to study *security*, *dependability* and *performance* aspects of blockchain systems, providing a foundational suite of *taxonomies*, novel *methodologies* and *benchmark* procedures to assess them both theoretically and experimentally.

The first contribution is a systematic taxonomy of security and dependability properties to assess blockchain systems according to platforms, consensus protocols, and smart contracts applications. Specifically, we defined the foundational attributes that represent those properties in blockchain, thus we evaluate whether such attributes are verified by each component analysed. In particular, we compared the security of five blockchain platforms and consensus protocols, divided into permissionless systems, such as Bitcoin with PoW, Ethereum 2.0 with PoS, Algorand with PPoS, and permissioned systems, such as Ethereum private network with PoA and Hyperledger Fabric with PBFT. From the analysis emerged that public permissionless blockchains are preferable when data integrity and availability are paramount properties in large decentralised systems, whereas permissisoned blockchains are a better choice when confidentiality, authentication, and authorisation guarantees are needed. Finally, we evaluate the vulnerabilities of smart contracts and we assert that decentralised applications are extremely susceptible to attacks that violate the confidentiality, integrity and authorisation of the systems.

Thereafter, we provided a comprehensive methodology for assessing security and performance of blockchain consensus protocols. Hence we introduced METHUS which defines both qualitative and quantitative approaches. We validate METHUS by comparing two families of consensus protocols, namely the PoA and PBFT, specifically we compared two PoAs, namely AuRA and Clique, against one PBFT-like protocol, namely IBFT. The METHUS analysis showed us that in a scenario with network faults and subverted nodes, the PoA consensus protocols lack integrity and availability guarantees, despite AuRa showing better performance. In particular, with the qualitative evaluation, we demonstrate that AuRa is subject to an attack that compromises the security, while

Clique may suffer from inconsistent states; then we also showed by experimental evaluation that, in presence of faults, PoAs violate integrity and availability, whereas in IBFT those properties are guaranteed despite slightly worse performance.

In the last part of this thesis, we define two benchmarking tools for blockchain systems, namely PETHARD and PERSECUS. The former has been designed according to the METHUS quantitative methodology, to evaluate and compare two consensus protocols used in permissionless blockchains, namely PoW and PPoS, whereas the latter has been proposed to overcome the limitations of state-of-the-art blockchain benchmarks. PETHARD showed us that PPoS represents a sustainable and efficient alternative to PoW for building open permissionless systems, providing better performance and simultaneously addressing the scalability issues that we rather measured for PoW. However, a fair comparison of security properties for these protocols was not possible with PETHARD, and we left it for future works. We, therefore, proposed PERSECUS ad the extension of PETHARD. It has been implemented with the following idea: propose a general benchmark for assessing complex blockchain systems fostering accurate measurement free from unbalances caused by unwanted overheads or bottlenecks. Specifically, PERSECUS adopts a systematic benchmarking procedure that fosters optimised deployment of the blockchain under test with tuned parameters, optimised workload generation, and efficient data collection techniques. Finally, we evaluate PERSECUS by measuring the security, performance, and scalability of two Ethereum blockchain platforms, namely Parity and GoQuorum. The experiment showed that regardless of the consensus protocol, the configuration of parameters and the transactions serialisation component of both platforms may cause security and performance issues.

In conclusion, in this thesis, we proposed an assessment of blockchain technology, in particular, focussing on its security, dependability, and performance aspects. We provide a comprehensive suite of tools to lay the foundation for future research studies and to support the implementation of secure, efficient, and reliable decentralised applications.

Chapter 8

Future Directions

Blockchain adoption is growing fast. Many sectors are developing pioneering use cases relying on different blockchain systems, ranging from permissioned private architectures, to open permissionless networks. For instance, some interesting use cases adopt blockchain for building trustworthy data infrastructures in the context of fragmented public and military organisations or investigate the integration between Internet-of-Things (IoT) and blockchain systems to enhance privacy and security of data devices' data without relying on Cloud[22] However, the massive adoption of blockchain has under the hood pitfalls. The increase of complex applications introduces more and more computation on the blockchain and generates growingly workload volumes. Consequently, today's networks are increasingly overloaded resulting in undesired congestions that cause performance issues or even outages (e.g., Ethereum transactions fees growing exponentially according to the network congestion). In addition, the increased complexity of applications implies the development of more complex smart contracts that consequently expose a larger attack surface introducing new and undiscovered vulnerabilities.

To overcome these pitfalls, we propose as a continuation of this work, to investigate and apply the concept of *elastic computing* to the blockchain. Elasticity [91] is a well-established concept in Cloud Computing, and it defines the ability of a system to adapt its resources to the workload variations by autonomously provisioning or under-provisioning resources in a way that the system guarantees adequate performance according to the load. Similarly to a Cloud infrastructure, a blockchain may incur performance and security issues at the workload variation. In particular, the underlying consensus protocol and the node's configuration parameters are crucial characteristics. We showed that a misconfigured node or the wrong consensus protocol could compromise the system's behaviour. To this extent, we aim to refine the concept of elasticity

[22]My contributions in this direction are (i) NATO Security Policy Document - blockchain and IoT position paper (visibility restricted), a baseline document for the official NATO 2019 policy on IoT and blockchain systems, and (ii) BlockIT [122], a blockchain-based infrastructure for reliable and cost-effective IoT-aided smart grids.

for blockchains. Specifically, we plan to propose a *Blockchain Elastic Framework* able to configure the system in an autonomic manner given a predefined set of requirements and therefore adjust its configuration according to the workload variation.

In this work, we learned that any claim of '*superior*' blockchain should be rejected. Conversely, we believe that the future Internet will be composed of an ecosystem of different blockchains each one with different security and performance trade-offs. The proposed Blockchain Elastic Framework aims to enhance the usability of such an ecosystem of blockchains, offering a solution that smoothly shifts from one system to another and autonomously adapts itself according to the application needs. In particular, the main objectives of this framework are:

1. configure an ad-hoc blockchain system, e.g. type of network, blockchain platform used, consensus protocol; according to specific application's performance and security requirements; it might also be possible to adopt different systems for a single application;
2. monitor the workload and at runtime autoscale the number of nodes in either private blockchain networks, or in public blockchain's infrastructures, and provide an elastic resources provisioning to each node controlled by the system;
3. create configuration *profiles* of the adopted blockchain(s), and autonomously tune the client node's parameters according to those profiles and the workload;
4. monitor the state of the adopted blockchain(s), and create alerts that indicate network congestions or ongoing attacks;
5. embed an autonomous decision support system (DSS) that applies machine learning rules to smoothly migrate from one blockchain to another if certain conditions are satisfied, e.g., congested network, malicious attacks, network updates, etc.;

To meet these objectives, we plan to integrate an extended version of PERSECUS within the framework that will work as a source of experimental data to be used for (i) building blockchain *profiles*, and (ii) training the DDS for understanding how systems behave and take decisions. To achieve this goal we need to obtain trustworthy data on systems' performance and security, therefore we need to improve PERSECUS with additional functionalities. Nowadays, PERSECUS reflects systems' performance in presence of simple transactions, but to simulate realistic scenarios we will need to extend the PERSECUS WGM by implementing more complex workload traces based on smart contracts. Despite the security assessment that PERSECUS provides, we will need to evaluate blockchain specific attack patterns for both permissioned and permissionless deployment scenarios, and in each context simulate adverse scenarios ranging from smart contract attacks to misbehaving nodes or groups of nodes.

"You keep on learning and learning,
and pretty soon you learn something
no one has learned before."
Richard Feynman

Bibliography

[1] *A disastrous vulnerability found in smart contracts of BeautyChain (BEC)*. https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e.

[2] *Algorand*. https://www.algorand.com.

[3] *Algorand - Java SDK*. https://github.com/algorand/java-algorand-sdk.

[4] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN: 0805301771.

[5] Raquel Almeida et al. 'How to Advance TPC Benchmarks with Dependability Aspects'. In: *Proceedings of the Second TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*. TPCTC'10. Singapore: Springer-Verlag, 2010, pp. 57–72. ISBN: 9783642182051.

[6] Yackolley Amoussou-Guenou et al. 'Correctness and Fairness of Tendermint-core Blockchains'. In: *ArXiv* abs/1805.08429 (2018).

[7] Stefano De Angelis. *A post-mortem analysis of the parity multisig hack*. https://medium.com/cybersoton/are-blockchains-really-safe-assessing-the-security-of-consensus-schema-84a838458aa3.

[8] Stefano De Angelis. *Docker image of Ethereum Parity with NetEm*. https://hub.docker.com/r/deanstef/parity.

[9] Stefano De Angelis. *Docker image of Quorum Ethereum with NetEm*. https://hub.docker.com/r/deanstef/quorum.

[10] Stefano De Angelis et al. *Blockchain and cybersecurity: a taxonomic approach*. Workshop. EU Blockchain Observatory, 2019.

[11] Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli. 'A Survey of Attacks on Ethereum Smart Contracts (SoK)'. In: *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Matteo Maffei and Mark Ryan. Vol. 10204. Lecture Notes in Computer Science. Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8.

[12] Algirdas Avizienis et al. 'Basic Concepts and Taxonomy of Dependable and Secure Computing'. In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (Jan. 2004), pp. 11–33.

[13] Algirdas Avizienis et al. 'Fundamental Concepts of Dependability'. In: (Apr. 2001).

[14] Arati Baliga et al. 'Performance Evaluation of the Quorum Blockchain Platform'. In: *CoRR* (2018).

[15] P. Baran. 'On Distributed Communications Networks'. In: *IEEE Transactions on Communications Systems* 12.1 (1964), pp. 1–9. DOI: 10.1109/TCOM.1964.1088883.

[16] Massimo Bartoletti et al. 'Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact'. In: *Future Generation Computer Systems* 102 (2020), pp. 259–277. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.08.014.

[17] Aaron Beitch et al. 'Rain: A Workload Generation Toolkit for Cloud Computing Applications'. In: 2010.

[18] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5.

[19] Eric Biederman and Linux Networx. 'Multiple Instances of the Global Linux Namespaces'. In: (Jan. 2006).

[20] BitFury Group. 'Incentive Mechanisms for Securing the Bitcoin Blockchain'. In: *White Paper* (2015). URL: %5Curl%7Bhttp://bitfury.com/content/5-white-papers-research/bitfury-incentive_mechanisms_for_securing_the_bitcoin_blockchain-1.pdf%7D.

[21] BitFury Group and Jeff Garzik. 'Public versus Private Blockchains Part 1: Permissioned Blockchains'. In: *White Paper* (2015). URL: %5Curl%7Bhttp://bitfury.com/content/5-white-papers-research/public-vs-private-pt1-1.pdf%7D.

[22] BitFury Group and Jeff Garzik. 'Public versus Private Blockchains Part 2: Permissionless Blockchains'. In: *White Paper* (2015). URL: %5Curl%7Bhttp://bitfury.com/content/5-white-papers-research/public-vs-private-pt2-1.pdf%7D.

[23] Manuel Blum, Paul Feldman and Silvio Micali. 'Non-Interactive Zero-Knowledge and Its Applications'. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112. ISBN: 0897912640. DOI: 10.1145/62212.62222.

[24] André B. Bondi. 'Characteristics of Scalability and Their Impact on Performance'. In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000, pp. 195–203. ISBN: 1-58113-195-X.

[25] Joseph Bonneau et al. 'Sok: Research perspectives and challenges for bitcoin and cryptocurrencies'. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 104–121.

[26] Robert L. Braddock, Michael R. Claunch and J. Walter Rainbolt. 'Operational Performance Metrics in a Distributed System. Part II.: Metrics and Interpretation'. In: *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*. SAC '92. Kansas City, Missouri, USA: ACM, 1992, pp. 873–882. ISBN: 0-89791-502-X.

[27] Vitalik Buterin. *A Proof of Stake Design Philosophy*. Medium Blog. URL: `%5Curl%7Bhttps://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51%7D`.

[28] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999. ISBN: 0130137847.

[29] Rajkumar Buyya and Manzur Murshed. 'GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing'. In: *Concurrency and Computation: Practice and Experience* 14 (Nov. 2002).

[30] Federico Caccia. *On Ethereum Performance Evaluation Using PoA*. `https://blog.coinfabrik.com/on-ethereum-performance-evaluation-using-poa/experimental-setup`.

[31] C. Cachin. 'Architecture of the Hyperledger blockchain fabric'. In: *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*. 2016.

[32] Christian Cachin, Klaus Kursawe and Victor Shoup. 'Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography'. In: *J. Cryptol.* 18.3 (July 2005), pp. 219–246. DOI: `10.1007/s00145-005-0318-0`.

[33] Christian Cachin and Marko Vukolic. 'Blockchain Consensus Protocols in the Wild'. In: *CoRR* abs/1707.01873 (2017). URL: `http://arxiv.org/abs/1707.01873`.

[34] Christian Cachin et al. 'Secure and Efficient Asynchronous Broadcast Protocols'. In: *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '01. London, UK, UK: Springer-Verlag, 2001, pp. 524–541.

[35] Rodrigo N. Calheiros et al. 'CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms'. In: *Softw. Pract. Exper.* 41.1 (Jan. 2011), pp. 23–50.

[36] Mark Carson and Darrin Santay. 'NIST Net: A Linux-based Network Emulation Tool'. In: *SIGCOMM Comput. Commun. Rev.* 33.3 (July 2003), pp. 111–126. ISSN: 0146-4833.

[37] Miguel Castro and Barbara Liskov. 'Practical Byzantine Fault Tolerance'. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1.

[38] CertiK. *The State of DeFi Security - 2021*. `https://certik-2.hubspotpagebuilder.com/the-state-of-defi-security-2021`.

[39] Huashan Chen et al. 'A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses'. In: *ACM Comput. Surv.* 53.3 (2020), 67:1–67:43. DOI: `10.1145/3391195`.

[40] Huashan Chen et al. 'A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses'. In: *ACM Comput. Surv.* 53.3 (June 2020). ISSN: 0360-0300. DOI: `10.1145/3391195`.

[41] Ting Chen et al. 'An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks'. In: (2017).

[42] ConsenSys. *GoQuorum - Raft consensus*. `https://consensys.net/docs/goquorum/en/latest/configure-and-manage/configure/consensus-protocols/raft`.

[43] ConsenSys. *GoQuorum. Ethereum-based protocol for private, permissioned networks*. `https://docs.goquorum.consensys.net/en/latest/`.

[44] ConsenSys. *Mythril - Security analysis tool for EVM bytecode*. `https://github.com/ConsenSys/mythril`.

[45] Ethereum Consortium. `https://www.ethereum.org`.

[46] Brian F. Cooper et al. 'Benchmarking Cloud Serving Systems with YCSB'. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0.

[47] Standard Performance Evaluation Corporation. `http://www.spec.org`.

[48] Transaction Processing Performance Council. `http://www.tpc.org`.

[49] F. Cristian et al. 'Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement'. In: *Information and Computation* 118.1 (1995), pp. 158–179.

[50] Frank Dabek et al. 'Towards a Common API for Structured Peer-to-Peer Overlays'. In: *Peer-to-Peer Systems II*. Ed. by M. Frans Kaashoek and Ion Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 33–44. ISBN: 978-3-540-45172-3.

[51] Stefano De Angelis et al. 'PBFT vs Proof-of-Authority: Applying the CAP Theorem to Permissioned Blockchain'. In: *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, February 6th - to - 9th, 2018*. Ed. by Elena Ferrari, Marco Baldi and Roberto Baldoni. Vol. 2058. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[52] Xavier Défago, André Schiper and Péter Urbán. 'Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey'. In: *ACM Comput. Surv.* 36.4 (Dec. 2004), pp. 372–421. DOI: 10.1145/1041680.1041682.

[53] Kevin Delmolino et al. 'Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab'. In: vol. 9604. Feb. 2016, pp. 79–94. ISBN: 978-3-662-53356-7. DOI: 10.1007/978-3-662-53357-4_6.

[54] Roger Dingledine, Nick Mathewson and Paul Syverson. 'Tor: The Second-Generation Onion Router'. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004.

[55] T. T. A. Dinh et al. 'BLOCKBENCH: A Framework for Analyzing Private Blockchains'. In: *SIGMOD*. ACM. 2017, pp. 1085–1100.

[56] *Docker*. https://www.docker.com.

[57] Pumba - Chaos Testing for Docker. https://alexei-led.github.io/post/pumba_docker_chaos_testing/.

[58] *Docker Compose*. https://docs.docker.com/compose/.

[59] *Docker SDK for Python*. https://docker-py.readthedocs.io/en/stable/.

[60] Cynthia Dwork, Nancy Lynch and Larry Stockmeyer. 'Consensus in the presence of partial synchrony'. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[61] Parinya Ekparinya, Vincent Gramoli and Guillaume Jourjon. 'The Attack of the Clones Against Proof-of-Authority'. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[62] Ethereum. *EIP 150 - Gas cost changes for IO-heavy operations*. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md.

[63] Ethereum. *Solidity*. https://soliditylang.org.

[64] Ethereum. *Vision of Ethereum2*. https://ethereum.org/en/upgrades/vision/.

[65] Ethereum community. *Ethereum Proof-of-Stake*. https://eth.wiki/en/concepts/proof-of-stake-faqs.

[66] *Ethereum Web3 - Java SDK*. https://github.com/web3j/web3j/.

[67] *Ethereum Web3 - Python SDK*. https://web3py.readthedocs.io/en/stable/.

[68] Etherscan. *Ethereum Average Gas Limit Chart*. `https://etherscan.io/chart/gaslimit/`.

[69] Ittay Eyal and Emin Gün Sirer. 'Majority is not enough: Bitcoin mining is vulnerable'. In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454.

[70] Michael J Fischer, Nancy A Lynch and Michael S Paterson. 'Impossibility of distributed consensus with one faulty process'. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[71] Paul J. Fortier and Howard Michel. *Computer Systems Performance Evaluation and Prediction*. Newton, MA, USA: Butterworth-Heinemann, 2002. ISBN: 1555582605.

[72] Ian Foster and Carl Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-475-8.

[73] Hyperledger Foundation. *Caliper Benchmark Framework*. `https://hyperledger.github.io/caliper/`.

[74] Nissim Francez. *Fairness*. Berlin, Heidelberg: Springer-Verlag, 1986. ISBN: 0-387-96235-2.

[75] Edoardo Gaetani et al. 'Blockchain-based Database to Ensure Data Integrity in Cloud Computing Environments'. In: *ITA-SEC*. Vol. 1816. CEUR-WS.org, 2017.

[76] Juan Garay, Aggelos Kiayias and Nikos Leonardos. 'The Bitcoin Backbone Protocol: Analysis and Applications'. In: *Advances in Cryptology - EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 281–310. ISBN: 978-3-662-46803-6.

[77] Faban Workload Generator. `http://faban.org`.

[78] Ethereum Geth. *Ethereum Private Network*. `https://geth.ethereum.org/docs/interface/private-network`.

[79] *geth - Geth Ethereum command line interface*. `https://geth.ethereum.org/docs/interface/command-line-options`.

[80] Yossi Gilad et al. 'Algorand: Scaling Byzantine Agreements for Cryptocurrencies'. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP17. Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68. ISBN: 9781450350853. DOI: `10.1145/3132747.3132757`.

[81] Seth Gilbert and Nancy Lynch. 'Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services'. In: *SIGACT News* 33.2 (June 2002), pp. 51–59.

[82] *Go Ethereum - Geth*. `https://geth.ethereum.org`.

[83] *goal - Algorand command line interface*. https://developer.algorand.org/docs/clis/goal/goal/.

[84] H-Store. *Smallbank benchmark*. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[85] Distributed Hacking. *An In-Depth Look at the Parity Multisig Bug*. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[86] Distributed Hacking. *The DAO Attack*. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[87] Vassos Hadzilacos and Sam Toueg. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Tech. rep. Ithaca, NY, USA, 1994.

[88] Yue Hao et al. 'Performance Analysis of Consensus Algorithm in Private Blockchain'. In: June 2018, pp. 280–285. DOI: 10.1109/IVS.2018.8500557.

[89] Stephen Hemminger. 'Network emulation with NetEm'. In: *Linux Conf Au* (May 2005).

[90] Ryan Henry, Amir Herzberg and Aniket Kate. 'Blockchain Access Privacy: Challenges and Directions'. In: *IEEE Security Privacy* 16.4 (2018), pp. 38–45. DOI: 10.1109/MSP.2018.3111245.

[91] Nikolas Roman Herbst, Samuel Kounev and Ralf Reussner. 'Elasticity in Cloud Computing: What It Is, and What It Is Not'. In: *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 23–27. ISBN: 978-1-931971-02-7.

[92] Maurice Herlihy and Mark Moir. 'Enhancing Accountability and Trust in Distributed Ledgers'. In: *CoRR* abs/1606.07490 (2016). arXiv: 1606.07490. URL: http://arxiv.org/abs/1606.07490.

[93] Bruce Herndon et al. *VMmark: A Scalable Benchmark for Virtualized Systems*. 2006.

[94] Garrick Hileman and Michel Rauchs. *2017 Global Blockchain Benchmarking Study*. SSRN, 2017.

[95] *Hyperledger Besu*. https://www.hyperledger.org/use/besu.

[96] *Hyperledger Fabric*. https://www.hyperledger.org/use/fabric.

[97] Apache JMeter. https://jmeter.apache.org.

[98] Don Johnson and Alfred Menezes. *The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Tech. rep. 1999.

[99] Sukrit Kalra et al. 'ZEUS: Analyzing Safety of Smart Contracts'. In: *NDSS*. 2018.

[100] Niclas Kannengießer et al. 'Trade-offs between Distributed Ledger Technology Characteristics'. In: *ACM Comput. Surv.* 53.2 (May 2020). ISSN: 0360-0300. DOI: 10.1145/3379463. URL: https://doi.org/10.1145/3379463.

[101] Ghassan O. Karame et al. 'Misbehavior in Bitcoin: A Study of Double-Spending and Accountability'. In: *ACM Trans. Inf. Syst. Secur.* 18.1 (May 2015), 2:1–2:32. ISSN: 1094-9224. DOI: 10.1145/2732196. URL: http://doi.acm.org/10.1145/2732196.

[102] Lucianna Kiffer, Rajmohan Rajaraman and Abhi Shelat. 'A Better Method to Analyze Blockchain Consistency'. In: *Proceedings of the 2018 SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, Oct 15-19*. Ed. by David Lie et al. ACM, 2018, pp. 729–744.

[103] Robin Konrad and Stephen Pinto. 'Bitcoin UTXO Lifespan Prediction'. In: 2015.

[104] Lyudmila Kovalchuk et al. 'Number of Confirmation Blocks for Bitcoin and GHOST Consensus Protocols on Networks with Delayed Message Delivery: Extended Abstract'. In: *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. New York, NY, USA: Association for Computing Machinery, 2018, 42?47.

[105] Andreas Krueger. *Chainhammer Ethereum Benchmarking*. https://github.com/drandreaskrueger/chainhammer. 2018.

[106] *Kubernetes*. https://kubernetes.io.

[107] Murat Kuzlu et al. 'Performance Analysis of a Hyperledger Fabric Blockchain Framework: Throughput, Latency and Scalability'. In: *2019 IEEE International Conference on Blockchain (Blockchain)*. 2019, pp. 536–540. DOI: 10.1109/Blockchain.2019.00003.

[108] Jae Kwon. 'Tendermint : Consensus without Mining'. In: 2014.

[109] L. Lamport. 'Proving the Correctness of Multiprocess Programs'. In: *IEEE Transactions on Software Engineering* SE-3.2 (1977), pp. 125–143.

[110] Leslie Lamport. 'Paxos Made Simple, Fast, and Byzantine'. In: *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*. Ed. by Alain Bui and Hacène Fouchal. Vol. 3. Studia Informatica Universalis. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.

[111] Leslie Lamport. 'The Implementation of Reliable Distributed Multiprocess Systems'. In: *Computer Networks* 2 (Aug. 1978), pp. 95–114.

[112] Leslie Lamport. 'The Part-time Parliament'. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229.

[113] Leslie Lamport. 'Time, Clocks, and the Ordering of Events in a Distributed System'. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565.

[114] Leslie Lamport. 'Using Time Instead of Timeout for Fault-Tolerant Distributed Systems.' In: *ACM Trans. Program. Lang. Syst.* 6.2 (Apr. 1984), pp. 254–280. DOI: 10.1145/2993.2994.

[115] Leslie Lamport, Robert Shostak and Marshall Pease. 'The Byzantine generals problem'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[116] Fatima Leal, Adriana Chis and Horacio Gonzalez-Velez. 'Performance Evaluation of Private Ethereum Networks'. In: *SN Computer Science* 1 (Aug. 2020). DOI: 10.1007/s42979-020-00289-7.

[117] Yu-Te Lin. *Istanbul Byzantine Fault Tolerance*. https://github.com/ethereum/EIPs/issues/650. EIP 650.

[118] *Linux Containers LXC*. https://linuxcontainers.org.

[119] *Linux KVM*. https://www.linux-kvm.org.

[120] *Linux VServer*. http://linux-vserver.org.

[121] Federico Lombardi. 'Autoscaling Techniques and Blockchain-based Architectures for Performant and Dependable Complex Distributed Systems'. PhD thesis. University of Rome La Sapienza, 2017.

[122] Federico Lombardi et al. 'A blockchain-based infrastructure for reliable and cost-effective IoT-aided smart grids'. In: *Living in the Internet of Things Conference: Cybersecurity of the IoT - A PETRAS, IoTUK & IET Event (27/03/18 - 28/03/18)*. Jan. 2018.

[123] R. Lübke et al. 'Measuring accuracy and performance of network emulators'. In: *2014 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. May 2014, pp. 63–65.

[124] Loi Luu et al. 'Making Smart Contracts Smarter'. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 254–269. ISBN: 9781450341394. DOI: 10.1145/2976749.2978309.

[125] Adrian Manning. *Solidity Security: Comprehensive list of known attack vectors and common anti-patterns*. https://github.com/sigp/solidity-security-blog.

[126] Marco Mazzoni, Antonio Corradi and Vincenzo Di Nicola. 'Performance evaluation of permissioned blockchains for financial applications: The ConsenSys Quorum case study'. In: *Blockchain: Research and Applications* (2021), p. 100026. ISSN: 2096-7209. DOI: https://doi.org/10.1016/j.bcra.2021.100026.

[127] Peter M. Mell and Timothy Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.

[128] Paul Menage. *Control groups definition, implementation details, examples and api*. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. 2004.

[129] Daniel A. Menascé. 'TPC-W: A Benchmark for E-Commerce'. In: *IEEE Internet Computing* 6.3 (May 2002), pp. 83–87.

[130] Alexander Mense and Markus Flatscher. 'Security Vulnerabilities in Ethereum Smart Contracts'. In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*. WAS2018. Yogyakarta, Indonesia: Association for Computing Machinery, 2018, pp. 375–380. ISBN: 9781450364799. DOI: 10.1145/3282373.3282419.

[131] Ralph C. Merkle. 'A Digital Signature Based on a Conventional Encryption Function'. In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[132] S. Micali, M. Rabin and S. Vadhan. 'Verifiable random functions'. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 120–130. DOI: 10.1109/SFFCS.1999.814584.

[133] Silvio Micali. *Algorand Co-Chains*. https://www.algorand.com/resources/blog/algorand-co-chains.

[134] D. Mingxiao et al. 'A Review on Consensus Algorithm of Blockchain'. In: ().

[135] Henrique Moniz. *The Istanbul BFT Consensus Algorithm*. 2020. eprint: 2002.03613.

[136] David Mosberger and Tai Jin. 'Httperf&Mdash;a Tool for Measuring Web Server Performance'. In: *SIGMETRICS Perform. Eval. Rev.* 26.3 (Dec. 1998), pp. 31–37. ISSN: 0163-5999.

[137] Malte Möser. *Anonymity of Bitcoin Transactions An Analysis of Mixing Services*. 2013.

[138] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[139] *Netflix Chaos Monkey*. https://github.com/Netflix/chaosmonkey.

[140] Lucas Nussbaum and Olivier Richard. 'A Comparative Study of Network Link Emulators'. In: *Proceedings of the 2009 Spring Simulation Multiconference*. SpringSim '09. San Diego, California: Society for Computer Simulation International, 2009, 85:1–85:8.

[141] Nxt community. *Nxt Whitepaper*. 2014. URL: %5Curl%7Bhttps://nxtwiki.org/wiki/Whitepaper:Nxt%7D.

[142] Brian M. Oki and Barbara H. Liskov. 'Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems'. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '88. Toronto, Ontario, Canada: ACM, 1988, pp. 8–17. ISBN: 0-89791-277-2. DOI: 10.1145/62546.62549.

[143] *OpenVZ*. https://openvz.org.

[144] OpenZeppelin. *Solidity SafeMath Library*. https://docs.openzeppelin.com/.

[145] *Optimisation of Ethereum Parity*. https://github.com/openethereum/openethereum/issues/10382. Answer from Parity developer Tomasz Drwi ga.

[146] Andy Oram, ed. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001. ISBN: 059600110X.

[147] *Pandas*. https://pandas.pydata.org.

[148] Parity. *AuRa - Authority Round consensus protocol*. https://wiki.parity.io/Aura.

[149] Parity. *Proof of Authority Chains*. https://openethereum.github.io/Proof-of-Authority-Chains.

[150] Parity. *Template of a Validator Contract in Parity*. https://openethereum.github.io/Validator-Set.

[151] *Parity Ethereum. A Blockchain Infrastructure for Decentralised Web*. https://www.parity.io. 2018.

[152] Rafael Pass, Lior Seeman and Abhi Shelat. 'Analysis of the Blockchain Protocol in Asynchronous Networks'. In: *Advances in Cryptology - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, Apr 30 - May 4*. Vol. 10211. Lecture Notes in Computer Science. 2017, pp. 643–673.

[153] M. Pease, R. Shostak and L. Lamport. 'Reaching Agreement in the Presence of Faults'. In: *J. ACM* 27.2 (Apr. 1980), pp. 228–234. DOI: 10.1145/322186.322188.

[154] *Peercoin*. https://peercoin.net.

[155] *Principles of Chaos Engineering*. https://principlesofchaos.org/?lang=ENcontent.

[156] Clique - Ethereum Proof-of-Authority consensus protocol. https://github.com/ethereum/EIPs/issues/225.

[157] Francois Raab. 'System Under Test'. In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Y. Zomaya. Cham: Springer International Publishing, 2019, pp. 1663–1665. ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_124.

[158] HariGovind V. Ramasamy and Christian Cachin. 'Parsimonious Asynchronous Byzantine-fault-tolerant Atomic Broadcast'. In: *Proceedings of the 9th International Conference on Principles of Distributed Systems*. OPODIS'05. Pisa, Italy: Springer-Verlag, 2006, pp. 88–102. DOI: 10.1007/11795490_9.

[159] RANDAO. *Random Number Generator*. https://github.com/randao/randao.

[160] N. Regola and J. Ducom. 'Recommendations for Virtualization Technologies in High Performance Computing'. In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. Nov. 2010, pp. 409–416.

[161] Fabricio Reis Furtado et al. 'Towards characterising architecture and performance in blockchain: a survey'. In: *International Journal of Blockchains and Cryptocurrencies* 1 (Jan. 2020), p. 121. DOI: 10.1504/IJBC.2020.109002.

[162] Luigi Rizzo. 'Dummynet: A Simple Approach to the Evaluation of Network Protocols'. In: *SIGCOMM Comput. Commun. Rev.* 27.1 (Jan. 1997), pp. 31–41. ISSN: 0146-4833.

[163] Michael Rodler et al. 'Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks'. In: *CoRR* abs/1812.05934 (2018). eprint: 1812.05934.

[164] Dimitri Saingre, Thomas Ledoux and Jean-Marc Menaud. 'BCTMark: a Framework for Benchmarking Blockchain Technologies'. In: *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*. 2020, pp. 1–8. DOI: 10.1109/AICCSA50499.2020.9316536.

[165] Kazue Sako. 'Digital Signature Schemes'. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 343–344. DOI: 10.1007/978-1-4419-5906-5_17.

[166] Roberto Saltini. 'Correctness Analysis of IBFT'. In: *CoRR* abs/1901.07160 (2019).

[167] Roberto Saltini. 'IBFT Liveness Analysis'. In: *International Conference on Blockchain, Blockchain 2019, Atlanta, Jul 14-17*. IEEE, 2019, pp. 245–252.

[168] Noama Fatima Samreen and Manar H. Alalfi. 'A Survey of Security Vulnerabilities in Ethereum Smart Contracts'. In: *CoRR* abs/2105.06974 (2021). URL: https://arxiv.org/abs/2105.06974.

[169] L. S. Sankar, M. Sindhu and M. Sethumadhavan. 'Survey of consensus protocols on blockchain applications'. In: *ICACCS*. IEEE. 2017, pp. 1–5.

[170] Markus Schäffer, Monika di Angelo and Gernot Salzer. 'Performance and Scalability of Private Ethereum Blockchains'. In: *Business Process Management: Blockchain and Central and Eastern Europe Forum*. Ed. by Claudio Di Ciccio et al. Springer International Publishing, 2019, pp. 103–118. ISBN: 978-3-030-30429-4.

[171] Fred B. Schneider. 'Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial'. In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. DOI: 10.1145/98163.98167.

[172] Gary Shapiro, Christopher Natoli and Vincent Gramoli. 'The Performance of Byzantine Fault Tolerant Blockchains'. In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. 2020, pp. 1–8. DOI: 10.1109/NCA51143.2020.9306742.

[173] Bano Shehar et al. 'SoK: Consensus in the Age of Blockchains'. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*. ACM, 2019, pp. 183–198. DOI: 10.1145/3318041.3355458.

[174] Will Sobel et al. *Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0*. 2008.

[175] Yonatan Sompolinsky and Aviv Zohar. 'Secure High-Rate Transaction Processing in Bitcoin'. In: *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*. Ed. by Rainer Böhme and Tatsuaki Okamoto. Vol. 8975. Lecture Notes in Computer Science. Springer, 2015, pp. 507–527. DOI: 10.1007/978-3-662-47854-7_32. URL: https://doi.org/10.1007/978-3-662-47854-7%5C_32.

[176] Derek Sorensen. 'Establishing Standards for Consensus on Blockchains'. In: *Blockchain Second International Conference, Held as Part of the Services Conference Federation, SCF 2019, San Diego, Jun 25-30, Proceedings*. Ed. by James Joshi et al. Vol. 11521. Lecture Notes in Computer Science. Springer, 2019, pp. 18–33.

[177] De Angelis Stefano and Samer Falah. *Fix for "VM in read-only mode" error seen in logs in stress test of clique networks*. https://github.com/ConsenSys/quorum/pull/1076.

[178] Harish Sukhwani et al. 'Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network)'. In: *17th International Symposium on Network Computing and Applications, NCA 2018, Cambridge, Nov 1-3*. IEEE, 2018, pp. 1–8.

[179] RUBiS - Rice University Bidding System. http://rubis.ow2.org.

[180] Parth Thakkar, Senthil Nathan and Balaji Viswanathan. 'Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform'. In: *CoRR* abs/1805.11390 (2018). eprint: 1805.11390.

[181] *The DAO*. https://github.com/blockchainsllc/DAO/tree/v1.0.

[182] *The ERC20 Short Address Attack Explained*. https://vessenes.com/the-erc20-short-address-attack-explained/.

[183] Locust Load Testing Tool. https://locust.io.

[184] Tsung - Distributed load testing tool. http://tsung.erlang-projects.org.

[185] Carmela Troncoso et al. 'Systematizing Decentralization and Privacy: Lessons from 15 Years of Research and Deployments'. In: *Proceedings on Privacy Enhancing Technologies* 2017 (2017), pp. 404–426.

[186] Marco Vieira and Henrique Madeira. 'From Performance to Dependability Benchmarking: A Mandatory Path'. In: vol. 5895. Aug. 2009, pp. 67–83. ISBN: 978-3-642-10423-7. DOI: 10.1007/978-3-642-10424-4_6.

[187] *VMware*. https://www.vmware.com.

[188] Marko Vukolic. 'Eventually Returning to Strong Consistency.' In: *IEEE Data Eng. Bull.* 39 (2016), pp. 39–44.

[189] Marko Vukolić. 'The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication'. In: *Open Problems in Network Security*. Ed. by Jan Camenisch and Doğan Kesdoğan. Cham: Springer International Publishing, 2016, pp. 112–125. ISBN: 978-3-319-39028-4.

[190] *Vyper smart-sontracts Programming Language*. https://vyper.readthedocs.io/en/stable/.

[191] J. P. Walters et al. 'A Comparison of Virtualization Technologies for HPC'. In: *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*. Mar. 2008, pp. 861–868.

[192] Ingo Weber et al. 'On availability for blockchain-based systems'. In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2017, pp. 64–73.

[193] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 9780080519562.

[194] Gavin Wood. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum Project Yellow Paper* (2014).

[195] *Xen*. https://xenproject.org.

[196] Yang Xiao et al. 'A Survey of Distributed Consensus Protocols for Blockchain Networks'. In: *IEEE Commun. Surv. Tutorials* 22.2 (2020), pp. 1432–1465.

[197] Xiwei Xu et al. 'A Taxonomy of Blockchain-Based Systems for Architecture Design'. In: *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*. IEEE Computer Society, 2017, pp. 243–252. DOI: 10.1109/ICSA.2017.33. URL: https://doi.org/10.1109/ICSA.2017.33.

[198] Renlord Yang et al. 'Empirically Analyzing Ethereum's Gas Mechanism'. In: *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 310–319. DOI: 10.1109/EuroSPW.2019.00041. URL: https://doi.org/10.1109/EuroSPW.2019.00041.

[199] Peilin Zheng et al. 'A detailed and real-time performance monitoring framework for blockchain systems'. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Frances Paulisch and Jan Bosch. ACM, 2018, pp. 134–143. DOI: 10.1145/3183519.3183546.

[200] Xiaoying Zheng, Yongxin Zhu and Xueming Si. 'A Survey on Challenges and Progresses in Blockchain Technologies: A Performance and Security Perspective'. In: *Applied Sciences* 9.22 (2019). ISSN: 2076-3417. DOI: 10.3390/app9224731.