UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science
Cyber Security Group

Enhancing Privacy and Scalability of Permissioned Blockchain

by

Gilberto Zanfino

MSc in Engineering in Computer Science

ORCID iD: 0000-0002-5576-3246

A thesis for the degree of
Doctor of Philosophy

June 2024

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

Enhancing Privacy and Scalability of Permissioned Blockchain

by Gilberto Zanfino

Blockchain is an emerging technology that offers fascinating properties of data integrity and non-repudiation. As the name suggests, it consists of consecutive chained blocks, replicated on all network nodes, containing asset transactions. Blocks are linked together via hashing procedures and created in a distributed fashion with consensus protocols. Besides, blockchain provides smart contracts, self-executable programs that allow to realise fully decentralised and tamper-proof applications. In this thesis, we analyse five prominent blockchain platforms, i.e. Bitcoin, Ethereum 2.0, Algorand, Ethereum-private and Hyperledger Fabric. We evaluate their security according to the used consensus algorithm, the overall infrastructure and the smart contracts vulnerabilities. We then focus on permissioned blockchains, operated by authenticated parties. Although they can enforce access control rules and are more efficient compared to permissionless, they still lack of data *privacy* and present *scalability* issues. A typical privacy solution is to encrypt data before being stored on blockchain, but the downside is that smart contracts can no longer execute functions on them. We thus propose to combine blockchain with Homomorphic Encryption (HE), a cryptographic model that allows to perform computations on ciphertexts. We show how blockchain coupled with HE can be beneficially applied to Smart Grid to realise privacy-preserving energy billing and trading. HE has a limitation, however: it precludes computations on data encrypted with different keys. To overcome this, we extend the previous integration with Multi-Key HE (MKHE). We thus present PANTHER, a MKHE-integrated permissioned blockchain, where users can run smart contracts over ciphertexts created with different keys. Results of MKHE computations are then decrypted in PANTHER via secure multiparty protocols among users. For scalability instead, blockchain exhibits an intrinsic problem: adding new nodes to cope with increasing demands worsens performance, due to a lengthening of the time to reach consensus. We thus present SHERLOCK, a permissioned blockchain in which consensus nodes are split into committees and disposed on a two-layer ring-based architecture. SHERLOCK uses the sharding technique to assign incoming transactions to committees, which process them in parallel, thereby boosting performance.

Contents

List of Figures

List of Tables

List of Protocols

List of Accompanying Material

Dataset https://doi.org/10.5258/SOTON/D3096

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:
    - Stefano De Angelis, Gilberto Zanfino, Leonardo Aniello, Federico Lombardi, and Vladimiro Sassone. Blockchain and cybersecurity: a taxonomic approach. *EU Blockchain Observatory and Forum*, 2019.
    - Stefano De Angelis, Gilberto Zanfino, Leonardo Aniello, Federico Lombardi, and Vladimiro Sassone. Evaluating blockchain systems: A comprehensive study of security and dependability attributes. In *Proceedings of the 4th Workshop on Distributed Ledger Technology co-located with the Italian Conference on Cybersecurity (ITASEC)*, volume 3166 of *CEUR Workshop Proceedings*, pages 18–32, 2022.

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

Acknowledgements

I really wish to thank my family with all my heart, especially my dad, my mom, my grandparents and Ludovica for your loving and constant support, your advise, and for always helping me to choose the right way. A special thanks goes to Vladimiro for giving me the opportunity to undertake this PhD and for introducing me to the world of research, and to Leonardo and Federico for guiding me along this journey. Thank you for being always available and your friendship.

To my grandmother Angelina

Abbreviations

CA	Certificate Authority
DER	Distributed Energy Resources
FHE	Fully Homomorphic Encryption
HE	Homomorphic Encryption
IoT	Internet of Things
LWE	Learning with Errors
MKFHE	Multi-Key Fully Homomorphic Encryption
MKHE	Multi-Key Homomorphic Encryption
MPC	Secure Multiparty Computation
PBFT	Practical Byzantine Fault Tolerance
PHE	Partial Homomorphic Encryption
PoA	Proof of Authority
PoS	Proof of Stake
PoW	Proof of Work
RES	Renewable Energy Sources
RLWE	Ring Learning With Errors
TFHE	Threshold Fully Homomorphic Encryption
THE	Threshold Homomorphic Encryption
VPN	Virtual Private Network
VRF	Verifiable Random Function
WSN	Wireless Sensor Network

Chapter 1

Introduction

The technology landscape has recently evolved into an environment of an increasing number of Internet-connected devices, which interact with existing online systems for a wide variety of purposes. They range from entertainment platforms, such as video streaming, music libraries, instant messaging, e-commerce and e-games, to business platforms like cloud infrastructure, tools for managing development and analysing data. This has led to an intensifying demand for services, resources and processing tasks requested by the end-users. Consequently, the need arises for these online systems to be built so as to guarantee security, fault tolerance, and high standards in terms of availability and performance. *Blockchain* is one of the most attractive and emerging technologies on the IT market that promotes delivering these attributes to a system. Firstly employed as a public ledger within Bitcoin's cryptocurrency (Nakamoto et al. (2008)), nowadays blockchain technology is considered as a paradigm for multi-party systems of organisations and enterprises. Besides the well-known, news-making, iconic applications to cryptocurrency, blockchain is breathing new life into subjects such as distributed and decentralised computing, as well as data assurance. In such fields, the impact of blockchain is showcased by its unmatched potential to transform '*work*' into '*trust*', which can change the face of computing in untrusted environments.

Indeed, blockchain is a peer-to-peer distributed network of nodes that share a replicated data structure without a central authority. The shared data structure, also called the *ledger*, consists of chained blocks, each one linked to the previous through referencing its hash digest. Each block contains a list of records that witness transactions occurred among participants. Such transactions represent an exchange of assets between a network's nodes (e.g., exchange of crypto-coins in the Bitcoin blockchain). The process of block creation is called *mining*, and is carried out by special nodes of the network called *miners*, which periodically activate a distributed *consensus* algorithm to select new blocks to be added to the chain. Once consensus is

achieved, all the nodes update their local copy of the blockchain and agree on the same state.

The blockchain's linked data structure allows all the nodes in the network to have the same view of the ledger. This means that they agree on all the previous blocks and on their relative order, which leads to high consistency of the information shared on the blockchain and its history. Indeed, even if only one record of a past block were modified or deleted on a single node, its hash digest would differ, and the following block would keep pointing to the original, unmodified block. In other words, in order to modify a block effectively, a node would need to also modify all subsequent blocks, each with the new hash of the precedent block, till the end of the chain. Furthermore, for these changes to be reflected on other nodes' local copies of the blockchain, a global agreement would be required. Hence, unless all nodes agree on all such changes, the consistency of the ledger would be unmistakably compromised. This is indeed a crucial feature of blockchain, which ensures *integrity* and *non-repudiation* of the shared data, and therefore gives rise to a trusted system that does not rely on any centralised trusted third-party authority. Due to these security and data assurance guarantees, blockchain pushed its boundaries beyond the realm of cryptocurrencies and became a subject of study in several other fields, notably including supply chain, provenance, IoT and other sectors where multiple parties want to share a computing/storage infrastructure. Indeed, modern blockchains have been endowed with nearly Turing-powerful programming languages that can be used to implement, deploy and execute fully decentralised and tamper-resistant applications, called *smart contracts*.

Notwithstanding its promising advantages, blockchain raises issues of security, scalability and performance that limit its deployment in real systems.
Blockchain-based systems being distributed over multiple nodes, they typically present a broad attack surface, as each one of the participants can come under attack. A crucial aspect of such distributed system is their chosen consensus algorithm, i.e., the protocol that in each blockchain the parties employ to agree on the occurrence and ordering of transactions. State-of-the-art consensus algorithms provide two sets of security properties, dubbed respectively *safety* and *liveness*. Safety states that the consensus algorithm should not do anything wrong during its normal execution: when an honest node accepts a transaction, then all the other honest nodes will make the same decision. If safety is violated at some point in time, it will never be satisfied again after that time. Liveness, instead, states that eventually something good happens: there is a time at which all honest nodes agree on the same order of transactions. Together they are used to ensure that the system protocols keep making progress towards an end, i.e. to verify and order transactions. Consensus algorithms 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. (Cachin and Vukolić (2017); Vukolić (2015); Xu et al.

(2017); Shehar et al. (2019)), 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 trade-off between all desired properties.

Against this backdrop, the study and understanding of the security of blockchain becomes paramount, both in terms of the properties afforded by the technology and in terms of its vulnerabilities. As a starting point of this thesis, we propose a comprehensive evaluation of blockchain's security. We refer to the well-established concepts of security and dependability, broadly used in distributed systems, to identify the fundamental security properties relevant to a blockchain. Thus, we use such properties to evaluate five of the most prominent blockchain platforms. In particular, we study three *permissionless* blockchains, namely Bitcoin, Ethereum 2.0, Algorand, and two *permissioned* blockchains, namely Ethereum-private and Hyperledger Fabric. We conduct a study of their security over three dimensions, i.e. the consensus protocols, infrastructure, and smart contracts.

Afterwards, as the main thematic of this thesis, we focus on permissioned blockchains where users and network nodes are authenticated, and authorised to execute operations only with proper permissions. Permissioned blockchains are a good fit in scenarios that require data authorisations and better performance. Indeed, the fact that participating nodes are authenticated allows to enforce access control policies and use more efficient consensus algorithms. Despite this, permissioned blockchain present two problematic aspect: the lack of data *privacy* and the lower *scalability* of network nodes. Indeed, on one hand, the inherent data transparency of blockchain does not sit well with situations where data providers are not keen to share their data with others. Although permissioned blockchains enforce access control rules to data, some privileged users of the system are still authorised to read others' data, or a large part thereof. Solutions based on storing encrypted data in the ledger have the limitation that smart contract computations cannot be carried out on these data. From the other hand, although permissioned blockchains are more efficient than permissionless ones, they still present bottlenecks when the magnitude of users' transactions is very large. To cope with a growing demand, in standard distributed systems it is possible to add nodes to the network and spread the workload among them. Conversely, in permissioned blockchains the nodes operate on the same bunch of transactions by collectively participating in consensus protocol. As a consequence, the addition of new nodes causes more message exchanges and a longer time to reach agreement on transactions ordering, which with cascading effect worsens performance.

Therefore, we investigate in this thesis how to enhance permissioned blockchains under the two directions of privacy and scalability. In the following, we detail their

respective problems and the research questions they raise. Then, we briefly describe the ideas we propose to address them.

Blockchain privacy

The data integrity and non-repudiation are not the sole properties the blockchain possesses. Blockchain additionally provides data transparency: being the ledger replicated on all network nodes, the transactions it contains are visible to all participants. This effect is amplified in permissionless blockchains, where any node can participate in the network without restrictions. Although a total transparency is a desirable feature for many public systems, there are several scenarios in which users prefer to keep their data confidential. Permissioned blockchains deal with this privacy requirement by establishing access rules on data. Users can see that a transaction has occurred in the system, but they can access its content only with proper permissions. However, permissioned settings only offer a trade-off between transparency and privacy. High privilege users with high-ranking permissions (e.g., organization admins) are legitimately authorised to read all data owned by low privilege users. One viable solution to keep users' data confidential, before they are permanently stored in the ledger, is to encrypt them. However, this approach opens up a new challenge. Once users' data have been encrypted in the ledger, blockchain can no longer perform any kind of computation on them via smart contracts. This restricts blockchain functioning to data storage only, and inhibits its deployment in scenarios requiring computations. Consider, for instance, a decentralised auction system based on blockchain, where users want to keep their bids secret. Then, each user encrypts her bid and submit it to the blockchain network. The only way for blockchain nodes to compute the winning bid is to decrypt users' bids, thus unsealing the bids and violating privacy. Hence, we ask ourselves the following research question:

How can we enable blockchain to perform computations on encrypted data?

To answer such question we investigate the exploitation of the *Homomorphic Encryption (HE)* model. Particularly, HE is a cryptographic-based computing model that allows computations to be performed on encrypted data without having to decrypt them first. This fascinating property can empower smart contracts to compute functions over ciphertexts stored in the ledger, and thus the blockchain system to realise privacy-preserving applications. The result of a HE-based computation is also encrypted, and can be securely stored in the blockchain ledger. Only the owner of the cryptographic key can later decrypt it to discover the function outcome. We thus propose in this thesis to integrate HE into a permissioned blockchain, creating an architecture whereby users can encrypt data and still avail of smart contract functions.

In order to demonstrate the benefits of this integration, as use case, we apply it to a cutting-edge Smart Grid system driven by IoT-devices. Besides improving decentralisation and system management, we show that the HE-equipped permissioned blockchain simultaneously gives Smart Grid the means (i) to guarantee the integrity and privacy of users' data (ii) and to carry out privacy-preserving operations. We implement this system in Hyperledger Fabric, enforcing authentication and authorisation policies to participants, i.e. both users and blockchain nodes. Considering the aforementioned advantages, we then propose a generalisation of this solution combining blockchain and HE to other IoT-based scenarios.

Notwithstanding the HE integration permits the blockchain to perform computations on ciphertexts, it comes with an intrinsic constraint. The homomorphic property of HE ensures that the result of a function executed on ciphertexts, once decrypted, matches the result of the function executed on the corresponding plaintexts. In traditional HE schemes this property only holds if the ciphertexts are encrypted under the same key, reason why are typically dubbed *single-key*. Indeed, in our previous solution for Smart Grid, we provide each system user with a single-key scheme, by which she encrypts her personal data. Although the smart contracts are able to execute per-user functions, they are not able to execute functions involving data from multiple users, as they are encrypted with different keys. A feasible workaround could be to employ one single-key HE scheme for the entire system, share the encryption key to all users, and leave the matching decryption key in the hands of a system administrator. However, with this workaround users just feed the system with their data and are not independent: even if they can autonomously call a smart contract with input ciphertexts, they will not be able to decrypt its result as they lack the decryption key. To overcome all these limitations we extend the previous HE-based solution with *Multi-Key HE (MKHE)*, a recent HE model wherein the homomorphic property is also valid for ciphertexts encrypted with distinct and unrelated keys. We thus propose PANTHER, a permissioned blockchain integrated with MKHE, in which each user can encrypt data with her own key and smart contracts computations are enabled on ciphertexts of different users.

Blockchain scalability

The scalability is the ability of a system to maintain an adequate level of performance in the face of a growing workload. There are two methods that can be applied to make a system scalable, i.e. *scale up* or *scale out*. The former consists of expanding the resources of a node, whereas the latter consists of adding nodes to the network. Scaling up can help a blockchain node to increase its ledger size (by expanding disk space) and to run smart contracts faster (by upgrading CPU), but does not help it to

process more transactions. This is because the transaction processing is driven by the consensus protocol, and both the processing time and the number of transactions processed depend on the consensus implementation rather than on the resource power of the participating nodes. Indeed, in permissioned settings, the consensus protocol is based on a communication-bound voting mechanism, where the performance of the protocol depends on the number of message exchanged. By contrast, scaling out a permissioned blockchain means increasing the number of nodes participating in the consensus, thus a higher message overhead and a longer time to reach agreement on transactions. As a result, the scale out method worsens performance instead of improving it. This scalability problem restricts the types and dimensions of a system where a permissioned blockchain can be employed, hindering its application in scenarios composed by a huge volume of users. Hence, we ask ourselves the following research question:

How can we improve blockchain performance by adding nodes to its network?

To answer such question we investigate the exploitation of the *sharding* technique. Particularly, sharding spreads users' transactions across network nodes, in a way that each of them holds a shard of the overall data, thus distributing and balancing the workload. In this thesis, we present SHERLOCK, a permissioned blockchain characterised by a two-layer ring-based architecture for nodes engaged in the consensus protocol. The first layer divides the nodes in committees, such that each committee is an instance of consensus protocol. This enables blockchain to parallelise the processing of transactions, i.e. transactions execution, ordering, and blocks creation. SHERLOCK then uses hash-based sharding to evenly spread the transactions across committees, which concurrently and in isolation create provisional blocks. Periodically, the committees in first layer send their generated blocks to the second layer, composed by a single committee of nodes responsible for finally providing a total order to received blocks. Adding new nodes to SHERLOCK means forming new committees and augmenting the parallelisation, towards a performance improvement.

Contributions

The contributions of this thesis can be summarised as follows:

- we analyse five prominent blockchain platforms and we evaluate their security according to the consensus protocol in place, their infrastructure and their smart contracts vulnerabilities (De Angelis et al. (2019, 2022));
- we propose a novel permissioned blockchain architecture integrated with the HE model, which provides data privacy while enabling smart contracts to carry out functions on ciphertexts;

- we extend the previous solution by proposing PANTHER, a novel permissioned blockchain integrated with the MKHE model, which further allows to perform computations on data encrypted under different keys and is secure against byzantine adversaries;
- we propose SHERLOCK, a novel permissioned blockchain featured by a two-layer ring-based architecture and sharding technique, which enhances scalability, transactions processing and performance.

Thesis structure

In Chapter 2 we first outline the context, by introducing the blockchain technology, the HE and the SMC models. We describe how they are composed, their functionalities, and the properties they provide. Then, in Chapter 3 we study the security of five different blockchain platforms, i.e. Bitcoin, Ethereum 2.0, Algorand, Ethereum-private and Hyperledger Fabric. We define the security and dependability properties of their different consensus protocols and infrastructure, along with an identification of smart contract issues that can affect them. According to this taxonomy we evaluate the security of examined blockchain platforms. In Chapter 4 we begin our research on enhancing blockchain privacy. We propose a novel blockchain architecture integrated with HE, capable of performing privacy-preserving functions via smart contracts. We apply it to a Smart Grid system to privately calculate energy bills and manage energy trading auctions. In Chapter 5 we present PANTHER, a permissioned blockchain integrated with MKHE and MPC, which extends the previous solution by enabling homomorphic computations on data encrypted under multiple unrelated keys. We detail PANTHER's protocols and security guarantees. In Chapter 6 we shift our research focus on enhancing blockchain scalability. We present SHERLOCK, a permissioned blockchain that uses sharding to allocate transactions across committees of consensus nodes. We detail how these committees in SHERLOCK parallelise the consensus process and improve the system performance. Finally, Chapter 7 sums up the thesis and discusses future research directions.

Chapter 2

Background and context

A distributed system is a set of autonomous and independent computational entities that work together in a way to appear as a single coherent system to the end-user. These entities, computers or software processes, coordinate their actions by exchanging messages over a communication network towards achieving a common goal, e.g. solve a computational problem, share resources, provide services. In this context, the network size may vary from a handful to millions of devices, which can be geographically dispersed and linked with wired and/or wireless connections. Compared to centralised systems, where there is one single central entity that serves all the other network nodes, a distributed system offers a better reliability and avoids performance bottlenecks. This because there is not a single point of failure and the system can add new nodes to cope with a growing number of end-user requests. Data availability can be enhanced by replicating the data among multiple nodes, as well as the system performance can be boosted by parallelising the tasks execution. Indeed, data replication guarantees an always-on system response even in the presence of faulty nodes, whereas parallel computation helps to smooth out and mitigate the workload. The expected objectives of distributed systems are:

- *availability*: provide users with constant access to the system's remote resources and services;
- *scalability*: increase the number of system nodes and/or their hardware specifications when the number of users' requests grows;
- *end-user transparency*: hide the whole system infrastructure and internal functional mechanisms from the end-user viewpoint;
- *extendability*: openness of a system to be extended or re-implemented.

Fulfil these objectives while building up a distributed system is, however, challenging. Some occurring pitfalls are related to the unreliability and heterogeneity of the

network, and that its topology may change at runtime. Furthermore, some malicious parties, whether internal or external to the network, can attack the system to obtain confidential information or disrupt the offered services. If proper protocols are not put in place these factors undermine the security and stability of the system, and consequently the users' confidence in using it. There are two main properties for a distributed system that relate to the security of users' data:

- *data confidentiality*: prevent malicious parties from acquiring sensitive information. This can be achieved through various degrees of privacy. A system can enforce authentication and access control policies, i.e. data can be accessed only with proper authorisations. A system can enforce cryptographic techniques, data can accessed only by the owner of the decryption key;
- *data integrity*: prevent malicious parties from altering data. A system can enforce verification processes based on cryptography to check whether data have been tampered with.

These two properties together with data availability form the so-called *CIA triad* of data security, typically used as a standard to assess the system vulnerabilities.

Examples of distributed systems cover various application fields:

- *data storage*: Blockchain, Cassandra (Lakshman and Malik (2010)), Amazon Dynamo (DeCandia et al. (2007));
- *stream processing*: Apache Storm (Apache Software Foundation (2011b)), IBM Spade (Gedik et al. (2008));
- *cryptographic computing*: Secure Multiparty Computation, Homomorphic Encryption
- *message brokering*: RabbitMQ (Mozilla (2010)), Apache Kafka (Apache Software Foundation (2011a));
- *file sharing*: BitTorrent (Cohen (2003)).

These implementations differ in form, purposes, and particularly in features they offer. For instance, blockchain provides trust among network nodes but present scalability issues when new nodes are added. On the contrary, distributed stream processing systems are able to scale but, since generally they are distributed over nodes within a single organisation domain, they simply assume trust.

In this thesis, we study the blockchain technology by analysing its manifold constructions and the different properties they provide. In Chapter 3 we describe five

different blockchains platforms and we evaluate their security. Following this evaluation, we further investigate permissioned blockchains and we lead our research in finding solutions to improve their privacy and scalability. To this aim, in Chapter 4 and Chapter 5 we present solutions to empower the blockchain with privacy-preserving computations, and in Chapter 6 we present a solution to scale out the blockchain nodes boosting performance. For enabling private computations, we investigate the combination of blockchain with Homomorphic Encryption (HE) and Secure Multiparty Computation (MPC) distributed computing models. Beyond providing privacy to blockchain data, these integrations vice-versa provide strong integrity to encrypted data generated and exchanged in HE and MPC models. For enabling scalability instead, we design a new architecture for blockchain nodes based on the sharding technique.

Thus, in this chapter we provide a succinct introduction to the blockchain, which will be expanded and deepened in Chapter 3, and we describe in detail HE and MPC, which will be applied to permissioned blockchain in Chapter 4 and Chapter 5.

Chapter structure

In Section 2.1 we first describe the blockchain technology, its cryptographic data structure, its distributed consensus protocol and the different settings it provides, i.e. permissionless and permissioned. Then, we describe two cryptographic-based distributed computing models, namely the MPC and HE. In Section 2.2 we describe MPC, detailing the secret sharing and secure computation techniques in Section 2.2.1 and Section 2.2.2 respectively, along with the MPC security in Section 2.2.3. In Section 2.3 we describe HE, detailing its properties (Section 2.3.1) and its MPC-based variants, i.e. Threshold HE (Section 2.3.3) and Multi-Key HE (Section 2.3.4). Also, Section 2.3.2 presents CKKS, a fully HE scheme performing arithmetic with approximate numbers.

2.1 Blockchain

Blockchain is a novel technology that has appeared on the market in recent years and has gained lots of attention in different applications, ranging from money and energy trading to supply chain and healthcare management. It was firstly used as a public ledger for the Bitcoin cryptocurrency (Nakamoto et al. (2008)) and consists of consecutive chained blocks containing transactions, that are replicated and stored by nodes of a peer-to-peer network. These transactions, occurred between some nodes of the network, can represent a cryptocurrency (e.g., the Bitcoin) or other kinds of assets.

Figure 2.1 shows a piece of the chain where each block is a container that aggregates transactions. Additionally, each block of the chain keeps track of its predecessor by using hashing procedures. At block creation, a hashing function takes in input the content of previous block and generates in output a one-off value, the hash, which makes previous block uniquely identifiable (Rogaway and Shrimpton (2004)). Before appending a block with the newly processed transactions to the chain, the hash of the previous block is stored inside it. As side-effect, any tampering attempt of a block's content will result in a change of its associated hash, which will make it different from the hash stored in following block.

The blocks are created in a distributed fashion by means of a general agreement among network nodes, without relying on a central trusted authority (e.g., banks or financial institutions). Specifically, when transactions occur on a blockchain, they are first validated and then included in a block by some distinguished nodes of the network. In Bitcoin these nodes are called *miners* and their block construction method is performed by a computational intensive hashing task, called Proof-of-Work (PoW), which defines a global order on transactions. This global ordering is known as *consensus*, and only once the network majority reaches it then the newly generated block is appended to blockchain. Different kinds of consensus algorithms with different properties have been developed and utilised within blockchain (Mingxiao et al. (2017)), like Proof-of-Stake (PoS), Proof-of-Authority (PoA) or Practical Byzantine Fault Tolerance (PBFT). The consensus process, together with the full data replication on a large number of nodes, allow blockchain to enjoy strong properties related to data integrity. Indeed, when a block is part of the chain, all the miners have agreed on its content and, hence, it is non-repudiable and persistent. The tampering with a block requires that its entire following chain must be modified, which is extremely difficult according to the consensus protocol in place. This because consensus protocols require a majority to create a block, which is a majority of hash power for PoW, a majority of stake for PoS and 2/3 of honest nodes for PBFT.

Differently from Bitcoin, new types of blockchain such as Ethereum (Wood et al. (2014)) have emerged introducing *smart contracts*, which are self-executable programs running across blockchain network. Smart contracts are more versatile and complicated than simple currency transactions. Being able to express conditions, constraints and business logic, they perfectly encompass and model the terms of a contract among entities to exchange assets (e.g., services or products). Smart contracts executions become traceable, verifiable and irreversible within blockchain ecosystem, thus enforcing the business logic contained therein and the agreement among involved parties. Furthermore, smart contracts permit creating the so-called *decentralised applications*, i.e. applications that operate autonomously and without any control by a system entity and whose logic is immutably stored on blockchain. For

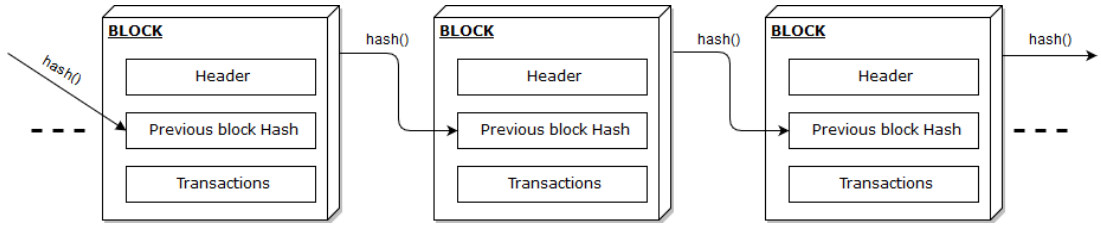


FIGURE 2.1: Blockchain data structure

instance, a smart contract can carry out a vote, an auction, calculate a bill, or it can even implement data mining and machine learning algorithms.

Bitcoin and Ethereum belong to a group of blockchain systems called *permissionless*, in which any node on Internet can join the blockchain network, write to the shared ledger by invoking transactions, and participate in the consensus process to form new blocks. Contrarily to this setting, other blockchain systems exist that offer an authentication and authorization layer, such as R3 Corda (Brown et al. (2016)) and Hyperledger Fabric (Androulaki et al. (2018)). They are known as *permissioned* blockchains and are operated by known entities. They can be either members of a consortium across multiple organizations or belong to one organization only. In permissioned blockchains the nodes are identified and enabled to issue transactions or to be involved in consensus process only with proper permissions. This allows to employ consensus schemas lighter than the PoW, like PoA or PBFT, making permissioned blockchains faster by design (Cachin and Vukolić (2017)). A comparison between the two settings is proposed by Swanson (2015). From one hand, permissionless blockchains exhibit better information transparency and auditability, but sacrifice privacy and mostly performance. On the other, permissioned blockchain show higher transactions processing rate and flexibility in changing and optimising the network rules, thus resulting in cost reduction.

2.2 Secure Multiparty Computation

In distributed systems context, and particularly in the field of distributed computing, one of the main objectives for a cluster of servers is to carry out some function on data they hold (e.g., data mining, elections, auctions). Considering the case in which these data are confidential, the first requirement is ensuring that nothing else about the data is leaked except the outcome of a pre-agreed function. *Secure Multiparty Computation (MPC)* is a sub-field of cryptography able to provide this functionality ensuring strong privacy guarantees on data handled. This is made possible by concealing the data being processed and also the whole elaboration logic. MPC has been a subject of academic studies since the 1980s, when the first constructions were proposed (Yao (1982)), but nevertheless a first attempt to use it for a real scenario only took place in

early 2000s (Bogetoft et al. (2009)). Thereafter, many MPC platforms have been developed with different type of MPC protocols available.

Specifically, MPC is a distributed computing model that allows a set of multiple untrusted parties to jointly perform functions on their private data, in such a way that each of them learns the result value of a chosen function but not each other's inputs data. MPC enables such privacy-preserving property by means of cryptographic protocols whose building blocks are:

- *Secret sharing of data*: Each party shares only a fragment of its data to the other parties involved;
- *Computation on data fragments*: Each party individually computes the function taking in input the fragments received from other parties involved.

At the end, all involved parties can discover the final result value of a chosen function by jointly merging the outputs coming from each party's computation on fragments.

2.2.1 Secret sharing

Secret sharing is a MPC technique introduced by Shamir Shamir (1979) and Blakley Blakley (1979) in 1979, where a secret value (e.g., a decryption key) is split in fragments, called *shares*, indistinguishable from random values. One share is delivered to one of the participating parties, in a way that none of them knows the secret and they can reconstruct it (only) by joining the shares they hold. For instance, assuming a secret S given by a sequence of bits, and N parties where $n = |N|$, we can construct a protocol to sharing S with N using XOR operation (i.e., $\oplus$). At first, generate $n-1$ random fragments (a.k.a. nonces) $s_1, \ldots, s_{n-1}$ of the same length of S. Then, set:

$$s_n = S \oplus s_1 \oplus \cdots \oplus s_{n-1}$$

and send the ith fragment s_i to the ith party in N. Hence, it follows that S can be reconstructed only by xoring all the shares, namely $S = s_1 \oplus \cdots \oplus s_n$. Note that this protocol is information-theoretically secure (i.e., it cannot be cryptographically broken even if an adversary has an infinite computational power) because: (i) even if the adversary knows $n-1$ shares it does not know enough information to reconstruct S, and (ii) actually knowing $n-1$ shares does not give more information than knowing one share.

Based on this preliminary method, Shamir proposes in Shamir (1979) a scheme called *(k, n)-threshold secret sharing* so defined:

Definition 2.1 (Shamir Secret Sharing)**.** Given a secret S and a pair (k, n), where the threshold k is $k > 1$ and $k \leq n$, find n shares $s_1, \ldots, s_n$ such that:

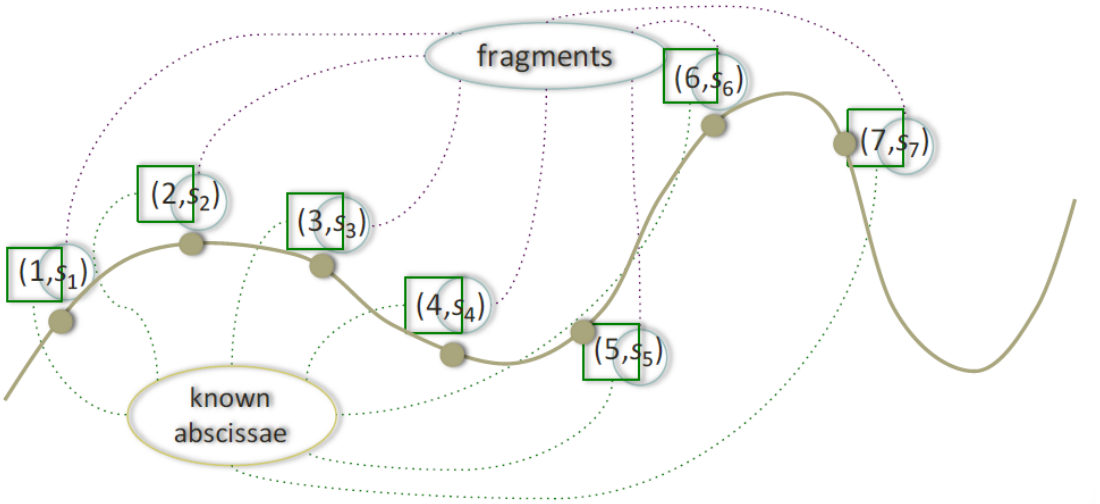


FIGURE 2.2: Secret fragments over the Cartesian coordinate system

- Knowledge of k or more shares makes it possible to reconstruct S;
- Knowledge of $k-1$ or fewer shares leaves S completely undetermined, in the sense that all its possible values are equally likely.

Note that in the case (n, n) all n shares are required to reconstruct the secret. To create this scheme Shamir leverages on the polynomial interpolation over finite fields Cramer et al. (2015). Specifically, it selects a prime number p bigger than both S and n, and creates a polynomial of the form

$$h(x) = a_{k-1}x^{k-1} + \cdots + a_1x^1 + a_0 \bmod p$$

where a_0 is S and coefficients $a_{k-1}, \ldots, a_1$ are picked at random over finite field $\mathbb{F}_p$. Then, the ith share s_i is the tuple of points $(i, h(i))$. An example of the interpolation polynomial on the Cartesian coordinate system is depicted in Figure 2.2, where the abscissae and fragments can be identified.

Using the Lagrange formula Cramer et al. (2015) is possible to reconstruct S with any combination of k shares.

Definition 2.2 (Lagrange formula)**.** Given a set of $k+1$ data points

$$(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$$

where no two x_j are the same, the interpolation polynomial in the Lagrange form is a linear combination

$$L(x) = \sum_{j=0}^{k} y_j l_j(x)$$

of Lagrange basis polynomials

$$l_j(x) = \prod_{0 \leqslant m < k}^{m \neq j} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_{j-k})}$$

Some of the useful properties of this (k, n) threshold secret sharing scheme are:

- The size of each fragment is not bigger than size of the secret $(|s_i|, |S| < |p|)$;
- If k is kept fixed, fragments can be dynamically added/deleted without affecting the other fragments;
- It is straightforward to generate a new set of fragments randomly building a new polynomial;
- It is possible to assign higher weights to involved parties by giving them more than one fragment.
- Each fragment can be delivered to one of the MPC computing nodes and then, as proposed in the *BGW* protocol (Ben-Or et al. (1988)), the sum operation can be calculated locally by performing sums on the collected fragments, while products and other functions can be calculated through interactions between the nodes.

2.2.2 Secure computation

The core of MPC model is represented by the *secure computation* protocol, in which involved parties individually compute over shares created with a secret sharing protocol. The term *individually* in this process means that each party locally and in parallel perform the same computation of the others but taking in input different shares. This because the secret sharing protocol splits data in shares and each party has one of them. The tasks range from simple mathematical operations, such as addition, multiplication or comparison, to more complex functions like searching a common keyword in a collection of confidential documents. Similar to secret sharing protocols, a secure computation protocol leverages on modular arithmetic to handle the shares without knowing the actual value they represent.

In order to make a proper distinction, the MPC model separates the actors according to the role they play. During a secret sharing protocol they are identified as *input parties*, whereas in secure computation protocol they are called *computational parties*. Another role is played by the *result parties*, which are those that request a desired function to be executed on the data of input parties. In particular, following the same logic of Section 2.2.1 for reconstructing the original value from shares, the result

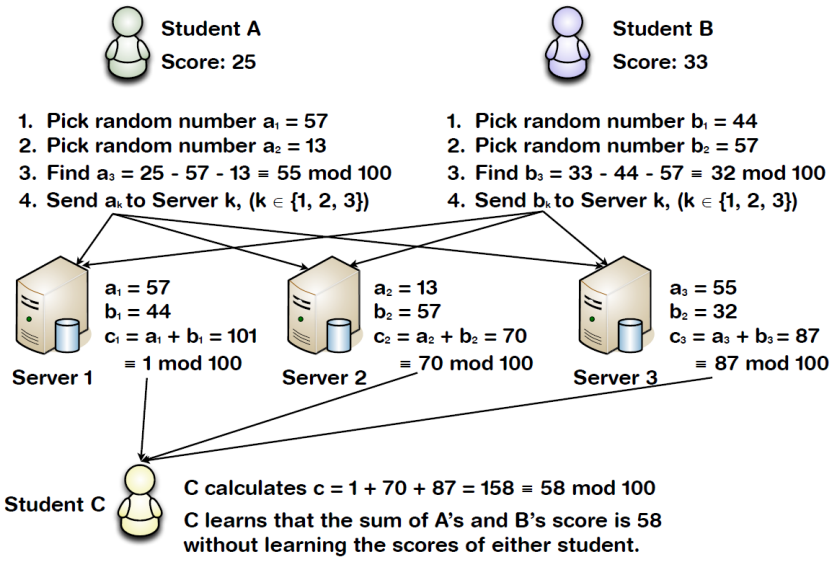


FIGURE 2.3: A MPC addition performed by three computational parties

parties merge the outputs of computational parties to locally reconstruct the outcome of pre-agreed function. Generally these groups are disjoint in a system employing the MPC model, although it can happen that they overlap and a party plays multiple roles, e.g. an input party can be a computational party. The separation of roles helps the MPC model to be more efficient, keeping the number of computing parties small and reducing the number of shares to be produced by input parties.

In a system where the MPC is in place, the typical working flow starts with each input party creating n shares for each of its confidential data (e.g., medical records, financial data or asset transactions) through secret sharing protocol, where n is the number of computational parties. Each share is delivered to exactly one party belonging to computational group. When a system user among the result parties requests to perform a desired function, then each computational party individually executes a secure computation protocol on shares it received from various input parties. Once the computation is completed, each computational party obtains a transient result, indistinguishable from a random value. Finally, the requesting result party collects all transient results produced by all computational parties and merge them together to reveal the outcome of the desired function.

An example of how this process works is shown in Figure 2.3, where a student queries the University system, composed by three servers, to learn the sum of other students scores. The MPC model is employed within such system to preserve data privacy of students (i.e., their scores), but at same time to allow statistical analysis on their data (e.g., average of scores, etc.). The students ”A” and ”B” represent input parties, the three servers represent computational parties, and the student ”C” represents a result party. The input parties execute a secret sharing protocol for splitting their scores in shares. Specifically, each of them picks two random numbers and creates three shares,

one for each computational party, using addition in modular arithmetic. Once received shares from input parties, each computational party fulfils result party's request by individually executing an addition-based secure computation protocol on shares it holds. The outputs deriving from secure computation protocols of servers are delivered to the result party to be later reconstructed. The student C finally learns the sum of A's and B's scores, while they both remain private. Regarding the overall system security, for the sake of completeness it is notable that no server knows neither the score of each student, nor the final sum.

2.2.3 Security of MPC-based protocols

As can be noted from previous sections, the combined usage of secret sharing and secure computation protocols of MPC model enables a system to store data in an encrypted form and subsequently to process them without decrypting. Such MPC-based protocols are conducted so that certain security properties, like privacy and correctness, are preserved. Specifically:

- *Privacy*: The only thing that a party can learn it is the output of the prescribed function, thereby no party should learn other parties inputs. For example in an election the winning candidate will be revealed with the relative total number of votes taken, but not who precisely voted for her;
- *Correctness*: The output of the prescribed function will always be correct. In the election example, this means that the candidate with the highest number of votes is guaranteed to win, and no party can influence this;

These properties should be ensured even if some of the participating parties or an external entity maliciously attack the protocol, i.e. the so-called *corrupted* parties. A corrupted party is passive (a.k.a. *honest-but-curious*) when it follows the protocol but tries to deduce secret information, whereas is active (a.k.a. *byzantine*) when it arbitrarily deviates from protocol. Generally speaking, privacy and correctness belong to a set of requirements that should be met and held for any secure protocol, and particularly for MPC they are:

- *Independence of inputs*: Corrupted parties should not be able to choose the inputs to be submitted according to those of the honest parties. Continuing with election example, a set of cheating parties cannot submit their votes knowing those of the honest ones, because this would allow them to subvert the election result;
- *Guaranteed output delivery*: Corrupted parties cannot prevent honest parties from receiving the function's output. Thus, in example, the adversary cannot

intercept and/or destroy the output in transit towards an honest party, nor hinder the computation;

- *Fairness*: Corrupted parties will receive function's output if and only if the honest parties also did. Thus, in example, election's result has to be received by all participating parties, regardless they wish to attack the protocol or not, otherwise the corrupted ones can gain crucial information (i.e., the winning candidate).

If these requirements are fulfilled the inputs of honest parties remain private and the corrupted parties cannot cause the computation result to deviate from the function determined by honest parties.

Goldreich et al. (1987) establish some feasibility results about MPC protocols, demonstrating that any distributed function can be securely compute in the presence of active adversary. Let n denote the number of parties participating in MPC protocols and let t denote those that may be corrupted:

1. For $t < n/3$, MPC protocols with fairness and guaranteed output delivery can be achieved for any function under computational security assuming a synchronous point-to-point network with authenticated channels, and under information-theoretic security assuming the channels are also private.
2. For $t < n/2$, MPC protocols with fairness and guaranteed output delivery can be achieved for any function under both computational and information-theoretic security, assuming that the parties also have access to a broadcast channel.
3. For $t \geq n/2$, MPC protocols without fairness or guaranteed output delivery can be achieved.

2.3 Homomorphic Encryption

In 1978, the early days of the asymmetric cryptography breakthrough, Rivest et al. (1978) suggested an innovative paradigm under the name "privacy homomorphisms," which introduces the ability to perform various algebraic functions on encrypted data without having to decrypt them first. This paradigm relies on an interesting *homomorphic property* exhibited by some public-key schemes, whereby the encrypted result of a function executed on ciphertexts, once decrypted, matches the result of the same function executed on corresponding plaintexts. In other words, a function $\mathcal{F}$ applied on two ciphertexts $c_1 = Enc(m_1)$ and $c_2 = Enc(m_2)$, resulting from the encryption of two plaintexts m_1 and m_2, produces an encrypted output c that decrypted is equal to the function applied on plaintexts,

$Dec(c) = \mathcal{F}(m_1, m_2)$. In particular, Rivest et al. (1978) observed that the RSA encryption scheme presents this homomorphic property for multiplication of encrypted values. Since then, many cryptographers have devised several encryption schemes that support computations on ciphertexts. Featured by the homomorphic property such novel schemes take the name of *Homomorphic Encryption (HE)* schemes.

Formally, a HE scheme is defined as follow, where the plaintext space is $\mathcal{M} \in \{0, 1\}$, and the set of algebraic functions it can carry out are represented in the form of boolean circuits $\mathcal{C}$ (Vollmer (1999))

Definition 2.3 (*$\mathcal{C}$-Homomorphic Encryption*)**.** For a class of circuits $\mathcal{C}$, a $\mathcal{C}$-HE scheme is a tuple of algorithms $\mathcal{E} = (\text{KEYGEN}, \text{ENC}, \text{EVAL}, \text{DEC})$ with the syntax:

- $\text{KEYGEN}(1^\lambda) \to (pk, sk, ek)$: For a security parameter λ, outputs a public key pk, a private key sk, and a (public) evaluation key ek;
- $\text{ENC}(pk, m) \to (c)$: Given a public key pk and a message m, outputs a ciphertext c;
- $\text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t) \to (c)$: Given an evaluation key ek, a (description of a) boolean circuit $\mathcal{F}$ and t ciphertexts $c_1, \ldots, c_t$, outputs a ciphertext c;
- $\text{DEC}(sk, c) \to (m)$: Given a private key sk and a ciphertext c, outputs a message m.

We require that for each kth circuit $\mathcal{F}_k \in \mathcal{C}$, all tuples (pk, sk, ek) in the support of $\text{KEYGEN}(1^\lambda)$ and all plaintexts $(m_1, \ldots, m_t)$ and ciphertexts $(c_1, \ldots, c_t)$ such that c_i is in the support of $\text{ENC}(pk, m_i)$, if $c = \text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t)$, then $\text{DEC}(sk, c) = \mathcal{F}(m_1, \ldots, m_t)$. The computational complexity of $\mathcal{E}$'s algorithms is polynomial in the security parameter λ, and in the case of the evaluation algorithm, the size of the circuit.

We refer to ciphertexts produced by ENC as *fresh ciphertexts*, whereas the ones produced by EVAL as *evaluated ciphertext*. The first generation of $\mathcal{C}$-HE schemes is usually called *Partial HE (PHE)*, since initially the schemes were composed of circuits that allow only one type of algebraic operation, either addition or multiplication, but not both. A PHE scheme is additively homomorphic if its class of circuits $\mathcal{C}$ contains only additions, or alternatively is multiplicative homomorphic if $\mathcal{C}$ contains only multiplications. Addition and multiplication are respectively expressed in boolean circuits as XOR (i.e., $\oplus$) and AND (i.e., $\wedge$) logic gates. Let $c_1 = \text{ENC}(pk, m_1)$ and $c_2 = \text{ENC}(pk, m_2)$ then:

$$\text{DEC}(sk, \text{EVAL}(ek, \oplus, c_1, c_2)) = m_1 + m_2;$$
$$\text{DEC}(sk, \text{EVAL}(ek, \wedge, c_1, c_2)) = m_1 \cdot m_2.$$

Examples of additively PHE schemes are Paillier (Paillier (1999)), DGK (Damgård et al. (2007)) and GM (Goldwasser and Micali (1982)), whereas multiplicative PHE schemes are RSA (Rivest et al. (1978)) and ElGamal (Elgamal (1985)).

Although the PHE schemes have revolutionised the world of cryptography by making it possible to compute on encrypted data, they are limited to support the execution of only one type of operation, i.e. one between addition and multiplication. That is the case until 2009, when the groundbreaking work of Gentry Gentry (2009a) establishes for the first time a blueprint to realise HE schemes capable of evaluating arbitrary types of functions, i.e. both addition and multiplication. This significant improvement goes under the name of *Fully HE (FHE)*. Specifically, a FHE encryption scheme $\mathcal{E}$ is a $\mathcal{C}$-HE for the class $\mathcal{C}$ of all circuits, composed by both $\oplus$ and $\wedge$ logic gates. The work of Gentry was followed by a sequence of rapid advancements (Brakerski et al. (2012); Brakerski and Vaikuntanathan (2011); Gentry et al. (2013); Van Dijk et al. (2010)), which have brought various optimisations to FHE from efficiency and security perspectives.

2.3.1 Properties of HE-based schemes

As defined by Goldwasser and Micali (1982), one essential property for a public-key scheme is *semantic security*, also called *indistinguishability under chosen plaintext attack (IND-CPA)*. Generally, such property is represented in the form of a game between an adversary and a challenger. The adversary is modelled with a *probabilistic polynomial-time (PPT)* Turing machine, meaning that it terminates in a polynomial number of steps. The challenger generates a pair of public-private keys and gives the public key pk to the adversary, which it may use to generate any number of ciphertexts. The adversary sends two equal-length messages m_0 and m_1 to the challenger, which selects one of them to be encrypted by randomly choosing a bit $b \in \{0, 1\}$. Then, the challenger sends back the ciphertext c_b=ENC(pk, m_b) to the adversary, which has to guess whether c_b is the encryption of m_0 or m_1. The adversary sends $b' \in \{0, 1\}$ to the challenger and wins if $b' = b$. The underlying cryptosystem is IND-CPA if the PPT adversary is not able to determine $b' = b$ with a probability significantly greater than $1/2$. This required the ENC algorithm to possess a component of randomness, so that c_b is only one of many valid ciphertexts for m_b. Otherwise, the adversary could simply deterministically encrypt m_0 and m_1, and then compare the resulting ciphertexts with c_b to successfully guess the challenger's choice.

The IND-CPA is a property required also for $\mathcal{C}$-HE schemes. Indeed PHE schemes like RSA, ElGamal, Paillier, DGK and GM are semantically secure. Semantic security of a $\mathcal{C}$-HE scheme follows directly from the semantic security of the underlying (asymmetric) encryption scheme in the presence of the (public) evaluation key ek (Goldwasser and Micali (1984); Halevi (2017)). This is because EVAL is a public

algorithm with no secrets, and it does not give any additional information to the adversary. Formally, we define semantic security for a $\mathcal{C}$-HE scheme as follow:

Definition 2.4 (Semantic Security)**.** A $\mathcal{C}$-HE scheme is semantically secure if for every PPT adversary $\mathcal{A}$, a sufficiently large λ, every equal-length messages $m_0, m_1 \in \mathcal{M}$ (i.e, $|m_0|=|m_1|$), an index bit $b \in \{0,1\}$, and a negligible function ε, then

$$\text{KEYGEN}(1^\lambda) \rightarrow (pk, sk), \tag{2.1}$$

$$\text{ENC}(pk, m_b) \rightarrow (c), \tag{2.2}$$

$$Pr\Big[\mathcal{A}(pk, m_0, m_1, c) = b : (2.1), (2.2)\Big] = \frac{1}{2} + \varepsilon(\lambda).$$

The probability of the adversary $\mathcal{A}$ guessing b, i.e. guessing whether c is the encryption of m_0 or m_1, knowing pk, is $1/2$ plus a negligible amount in λ.

The semantic security belongs to a group of properties that any $\mathcal{C}$-HE scheme should possess. Formally, they are:

Definition 2.5 (*Correctness*)**.** Let $\mathcal{C}$ be some class of circuits. A $\mathcal{C}$-HE scheme $\mathcal{E}$ = (KEYGEN, ENC, DEC, EVAL) is correct for $\mathcal{C}$ if it correctly decrypts ciphertexts generated from both ENC and EVAL algorithms. Namely, for all $\lambda \in \mathbb{N}$, the following two conditions hold:

- For any $m \in \mathcal{M}$,

$$\text{ENC}(pk, m) \rightarrow (c),$$
$$Pr\Big[\text{DEC}(sk, c) = m\Big] = 1;$$

- For every circuit $\mathcal{F} \in \mathcal{C}$ and for any possible combinations of t ciphertexts $c_1,\ldots,c_t$, where $c_i = \text{ENC}(pk, m_i)$,

$$\text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t) \rightarrow (c),$$
$$Pr\Big[\text{DEC}(sk, c) = \mathcal{F}(m_1, \ldots, m_t)\Big] = 1.$$

Definition 2.6 (*Compactness*)**.** A $\mathcal{C}$–HE scheme is compact if the size of evaluated ciphertexts does not grow with the complexity of evaluated circuits, but it only depends on the security parameter λ. Namely, if there exists a polynomial P, such that for all $\lambda \in \mathbb{N}$, for every circuit $\mathcal{F} \in \mathcal{C}$ and for any ciphertext c generated from $\text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t)$, where $c_i = \text{ENC}(pk, m_i)$,

$$Pr\Big[|c| \leq P(\lambda)\Big] = 1.$$

Definition 2.7 (*Circuit Privacy*)**.** A $\mathcal{C}$–HE scheme, correct for the class of circuits $\mathcal{C}$, is circuit private for $\mathcal{C}$, if for every circuit $\mathcal{F} \in \mathcal{C}$, and any possible combination of t

ciphertexts $c_1, \ldots, c_t$, where $c_i = \text{ENC}(pk, m_i)$, the distributions (2.3) and (2.4):

$$\text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t) \tag{2.3}$$

$$\text{ENC}(pk, \mathcal{F}(m_1, \ldots, m_t)) \tag{2.4}$$

taken over the randomness of each algorithm are (statistically) indistinguishable, in symbols $(2.3) \approx (2.4)$.

Note that Definition 2.7, also known as *function privacy*, refers to the indistinguishability between the probability distributions of EVAL and ENC algorithms. Basically, this states that even if two ciphertexts $c' = \text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t)$ and $c'' = \text{ENC}(pk, f(m_1, \ldots, m_t))$ are different in value and in generation process, they belong to the same distribution. The interpretation of such property is that c' can be generated by $\text{ENC}(pk, \mathcal{F}(m_1, \ldots, m_t))$, and vice-versa c'' is a valid output from $\text{EVAL}(ek, \mathcal{F}, c_1, \ldots, c_t)$. Therefore, it is difficult to determine which circuit has been applied resulting in c', since c' can likewise result from the encryption of a particular plaintext $\tilde{m}$ corresponding to $\mathcal{F}(m_1, \ldots, m_t)$. For the correctness property, only when decrypted c' and c'' are equal, i.e., $\text{DEC}(sk, c') = \text{DEC}(sk, c'')$. Hence, a ciphertext generated by EVAL does not reveal anything about the evaluated circuit, beyond its output value.

2.3.2 CKKS: a FHE scheme

The *CKKS* scheme Cheon et al. (2017), also called Homomorphic Encryption for Arithmetic of Approximate Numbers (*HEAAN*), is a FHE scheme that relies as a computational model on polynomial and modular arithmetic over the rings (Shoup (2006)). Unlike many HE schemes that primarily work with integers, CKKS supports computations with real and complex numbers. This makes it particularly well-suited for applications where numerical precision is crucial, such as in machine learning models and data analytics. In particular, CKKS provides a mechanism to encode complex values into polynomials with integer coefficients, and vice-versa decode polynomials back into complex values. Encryption, decryption and evaluation algorithms are then applied to these polynomials. In CKKS, additions and multiplications perform an approximate computation, which means that after being decrypted and decoded the evaluation output is an approximate result. The accuracy of this approximation depends on a scale factor Δ applied during encoding and a rescaling procedure applied during decoding. This rescaling procedure is also used by CKKS in the evaluation algorithm to reduce the magnitude of the evaluated ciphertext following a multiplication. Moreover, CKKS is built upon the *Ring Learning With Errors* (RLWE) mathematical problem Lyubashevsky et al. (2013), and the security of the scheme relies on the hardness of solving it. Indeed, as in the RLWE problem, the

CKKS scheme include a noise during encryption to make the polynomial ciphertext indistinguishable. The main idea behind CKKS is to treat this RLWE noise as part of error occurring during approximate computations. That is, the encryption takes in a message m and a noise e as input, and generates a ciphertext of $m' = m + e$ representing an approximate value of the message. If e is much small compared to m, the noise is not likely to destroy the significant figures of m and the whole value $m' = m + e$ can replace the original message in approximate arithmetic. The combination of this approach with the scaling and rescaling procedures enables CKKS to reduce the precision loss due to noise.

2.3.2.1 Notation

We use regular letters for integers and polynomials, and bold letters for vectors. Let n be a power of two and q be an integer. We denote by $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ the polynomial ring of the $(2n)$-th cyclotomic field and $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ the residue ring of $\mathcal{R}$ modulo q. For an integer q, we use $\mathbb{Z} \cap (-\frac{q}{2}, \frac{q}{2}]$ as a representative of $\mathbb{Z}_q$. We denote $[a]_q$ the reduction of an integer modulo q. For a real number r, we denote by $\lfloor r \rceil$ the nearest integer to r, rounding upwards in case of a tie. When applied to polynomials, these operations are performed coefficient-wise. For a polynomial a in $\mathcal{R}$ or $\mathcal{R}_q$, we denote its infinity norm by $\|a\|_\infty$.

We use $x \leftarrow D$ to denote that x is sampled from the distribution D. For a finite set S, we denote $\mathcal{U}(S)$ as the uniform distribution on S. For $\sigma > 0$, we denote by D_σ a distribution over $\mathcal{R}$ sampling n coefficients independently from the discrete Gaussian distribution of variance σ^2, and B_σ an (overwhelming probability) upper bound of D_σ with respect to the infinity norm.

2.3.2.2 Ring Learning With Errors

RLWE is a mathematical problem in the field of lattice-based cryptography Lyubashevsky et al. (2013). Given the parameters $(n, q, \psi, \mathcal{X}_\sigma)$, consider the polynomial number of samples $(a_i, b_i) \in \mathcal{R}_q^2$, where $a_i \leftarrow \mathcal{U}(\mathcal{R}_q)$, $b_i = s \cdot a_i + e_i \pmod q$ and error $e_i \leftarrow \mathcal{X}_\sigma$ for a fixed private key $s \leftarrow \psi$ from the key distribution ψ. We consider that the private key s has ternary coefficients in $\{-1, 0, 1\}$. The RLWE assumption states that the distribution of RLWE samples (a_i, b_i) is computationally indistinguishable from $\mathcal{U}(\mathcal{R}_q^2)$. That is, in other words, it is computationally hard for an adversary that does not know s and e_i to distinguish between the distribution of $(a_i, s \cdot a_i + e_i)$ and that of (a_i, c_i) where $c_i \leftarrow \mathcal{U}(\mathcal{R}_q)$.

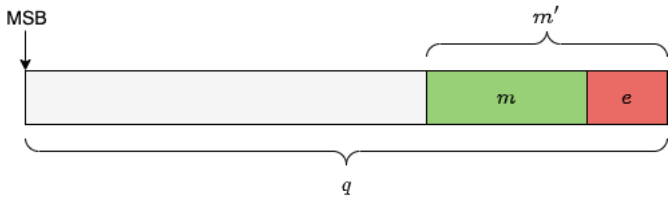


FIGURE 2.4: Ciphertext structure in the CKKS scheme.

2.3.2.3 Plaintext and ciphertext spaces

In CKKS, both the plaintext and ciphertext spaces include elements of the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$, where q is called the coefficient modulus and $X^n + 1$ is an irreducible polynomial called the polynomial modulus. Elements of $\mathcal{R}_q$ are polynomials with integer coefficients bounded by q, i.e. they can take any value between 0 and $q - 1$. The degrees of polynomials in $\mathcal{R}_q$ are bounded by n. In CKKS, plaintexts differ from ciphertexts in the number of polynomials they contain. An instance of plaintext contains one polynomial, whereas an instance of ciphertext contains two polynomials, of which one containing the encrypted message and the other dedicated to decryption.

Figure 2.4 shows the structure of a CKKS ciphertext. For ease of illustration, we simply report the polynomial of the ciphertext containing the message, and that the polynomial is of degree 0 (i.e., is equal to aX^0 where a is the coefficient). In Figure 2.4, MSB stands for the most significant bit, q is the coefficient modulus, $m' = m + e$ is the approximate message with the noise on the LSB (least significant bit). The space is bounded by the value of q, and the approximate message m' can freely move in this space, where the grey-colored area is the open space remaining. Before the message encryption, CKKS scales the message m by the scale factor Δ, moving the most significant bits of m to the left, further away from e. Such scaling procedure ensures that e does not distort or corrupt the value of m, and that m' is a good approximation of m. The reason is that the least significant bits of m, which will be distorted by adding e, are not of great importance since they will be rounded off after rescaling m'.

2.3.2.4 Parameters

The RLWE-based parameters n and q have an impact on both security and performance of the CKKS scheme. Larger n increase the security, but at the same time decrease the performance. For a fixed n, a larger q implies both lower security and lower performance. Hence, n and q should be chosen carefully to suit the intended use of the scheme. The homomorphic encryption standard Albrecht et al. (2018) suggests pairs of (n, q) for different levels of security (i.e., 128, 192 or 256 bits). Specifically, for a given n, it recommends a value of q which will achieve a given level

of security, e.g. 128. Moreover, it should be noted that the CKKS is a scale-variant scheme. When a homomorphic multiplication is performed, the ciphertext is scaled down by Δ. This results in reducing the size of q by Δ and producing a polynomial whose coefficient modulus is $q' = \frac{q}{\Delta}$. Hereafter, we refer to the coefficient modulus at level l by q_l, where $1 \leq l \leq L$, and L is the level of a fresh ciphertext. Therefore, CKKS ciphertext coefficients are related to each other by:

$$q_L > q_{L-1} > \ldots > q_1$$

Beyond the aforementioned parameters q, n, and Δ, CKKS uses the following additional RLWE-specific distributions in its instantiation:

- $\mathcal{R}_3$: is the (private) key distribution that uniformly samples polynomials with integer coefficients in $\{-1, 0, 1\}$.
- $\mathcal{X}_\sigma$: is the error distribution defined as a discrete Gaussian distribution over $\mathcal{R}$ of variance σ^2, bounded by some integer B_σ. According to the current version of the homomorphic encryption standard Albrecht et al. (2018), (σ, B_σ) are set as $(\frac{8}{\sqrt{2\pi}} \approx 3.2, \lfloor 6 \cdot \sigma \rceil = 19)$.
- $\mathcal{U}(\mathcal{R}_q)$: is a uniform random distribution over $\mathcal{R}_q$.

2.3.2.5 Encoding and decoding

The encode algorithm of the CKKS scheme takes in input a $\frac{n}{2}$-vector of complex numbers $\mathbf{z} \in \mathbb{C}^{\frac{n}{2}}$ and outputs a polynomial m from ring $\mathcal{R}$. The CKKS decoding does the reverse, by taking in a polynomial m and returning a vector of complex numbers. The following Equation 2.5 and Equation 2.6 show the encoding and decoding respectively. The map π is the complex canonical embedding which is a variant of the Fourier transform (Cheon et al. (2017)).

$$\text{ECD}(\mathbf{z}, \Delta) = \lfloor \Delta \cdot \pi^{-1}(\mathbf{z}) \rceil = m \tag{2.5}$$

$$\text{DCD}(m, \Delta) = \pi\left(\frac{1}{\Delta} \cdot m\right) = \mathbf{z} \tag{2.6}$$

In the encode algorithm, the scaling is performed by multiplying the message by Δ, and removal of least significant fractional parts is performed via rounding. In the decode algorithm, this procedure is reversed by dividing by Δ.

2.3.2.6 Relinearization and rescaling

As in any FHE scheme, in CKKS the magnitude of an evaluated ciphertext grows after multiplication. Consider two input ciphertexts $c' = (c'_0, c'_1)$ and $c'' = (c''_0, c''_1)$ both in $\mathcal{R}_{q_l}$, and perform multiplication between them:

$$c' \cdot c'' = (c'_0, c'_1) \cdot (c''_0, c''_1) = [c'_0 \cdot c''_0]_{q_l}, [c'_0 \cdot c''_1 + c'_1 \cdot c''_0]_{q_l}, [c'_1 \cdot c''_1]_{q_l} = (\tilde{c}_0, \tilde{c}_1, \tilde{c}_2) = \tilde{c}$$

The ciphertext $\tilde{c}$ contains three polynomials as opposed to input ciphertexts c' and c'' that contain two. CKKS uses the *relinearization* procedure to reduce the size of $\tilde{c}$ to two polynomials. That is, the goal of relinearization is to find a $\bar{c} = (\bar{c}_0, \bar{c}_1)$ such that:

$$\begin{aligned} \textsc{Dec}(\bar{c}, sk) = \bar{c}_0 + \bar{c}_1 \cdot sk &= \textsc{Dec}(\tilde{c}, sk) \\ &= \textsc{Dec}(c', sk) \cdot \textsc{Dec}(c'', sk) \\ &= (c'_0 + c'_1 \cdot sk) \cdot (c''_0 + c''_1 \cdot sk) \\ &= c'_0 \cdot c''_0 + (c'_0 \cdot c''_1 + c'_1 \cdot c''_0) sk + c'_1 \cdot c''_1 \cdot sk^2 \\ &= \tilde{c}_0 + \tilde{c}_1 \cdot sk + \tilde{c}_2 \cdot sk^2 \end{aligned}$$

In order to achieve this equality, the reduced ciphertext needs to be defined as $(\bar{c}_0, \bar{c}_1) = (\tilde{c}_0, \tilde{c}_1) + p$ where p is a pair of polynomials by which:

$$\textsc{Dec}(\bar{c}, sk) = \textsc{Dec}((\tilde{c}_0, \tilde{c}_1), sk) + \textsc{Dec}(p, sk) = \tilde{c}_0 + \tilde{c}_1 \cdot sk + \tilde{c}_2 \cdot sk^2$$

To compute p, the relinearization uses the evaluation key $ek = (b, a) \in \mathcal{R}^2_{P \cdot q_l}$, where $b = [-a \cdot sk + e + P \cdot sk^2]_{P \cdot q_l}$, $a \leftarrow \mathcal{U}(\mathcal{R}_{P \cdot q_l})$, $e \leftarrow \mathcal{X}_\sigma$ and P is a big integer. The idea is to multiply ek by $\tilde{c}_2$ and divide by P to reduce the noise (since $\tilde{c}_2$ is a big polynomial). Hence, $p = \lfloor P^{-1} \cdot \tilde{c}_2 \cdot ek \rceil \pmod{q_l}$ and its decryption is:

$$\textsc{Dec}(p, sk) = \frac{\tilde{c}_2}{P}(-a \cdot sk + e + P \cdot sk^2) + \frac{\tilde{c}_2}{P} \cdot a \cdot sk = \frac{\tilde{c}_2}{P} \cdot e + \tilde{c}_2 \cdot sk^2 \approx \tilde{c}_2 \cdot sk^2$$

After the relinearization, the non-expanded ciphertext $\bar{c} \in \mathcal{R}^2_{q_l}$ encrypts the product of c' and c'' but with a squared scale factor Δ^2. This because the plaintexts m' and m'', encrypted by c' and c'' respectively, are both scaled up by Δ during encoding. Thus, CKKS applies the *rescaling* procedure to $\bar{c}$ that scales it down by Δ and generates an its equivalent ciphertext $\hat{c} \in \mathcal{R}^2_{q_{l-1}}$ with a reduced coefficient modulus q_{l-1}. Equation 2.7 defines the rescaling algorithm:

$$\textsc{Rsc}(\bar{c}, \Delta) = \frac{1}{\Delta} \cdot [\bar{c}]_{q_l} = [\hat{c}]_{q_{l-1}} \tag{2.7}$$

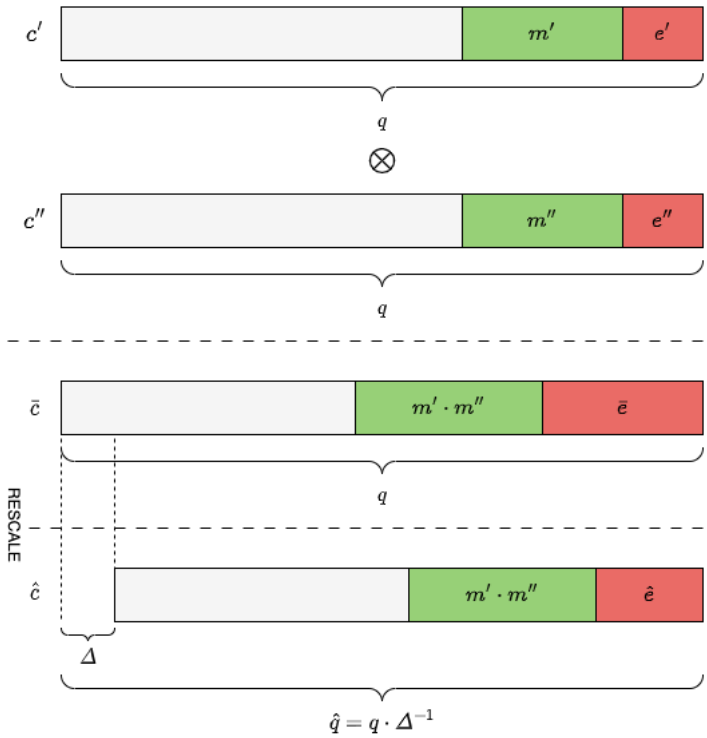


FIGURE 2.5: CKKS multiplication and rescaling.

The rescaled ciphertext $\hat{c}$ encrypts the same plaintext of $\bar{c}$, i.e. $m' \cdot m''$ with a reduce scale factor and reduced noise, which results in a coefficient modular reduction from level l to level $l-1$. Figure 2.5 shows this procedure. For each ciphertext we simply report the polynomial containing the message (i.e., c'_0, c''_0, $\bar{c}_0$, $\hat{c}_0$), and we assume that the polynomial is of degree 0.

2.3.2.7 HE primitives

Figure 2.6 shows an high level view of the CKKS scheme. Firstly, a $\frac{n}{2}$-vector of complex numbers $\mathbf{z} \in \mathbb{C}^{\frac{n}{2}}$ is encoded into a plaintext $m \in \mathcal{R}$, where $\mathcal{R}$ is the ring $\mathbb{Z}[X]/(X^n+1)$. Recall that during the encoding, $\mathbf{z}$ is scaled by a factor Δ. Then, m is encrypted with the public key pk, which is composed by a pair of polynomials (pk_1, pk_2) defined as:

$$pk_1 = [-a \cdot sk + e]_{q_L}$$
$$pk_2 = a \leftarrow \mathcal{U}(\mathcal{R}_{q_L})$$

The polynomial sk is the private key sampled from $\mathcal{R}_3$, a is a random polynomial sampled uniformly from $\mathcal{R}_{q_L}$, and e is a random error polynomial sampled from $\mathcal{X}_\sigma$.

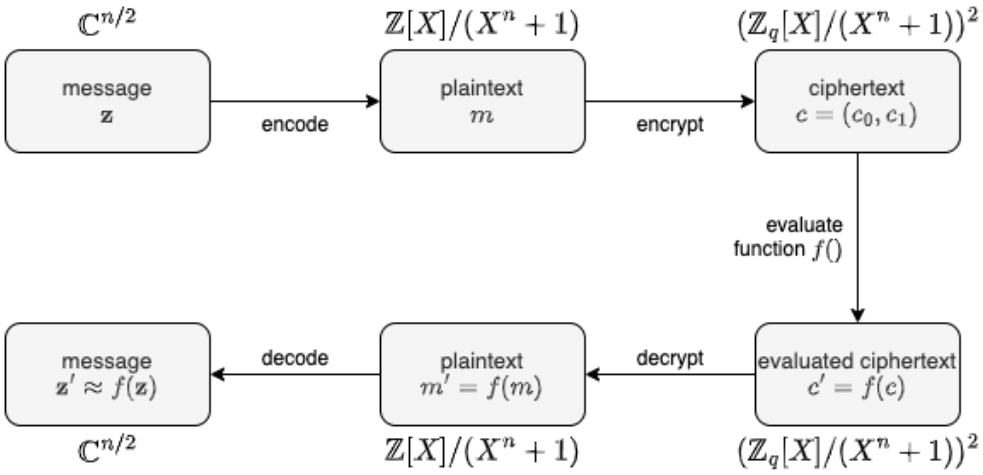


FIGURE 2.6: The CKKS algorithms and their interactions.

The encryption of the plaintext m generates the ciphertext $c \in \mathcal{R}_{q_l}^2$, i.e. c is composed by two polynomials (c_0, c_1) of this form:

$$c_0 = [pk_1 \cdot u + m + e_1]_{q_l}$$
$$c_1 = [pk_2 \cdot u + e_2]_{q_l}$$

The polynomial u is sampled from $\mathcal{R}_3$, and the random polynomials e_1 and e_2 are sampled from $\mathcal{X}_\sigma$. Note that the CKKS encryption generates ciphertexts at any level l.

Then, c is evaluated with the function $f()$ that generates the evaluated ciphertext $c' = f(c)$ in $\mathcal{R}_{q_l}^2$, which is decrypted by multiplying the private key sk with c'_1 and adding it with c'_0. That is:

$$\begin{aligned}
[c'_0 + c'_1 \cdot sk]_{q_l} &= [pk_1 \cdot u + m' + e_1 + (pk_2 \cdot u + e_2)sk]_{q_l} \\
&= [(-a \cdot sk + e)u + m' + e_1 + (a \cdot u + e_2)sk]_{q_l} \\
&= [m' + e \cdot u + e_1 + e_2 \cdot sk]_{q_l} \\
&\approx m' = f(m)
\end{aligned}$$

Finally, the plaintext m' is decoded returning the message $\mathbf{z}' \in \mathbb{C}^{\frac{n}{2}}$ of complex values, which is an approximate result of $f(\mathbf{z})$. Recall that during the decoding, $\mathbf{z}'$ is rescaled by Δ^{-1}.

Formally, a CKKS scheme is defined as follow:

Definition 2.8 (*CKKS scheme*)**.** Let $d > 0$ be a fixed base for scaling in approximate computations and q_0 be a modulus, and let $q_l = d^l \cdot q_0$ for $0 < l \leq L$. A CKKS scheme is a tuple of algorithms $\mathcal{E} =$ (KEYGEN, ENC, DEC, ADD, MUL, RSC) with the syntax:

- KEYGEN$(1^\lambda) \rightarrow (pk, sk, ek)$: For a security parameter λ, outputs a public key pk, a private key sk, and a (public) evaluation key ek.
- ENC$(pk, m) \rightarrow (c)$: For a given polynomial $m \in \mathcal{R}$, outputs a ciphertext $c \in \mathcal{R}^2_{q_L}$.
- DEC$(sk, c) \rightarrow (m)$: For a ciphertext $c \in \mathcal{R}^2_{q_l}$ at level l and a private key sk, outputs a polynomial $m \in \mathcal{R}$.
- ADD$(c_1, \ldots, c_t) \rightarrow (c)$: For t ciphertexts $c_1, \ldots, c_t$ encrypting polynomials $m_1, \ldots, m_t$, outputs a ciphertext $c \in \mathcal{R}^2_{q_l}$ that is the encryption of their approximate addition $\sum_{i=1}^{t} m_i$. The level l of c is the minimum between the levels of the input ciphertexts and an error of c is bounded by sum of errors in input ciphertexts.
- MUL$(ek, c_1, \ldots, c_t) \rightarrow (c)$: For an evaluation key ek and t ciphertexts $c_1, \ldots, c_t \in \mathcal{R}^2_{q_l}$ encrypting polynomials $m_1, \ldots, m_t$, outputs a ciphertext $c \in \mathcal{R}^2_{q_l}$ that is the encryption of their approximate multiplication $\prod_{i=1}^{t} m_i$. A pairwise multiplication between input ciphertexts is $c_i \cdot c_j = (c_{i0} \cdot c_{j0}, c_{i0} \cdot c_{j1} + c_{i1} \cdot c_{j0}) + \lfloor P^{-1} \cdot c_{i1} \cdot c_{j1} \cdot ek \rceil \pmod{q_l}$, where P is a big integer.
- RSC$(c) \rightarrow (c')$: For a ciphertext $c \in \mathcal{R}^2_{q_l}$ at level l resulting from a multiplication, outputs a ciphertext $c' \leftarrow \lfloor \frac{q_{l'}}{q_l} c \rceil$ in $\mathcal{R}^2_{q_{l'}}$ where $l' < l$, i.e. c' is obtained by scaling $\frac{q_{l'}}{q_l}$ to the entries of c and rounding the coefficients to the closest integers.

2.3.3 Threshold Homomorphic Encryption

The idea of including MPC-based protocols (Chapter 2.2) in HE schemes was first-time presented by Cramer et al. (2001). In such work they proposed the usage of a MPC secret sharing protocol (2.2.1) to generate the cryptographic keys of a semantically secure PHE scheme. More precisely, it allows parties of a PHE scheme to collectively agree on a common public key, and to split the matching private key in fragments, so that each party holds one of them. The parties can then encrypt their individual inputs under the common public key and evaluate their desired functions homomorphically. To decrypt evaluated ciphertexts another MPC protocol is enforced, where parties reconstruct the private key by putting together their fragments. Cramer defines this new form of HE as *Threshold HE (THE)*, because a fixed threshold of fragments is needed to reconstruct the private key and enable decryption. Depending on the application context, the required threshold of fragments can vary, albeit the most secure configuration is a threshold equals to the number of parties, which forces a malicious adversary to take control of all parties to obtain all fragments.

Following Cramer's insight, in his PhD thesis (Gentry (2009b)) Gentry presents the notion of *Threshold Fully Homomorphic Encryption (TFHE)* with the purpose of applying FHE on multiparty setting. Basically, a TFHE scheme is a FHE scheme with the

difference that KEYGEN and DECRYPTION are now N-party MPC protocols instead of non-interactive algorithms.

Formally, we define a TFHE scheme for a group of N parties as follows, where the threshold of fragments required to decrypt is set to N:

Definition 2.9 (*Threshold Fully Homomorphic Encryption*)**.** Let $N = \{P_1, \ldots, P_n\}$ be a group of parties, where $n = |N|$. Each P_j holds a FHE scheme $\mathcal{E}_j$. A group $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ of schemes is threshold fully homomorphic TFHE, if each $\mathcal{E}_j$ = (MPC-KEYGEN, ENC, EVAL, DEC, MPC-DEC) has the following sintax:

- MPC-KEYGEN$(P_1, \ldots, P_n, 1^\lambda) \rightarrow (pk, sk_j, ek)$: Given all N parties and a security parameter λ, outputs a common public key pk, a fragment of the private key sk_j for P_j, and a (public) evaluation key ek, by executing a MPC protocol among N;
- ENC$(pk, m) \rightarrow (c)$: Given a public key pk and a message m, outputs a ciphertext c;
- EVAL$(pk, ek, \mathcal{F}, c_1, \ldots, c_t) \rightarrow (c)$: Given a public key pk, an evaluation key ek, a (description of a) boolean circuit $\mathcal{F}$ and t ciphertexts $c_1, \ldots, c_t$, outputs a ciphertext c;
- DEC$(sk_j, c) \rightarrow (c_j)$: Given a private key sk_j and a ciphertext c, outputs a partial decryption ciphertext c_j;
- MPC-DEC$(P_1, \ldots, P_n, c) \rightarrow (m)$: Given all N parties and an evaluated ciphertext c, outputs a message m by executing a MPC protocol among N.

We require absence of decryption failures and compactness of ciphertexts. Formally, for the class $\mathcal{C}$ of all circuits (i.e., both $\oplus$ and $\wedge$ logic gates), all sequences of n key tuples $(pk, sk_j, ek)_{j \in N}$ each of which is in the support of MPC-KEYGEN$(P_1, \ldots, P_n, 1^\lambda)$, all plaintexts $(m_1, \ldots, m_t)$ and ciphertexts $(c_1, \ldots, c_t)$, such that c_i in the support of ENC(pk, m_i), EVAL satisfies the following properties:

- *Correctness*: Let $c =$ EVAL$(pk, ek, \mathcal{F}, c_1, \ldots, c_t)$. Then MPC-DEC$(P_1, \ldots, P_n, c)$ $= \mathcal{F}(m_1, \ldots, m_t)$;
- *Compactness*: Let $c =$ EVAL$(pk, ek, \mathcal{F}, c_1, \ldots, c_t)$. There exists a polynomial p such that $|c| \leq p(\lambda, n)$. In other words, the size of c is independent of t and $|\mathcal{C}|$. Note, however, that we allow the evaluated ciphertext to depend on the n number of keys.

Note from definition 2.9 that the MPC-DEC algorithm requires all fragments of the private key to correctly decrypt an evaluated ciphertext, i.e. the threshold is

N-out-of-N. This provides a high level of security, since even a collusion of $n-1$ parties cannot reconstruct the private key. However, with a N-out-of-N threshold, if a party decides to join or leave the protocol, then is necessary to create a new set of fragments for the new set of parties by re-running the MPC-KEYGEN algorithm. It causes an extra parties' computational effort and can lead to serious performance issues if the number of parties wanting to join or leave significantly increases.

Last considerations about TFHE concern its efficiency compared to traditional HE schemes. As can be noted, the presence of MPC protocols in TFHE brings additional communication and computational overhead. Nevertheless, if the employed MPC protocols are efficient in terms of round of interactions among parties, then the overall TFHE scheme remains feasible and efficient. In this context is worth mentioning the work of Asharov et al. (2012) on TFHE, which proposes an efficient 3-round MPC protocol: one round for keys generation, one round for evaluation, and one round for decryption. In addition, they propose to outsource the bulk of computation, i.e. the EVAL algorithm, to an external Cloud service with the aim of further alleviating parties' effort.

2.3.4 Multi-key Homomorphic Encryption

In 2012, López-Alt et al. (2012) introduce the concept of *Multi-Key HE (MKHE)*, a new HE model capable of performing computations on data encrypted under multiple unrelated keys of different parties. In MKHE each party generates its own cryptographic keys and encrypts its confidential data with its public key. When a function needs to be evaluated on some input ciphertexts $c_1, \ldots, c_t$, where c_i is encrypted with pk_i, then each ith input ciphertext c_i is extended with the other $t-1$ public keys. Typically, extending a public key of a ciphertext with another is referred in literature as *key-switch* or *relinearization* (Brakerski and Vaikuntanathan (2011); Brakerski et al. (2012)). To later decrypt an evaluated ciphertext $\tilde{c}$ in MKHE model, all parties who supplied an input ciphertext in $\tilde{c}$ collectively run a MPC protocol using their respective private keys.

Formally, we define MKHE for a group of N parties as follows, where each party holds a $\mathcal{C}$-HE scheme and can run MPC decryption protocols:

Definition 2.10 (*Multi-Key $\mathcal{C}$-Homomorphic Encryption*)**.** Let $\mathcal{C}$ be a class of circuits. Let $N = \{P_1, \ldots, P_n\}$ be a group of parties, where $n = |N|$. Each P_j holds a $\mathcal{C}$-HE scheme $\mathcal{E}_j$. A group $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ of schemes is multi-key homomorphic MKHE, if each $\mathcal{E}_j$ = (KEYGEN, ENC, EVAL, DEC, MPC-DEC) has the following sintax:

- KEYGEN$(1^\lambda) \to (pk_j, sk_j, ek_j)$: For a security parameter λ, outputs a public key pk_j, a private key sk_j, and a (public) evaluation key ek_j for party P_j;

- $\text{ENC}(pk_j, m) \rightarrow (c)$: Given a public key pk_j and a message m, outputs a ciphertext c;
- $\text{EVAL}(\mathcal{F}, (c_1, pk_1, ek_1), \ldots, (c_t, pk_t, ek_t)) \rightarrow (c)$: Given a (description of a) boolean circuit $\mathcal{F}$ along with t tuples (c_i, pk_i, ek_i), each comprising of a ciphertext c_i, a public key pk_i, and an evaluation key ek_i, outputs a ciphertext c;
- $\text{DEC}(sk_j, c) \rightarrow (m)$: Given a private key sk and a ciphertext c, outputs a message m;
- $\text{MPC-DEC}(P_1, \ldots, P_t, c) \rightarrow (m)$: Given a set $T = \{P_1, \ldots, P_t\}$ of parties, where $T \subseteq N$, and an evaluated ciphertext c produced with $t = |T|$ public keys, outputs a message m by executing a MPC protocol among T.

We require absence of decryption failures and compactness of ciphertexts. Formally, for every kth circuit $\mathcal{F}_k \in \mathcal{C}$, all sequences of n key tuples $(pk_j, sk_j, ek_j)_{j \in N}$ each of which is in the support of $\text{KEYGEN}(1^\lambda)$, all sequences of t tuples $(pk_i, sk_i, ek_i)_{i \in T}$ each of which is in $(pk_j, sk_j, ek_j)_{j \in N}$, and all plaintexts $(m_1, \ldots, m_t)$ and ciphertexts $(c_1, \ldots, c_t)$ such that c_i in the support of $\text{ENC}(pk_i, m_i)$, EVAL satisfies the following properties:

- *Correctness*: Let $c = \text{EVAL}(\mathcal{F}, (c_1, pk_1, ek_1), \ldots, (c_t, pk_t, ek_t))$. Then $\text{MPC-DEC}(P_1, \ldots, P_t, c) = \mathcal{F}(m_1, \ldots, m_t)$;
- *Compactness*: Let $c = \text{EVAL}(\mathcal{F}, (c_1, pk_1, ek_1), \ldots, (c_t, pk_t, ek_t))$. There exists a polynomial p such that $|c| \leq p(\lambda, n)$. In other words, the size of c is independent of t and $|\mathcal{C}|$. Note, however, that we allow the evaluated ciphertext to depend on the n number of keys.

On the basis of this Definition 2.10 we define *Multi-Key Fully HE (MKFHE)* as follows:

Definition 2.11 (*Multi-key Fully Homomorphic Encryption*)**.** A group of encryption schemes $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ is multi-key fully homomorphic MKFHE, if each $\mathcal{E}_j$ is multi-key for the class $\mathcal{C}$ of all circuits.

As per $\mathcal{C}$-HE, semantic security of MKFHE follows directly from the semantic security of the underlying encryption schemes in the presence of the evaluation keys $ek_1, \ldots, ek_N$. This is because, given a subset $ek_1, \ldots, ek_t$, the adversary can execute the algorithm EVAL by herself. Note that taking $N = 1$ in Definition 2.10 and Definition 2.11 yield the standard definitions of $\mathcal{C}$-HE and FHE.

In a later extension of their work, in 2013, López-Alt et al. (2013) propose to use the *onion encryption* technique to extend input ciphertexts instead of conventional relinearization techniques. Practically, onion encryption wraps each c_i around $t - 1$ layers of encryption recursively. That is, input ciphertext c_i is first encrypted with pk_t,

its result is encrypted with pk_{t-1}, and so on until pk_{i+1}. The resulting ciphertext is then encrypted with pk_{i-1} and so again up to pk_1, finally producing the "onion" ciphertext z_i. This creates a new set $z_1, \ldots, z_t$ of onion ciphertexts, by which $\mathcal{F}$ can be homomorphically evaluated.

A different realisation of MKFHE under the *Learning with Errors* (LWE) (Regev (2005)) assumption has been proposed by Clear and McGoldrick (2015). The proposed MKFHE scheme supports an arbitrary number of parties by relying on a common reference string that must be known at key-generation time. Later on, the LWE-based MKFHE model was significantly simplified by Mukherjee and Wichs (2016) and further improved by Peikert and Shiehian (2016).

Chapter 3

Evaluating Blockchain systems: A comprehensive study of security and dependability attributes

Blockchain replaces traditional centralised infrastructures through a distributed network of entities that collectively fulfil operations without the need of trusting each other. It fosters the decentralisation of the infrastructure, in which there is not any trusted authority in control of the network. The advantages of decentralisation are threefold: there is not a single point of failure, trust is distributed across the network, and the whole system results harder to compromise (Troncoso et al. (2017)). However, the enhancements of decentralisation do not come at free cost. To ensure correctness and reliability, blockchains require complex distributed computing procedures. The lack of trust and the increased complexity can turn into system vulnerabilities. In particular, an adversary can take advantage of a wide attack surface to wreak havoc on the network and compromise its security. For instance, in 2021 about US$ 1.3B got stolen in decentralised finance applications (CertiK (2021)) by exploiting code vulnerabilities related to smart contracts - programs deployed and executed on the blockchain.

Security is nowadays a paramount need for blockchains. In traditional distributed systems, security is often paired with *dependability*. Those properties include a set of attributes that identify the reliability, availability, confidentiality, and integrity of a system during its execution (Avizienis et al. (2004, 2001)). In a blockchain context, where several parties exchange value via peer-to-peer transactions, it is crucial ensuring that the system remains secure and dependable thus avoiding problems like double-spending. However, blockchain systems entail several infrastructures and architectural choices such as the use of either a permissionless or permissioned

network, the *consensus* protocol, and the use of *smart-contracts* for applications. As a result, assessing the security of each component might be a challenging task.

In literature, some effort has been devoted to studying security in consensus protocols employed for blockchain systems (Cachin and Vukolić (2017); Shehar et al. (2019); Vukolić (2015)). However, a fair comparison is elusive due to several contrasting assumptions. Moreover, some works attempted to provide security evaluation of blockchains applications by assessing exploited vulnerabilities of smart contracts (Mense and Flatscher (2018); Atzei et al. (2017); Samreen and Alalfi (2021)), however, most of these studies mainly focus on the Ethereum platform (Wood et al. (2014)).

In this chapter, we provide a comprehensive evaluation of security in blockchains. To this extent, we propose a refined definition of security and dependability properties by referencing traditional CIA Triad - *confidentiality*, *integrity*, and *availability*, and we additionally introduce two new properties, namely *profiling* and *fairness*. The former determines the ability of a blockchain to authenticate participants and to define access control rules. The latter models the willingness of a system to be accessible by any participant and to process operations democratically. We therefore analyse blockchain systems by evaluating those properties with respect to three dimensions, namely *consensus*, *infrastructure* and *smart contracts*. We consider five most prominent blockchain platforms, namely *Bitcoin* (Nakamoto et al. (2008)), *Ethereum 2.0* (Ethereum (2022b)), *Algorand* (Gilad et al. (2017)) *Hyperledger Fabric* (Androulaki et al. (2018)) and a private instance of Ethereum, called *Ethereum private* (Go-Ethereum Docs (2022)). Firstly, we study the architectural models of these platforms to assess their security at the infrastructure level. Therefore, we focus on their underlying consensus protocol, i.e. the mechanism used by the network to democratically agree on the order operations. In particular we consider five consensus protocols, respectively called *Proof-of-Work* (PoW) (Nakamoto et al. (2008)), *Casper Proof-of-Stake* (PoS) (Ethereum (2022a)), *Pure Proof-of-Stake* (PPoS) (Gilad et al. (2017)), *Practical Byzantine Fault Tolerance* (PBFT) (Castro and Liskov (1999)), and *Proof-of-Authority* (PoA) (De Angelis et al. (2018)). Although consensus aims at guaranteeing security-by-design, any implementation relies on different assumptions under which the correctness of the protocol is guaranteed (Cachin and Vukolić (2017)). To this extent, we propose an analysis based on the tolerance to attacks. Specifically, we distinguish three different types of attack vectors that target *assets*, i.e. 'computing' in PoW and 'stake' in PoS, or *network nodes*, i.e. the maximum number of subverted nodes that PoA and PBFT can tolerate. Finally, we drift the analysis to the application layer built on top of the smart contracts capabilities of blockchains. We provide a detailed description of well-known code issues affecting smart contracts and thus evaluate how each issue impacts the security properties. The analysis evaluates then how the three smart contract enabled platforms considered in this study, i.e. Ethereum, Algorand and Hyperledger Fabric, address those issues and their mitigation effects.

Contributions

The content of this chapter has been published in De Angelis et al. (2019) and De Angelis et al. (2022). The contributions of this work are:

- a refinement of the security and dependability attributes of blockchain systems, by introducing and defining two novel security properties, i.e. profiling and fairness;
- a comprehensive security evaluation of the five prominent blockchain platforms, i.e. Bitcoin, Ethereum 2.0, Algorand, Ethereum-private and Hyperledger Fabric, with respect to their consensus algorithm, infrastructure, and smart contracts.

Chapter structure

Section 3.1 describes the distinction between permissioned and permissionless blockchain. Section 3.2 introduces the blockchain platforms we consider in our study and Section 3.3 describes their underlying consensus protocols, whereas Section 3.4 presents a collection of smart contract code issues. Then, Section 3.5 defines the refined security and dependability properties and the security analysis of those properties at consensus, platform and smart contract layers. Finally, Section 3.6 sums up the results.

3.1 Permissionless versus Permissioned

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*).

BitFury Group and Garzik (2015) define blockchain systems on the basis of these rights. Specifically, regarding Read operations a blockchain can essentially be divided in 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:

	read	**write**	**commit**
public permissionless	anyone	anyone	anyone
public permissioned	anyone	authorised participants	all or a subset of authorised participants
private permissioned	restricted to a subset of authorised participants	authorised participants	all or a subset of authorised participants

TABLE 3.1: Operations allowed on permissionless and permissioned blockchains

- *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 execute Commit and Write operations. On the opposite side, nodes of a permissioned blockchain are known at the outset, thanks to an 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 are able to execute Read operations. Table 3.1 shows a comparison between the identified models. *Public permissionless* blockchains, as 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, which can be stored on a digital *wallet*. Indeed, a cryptocurrency wallet stores the public and/or private keys for the accounts, and can be used to track ownership, receive or spend cryptocurrencies. 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 benefits from fairness property that derive from it.

3.2 Blockchain platforms

In this section, we describe the blockchain platforms considered in our analysis, namely Bitcoin, Ethereum, Algorand, Ethereum-private and Hyperledger Fabric. We briefly introduce the architectures, yet an overview of their performance and security.

3.2.1 Bitcoin

Bitcoin (Nakamoto et al. (2008)) is the first, open-source, permissionless blockchain born for electronic *machine-to-machine* payment without need of any central authority. Bitcoin transactions are processed in a fully decentralised manner and their ordering is guaranteed by an underlying *lottery-based* (i.e. probabilistic) consensus mechanism, i.e., the PoW. Such a solution allows a consistent, immutable, and therefore trustworthy, public ledger of transactions ever made. However, since the lottery-based consensus makes miners work in a decentralised way, multiple valid blocks can get mined at the same time; this makes it possible to fork the blockchain in multiple valid branches. However, forks eventually converge to a single branch through the longest-chain probabilistic rule. To compromise a block an attacker needs to acquire (or make collude) the 51% of the computational power of the miners. Compromising more than 6 blocks is considered computationally infeasible, therefore a block is considered final after ≈ 6 blocks. To avoid double spending and/or avoid spending tokens not owned, Bitcoin uses the UTXO model, i.e., each transaction is composed of a list of unspent transactions indicating the balance of accounts. Besides, the sender of a transaction is charged a mining fee (in BTC, the Bitcoin cryptocurrency) whose amount depends on the size (in byte) of the transaction, i.e., the number of UTXO addresses used. Miners must produce blocks with a maximum block size equal to 1MB. To ensure strong (eventual) integrity Bitcoin sacrifices performance, indeed the throughput is only about 5 txn/s with a block confirmation period of about 10 minutes. The difficulty of the PoW is adjusted over time according to the computational power to keep a fixed block confirmation time of about 10 minutes.

3.2.2 Ethereum

Ethereum (Wood et al. (2014)) is the second main open-source blockchain project. The underlying idea is to make the blockchain programmable through *smart contracts*, i.e., immutable pieces of code deployed and executed autonomously on the so-called *Ethereum Virtual Machine* (EVM). Smart contracts are developed in *Solidity* (Ethereum (2018b)), a Turing-complete programming language which can be interpreted by the EVM. The first version of Ethereum is based on the PoW consensus, like Bitcoin, but with a shorter confirmation time (about 14 seconds) which increases the throughput to about 30 txn/sec. This makes Ethereum more prone to forks than Bitcoin which are similarly solved with a longest-chain rule. The PoW makes Ethereum vulnerable to 51% attacks, like Bitcoin, therefore a block is considered final after 6 blocks. Ethereum does not employ a UTXO model to manage transactions, but an *account-based* model, i.e., each account has its balance stored within the state of the ledger. There are two types of accounts: *externally owned accounts* (EOA) and *contract accounts*. The former

are similar to standard Bitcoin accounts, whereas the latter can perform smart contract execution. Each transaction is charged according to:

- *gas price*: the amount of ETH (the Ethereum's cryptocurrency) to be paid for each computational step of the transaction's execution;
- *gas limit*: a scalar value representing the total amount of gas that can be consumed by the transactions in a block. It also determines the number of transactions handled in a block.

Each transaction has an overall limit to the gas it can consume to prevent runaway transactions, which accidentally or maliciously engage in never-ending computations.

3.2.2.1 Ethereum 2.0

It is the most important update of the Ethereum protocol to cope with scalability and performance issues. Among others, it proposes two major improvements, such as the shift from PoW to a new PoS implementation called *Casper Proof of Stake*, and the implementation of *Shard Chains*. The upgrade to PoS should evolve to a more energy-efficient platform, while Shard Chains may drastically improve scalability by changing the way the blockchain is replicated across the nodes of the network. Indeed, the traditional Ethereum implementation has a single blockchain replicated over thousands of nodes, conversely, sharding implies splitting the blockchain into smaller pieces, i.e., *shards*, which are distributed among a set of nodes which are responsible to handle only data of its shards. This allows the parallel execution of transactions, enabling the achievement of better throughputs. However this will come at a security cost since each shard is not managed by the entire network, thus it would be easier to tamper with it.

3.2.2.2 Ethereum Private Networks

Many implementations of the Ethereum protocol offer the possibility to be used in private settings. We refer them *Ethereum-private*. Two of the most common Ethereum clients are *Geth* (Ethereum (2013)), the Ethereum implementation in *Golang* language, and *Parity* (Ethereum (2018a)), a *Rust*-based implementation. Both permit the creation of a private instance of Ethereum, 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 types of chains are mainly used as *testnets*, where distributed applications are tested and debugged before deployment on the main Ethereum blockchain. Nevertheless, they are getting increasingly popular for business-to-business private enterprise settings which

require higher performance (hundreds of txn/sec) and higher privacy guarantees. The security of Ethereum-private does not depend on computational power, but on the number of nodes, the attacker can control (referred to as *byzantine*). The attacker needs to control at least 1/3 of nodes, but a wrong consensus implementation may drastically increase the probability of attack success.

3.2.3 Algorand

Algorand (Algorand Foundation (2017); Chen and Micali (2019)) is a novel permissionless blockchain platform that aims at solving the so-called blockchain trilemma, namely, scalability, decentralisation, and security. Algorand embeds a distributed computation engine, i.e., *Algorand Virtual Machine* (AVM), that runs on every node of the network and executes smart contracts, similarly to Ethereum. 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. The AVM interprets smart contracts written in an assembler-like language called *Transaction Execution Approval Language* (TEAL). The transaction model is similar to Ethereum, namely is account-based. Algorand's core innovation is its new consensus protocol, PPoS, which can reach agreement in large networks without giving up neither scalability nor security. Algorand blockchain is designed not to fork ever, transactions are considered final as soon as executed and included in a block. This makes Algorand much faster than Ethereum with a block time of about 4.5 sec and throughput of about 1000 txn/s. Compromising Algorand requires an attacker to control 1/3 of the stake.

3.2.4 Hyperledger Fabric

Hyperledger Fabric (Androulaki et al. (2018)) is a permissioned blockchain platform featured by a modular architecture in which each component can be plugged, including the consensus algorithm. The distinguishing characteristic of Fabric is that it splits the transactions ordering, i.e. the consensus process, from transactions execution, i.e. the operations on users' assets. The assets within the ledger state are represented as a collection of *key-value* pairs, and through smart contracts (called *chain-codes* in Fabric's jargon), it is possible to combine their values to carry out complex functions according to users' needs, e.g. to perform an auction. Being permissioned, Fabric offers an *authentication* layer that identifies the system entities by issuing X.509 digital certificates. Additionally, the authentication process enforces authorisation policies on the operations. In such a permissioned context, the risk of a participant intentionally introducing malicious 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. Differently from other blockchain platforms, Fabric introduces the concept of *channels*. Each channel represents a private blockchain restricted to specifically authorised members of a consortium. Transactions within a channel remain private and shared only across 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*: used to request read or write operations to a channel ledger and invoke smart contracts, if holding proper permissions;
- *peers*: hold a ledger for each channel they are registered in. Additionally, and if so designated, run and endorse smart contracts to execute functions on ledger data;
- *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, 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. As the operating environment is more trusted than a permissionless setting, it allows employing of lighter consensus schemas, for instance, PBFT, which results in better performances (3000 txn/sec), whereas tolerating up to 1/3 of subverted nodes.

3.3 Blockchain consensus protocols

A core component of blockchain systems is the underlying consensus protocol. It is employed to guarantee total agreement on the order of transactions and have decisive impact on the performance and security of a blockchain protocol. Consensus in blockchain networks is achieved essentially following one of two different approaches:

- *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. In particular, lottery-based algorithms can scale to a large number of nodes, hence are more feasible in permissionless networks where the elected leader simply propose a new block by a broadcast to the rest of the network. However, in this protocols the election of a leader can be extremely expensive in terms of time and resources, leading to a potentially crippling performance degradation. On the other hand, the voting-based approach is advantageous for permissioned networks, where the number of participants is limited and known. Voting consensus protocols afford lower latencies than the lottery-based: as soon as a majority of nodes agrees on a transaction, consensus is achieved. Yet, voting-based algorithms typically require intense message exchanges. Therefore, the higher the number of nodes in the network, the higher message exchanges is required to reach consensus. For this reason, in presence of large-scale networks, voting-based protocols performance may degrade due to intense communications, leading to very bad scalability.

Even though blockchain systems boast distributed consensus, the lottery-based approach differs from classical strong consistent consensus protocols, which implement total order broadcast and state machine replication. Indeed, blockchain lottery-based algorithms may admit multiple winners and, therefore, lead to forks in the blockchain. 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 transaction can never be removed from the blockchain once its block is appended to it (Vukolić (2015); Vukolic (2016)). Instead, blockchains powered by the voting-based approach implement the classical strong consistent distributed consensus. This implies the use of classical *Byzantine Fault Tolerance* (BFT) protocols, which ensure low latencies and high performance as well as guaranteeing consensus finality to blockchain. However, due to the inherent scalability limits of BFT, many hybrid protocols have been recently proposed by the blockchain community, such as PPoS and PoA. These are based on a combination of lottery and voting approaches. They aim to boost the power and applicability of blockchain platforms by overcoming both the scalability issues of voting-based protocols and the performance issues of lottery-based protocols.

In rest of this section, we describe the consensus underlying the blockchain platforms mentioned in the previous section, namely PoW, CPoS, PPoS, PoA and PBFT.

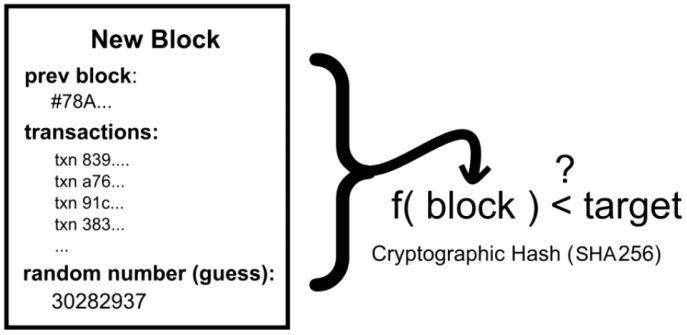


FIGURE 3.1: Proof-of-Work as a computational puzzle to solve a block

3.3.1 Proof-of-Work

The PoW is a lottery-based consensus schema consisting of computationally-intensive hashing tasks executed by some distinctive network nodes, called miners. Specifically, miners create a block by retrieving transactions from a *mempool* and compete with each other to find a random number (a.k.a., *guess*) such that, if concatenated with the transactions included in a block, makes the hash of the block lower than a target number (see Figure 3.1). Such target number is adjusted over time according to a desired *difficulty*, which is chosen to keep constant the *block period*, i.e., the average time required by miners to solve the puzzle. The more the global computational power of the network, the higher the difficulty, thus the lower is the target number.

Once resolved the guess, the miner can broadcast the corresponding block to the network for being accepted by other nodes. If accepted, all the correct nodes consider it as the latest block in the chain and start mining new blocks on top of it. In the event that multiple miners concurrently create and propose new blocks, a transient fork is created. It is the responsibility of the platform implementing the PoW to find a strategy to cope with forks. The standard approach used by Bitcoin and Ethereum, as mentioned in the previous section, is the longest-chain rule. Miners are incentivised to support the network honestly through a rewards mechanism: for each mined block, the miner receives a reward according to a certain (platform) policy. Besides, transactions may include a mining fee to incentive miners to pick that from the mempool. As side-effect, nevertheless, transactions with zero or low mining fees may never be included in a block generating *starvation* (BitFury Group (2015); Nakamoto et al. (2008)). The verification and subsequent acceptance procedures happening in PoW make a block *persistent* unless an attacker controls the majority of the miners' hash power (the aforementioned 51% attack), which enables it to create a chain fork with modified transactions. However, being based on computational power rather than several nodes, it is not vulnerable to *sibling attacks*.

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. Moreover, PoW is *energetically inefficient*, leading a huge waste of money and resources to make the hashing computations.

3.3.2 Casper Proof-of-Stake

The *Proof-of-Stake* (PoS) works by deterministically selecting a set of validators according to their cryptocurrency holdings, i.e. their stake. Any node committing a stake can become a validator by sending a special type of transaction that locks up their stake amount into a deposit. The validators propose and vote on the next block, and the weight of each validator's vote depends on the deposit amount. In Ethereum's PoS implementation (Ethereum (2022a)), called *Casper* (CPoS), each validator's turn is determined by one of the following techniques:

- *Chain-based PoS*: the algorithm pseudo-randomly selects a validator during each time slot to propose a block. The block is then appended to the blockchain;
- *BFT-style PoS*: validators are randomly assigned the right to propose blocks. The block agreement is found through a multi-round process, in which each validator sends its vote for a specific block. 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.

Conversely to PoW, the CPoS protocol causes no waste of energy since it does not requires computational tasks to be solved, therefore, performance can be much better. In PoS validators earn a reward proportional to their deposit stake, for every proposed block that is accepted by the majority, which induces them to act honestly. If a validator does not follow the consensus rules, PoS applies penalties. Attacking a PoS requires an attacker to control the majority of committed staking, making it not vulnerable to *sibling attacks*. However, it is crucial not to make predictable the leader, otherwise, the attacker just needs to compromise a much smaller set of M nodes that may be elected as a leader; in this case, the security drops from the ideal majority of committed staking to M compromised nodes.

3.3.3 Pure Proof-of-Stake

The PPoS (Gilad et al. (2017)) is the underlying consensus of Algorand. It leverages on the Verifiable Random Functions (VRF) concept (Micali et al. (1999)) to significantly decrease the high volume of exchanged messages occurring in traditional voting-based and lottery-based consensus. The VRF makes PPoS 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. PPoS works as follows: it proceeds in rounds, and for each round there are three phases: *block proposal*, *soft vote*, and *certify vote*. When a round starts, users use the VRF to select themselves as leader and/or committee members. VRF then randomly choose among them for the round. The leader's role is to propose a block, whereas committee's role is to vote and validate that proposal. The usage of randomness, besides allowing anyone in the network to participate in the consensus, discourages adversaries from compromising consensus nodes, since they randomly change in each round. The phases for each round in PPoS proceed 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:* a selected committee of users vote on the block proposals. Given that, for each round, multiple users might be elected as a leader and propose a block, this phase filters the number of proposals down to one, guaranteeing that only one block gets certified in a 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 certified 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 to agree on a new leader and initiate a new round. This approach ensures safety when partitions occur without making the blockchain fork.

PPoS can achieve higher throughput and lower block time than traditional PoS due to a reduced message exchange. Furthermore, the VRF makes the leader unpredictable, dismissing the possibility of M-points-of-failure of traditional PoS.

3.3.4 Practical Byzantine Fault Tolerance

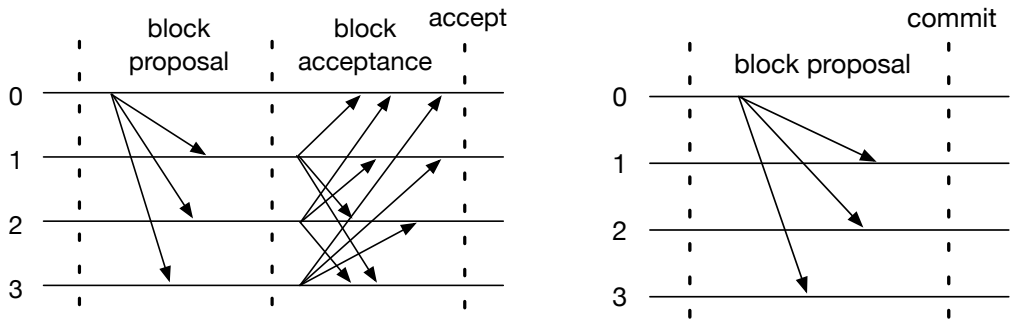
The *PBFT* consensus protocol (Castro and Liskov (1999)) is an extension of the Paxos/VSR (*ViewStamped Replication*) (Lamport (2002)) family and it is characterised by a single-leader and view-change approach. The algorithm proceeds in views, for each view there exists a leader and a set of replicas. Each view executes a *three-phase* protocol where consensus nodes exchange messages to reach the total order of transactions:

- *Pre-prepare*: in the first phase the leader broadcasts to replicas the set of incoming transactions for the current view.
- *Prepare*: Once checked the signature of pre-prepare message and the validity of its contained transactions, each replica orders transactions and broadcasts them to others (including the leader).
- *Commit*: the leader and the replicas first verify the signatures of received prepare messages. Then, they check whether the prepare messages are coherent with those of pre-prepare phase: they are from the same view, proposed by the same leader, and with the same transactions set. If these conditions are satisfied they finally commit on the transactions ordering.

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 (Castro and Liskov (1999)). Furthermore, this algorithm is vulnerable to sibling attacks, since it cannot distinguish if an attacker is falsely impersonating multiple nodes. This makes PBFT unusable on permissionless settings.

3.3.5 Proof-of-Authority

PoA was originally proposed as part of the Ethereum consortium for private networks and implemented with the protocols called *AuRa* (Parity (2017)) and *Clique* (Szilágyi (2017)). PoA algorithms rely on a set of trusted miners called *authorities*, identified by a unique *id*. Consensus in PoA relies on a *mining rotation* schema, which distributes the responsibility of block creation among authorities (BitFury Group and Garzik (2015); Gaetani et al. (2017)). Time is divided into *steps*. In each step, an authority is elected as block proposer, i.e. as a leader. The way authority is elected differs in the



(A) Aura (B) Clique

FIGURE 3.2: Example of message exchanges during a step for Aura and Clique. In both there are 4 authorities with id 0, 1, 2 and 3, and the step leader is authority 0.

two consensus protocols. AuRa proposes a deterministic function based on UNIX times, which requires strong synchronisation assumptions on the network. Conversely, Clique computes leaders according to the number of the next block on the blockchain. As can be gathered from Fig. 3.2, 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 a constant rate Parity (2017). These algorithms are vulnerable to sibling attacks, and thus cannot be used in permissionless settings.

3.4 Smart Contract issues

Beyond secure-by-design due to consensus algorithms, a prominent security role is played by smart contracts. In this section, we evaluate potential issues that affect the smart contract-enabled platforms, such as Ethereum, Algornad, and Hyperledger Fabric, thus we consider possible preventions/mitigation methods.

(I_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 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 infinitely. Ethereum is vulnerable to reentrancy and its exploitation may lead to indefinite withdrawal calls which terminate once the caller contract is drained of ETH or the transaction runs out of gas. Two reasons cause this vulnerability (Rodler et al. (2018)): (i) validity checks are handled by state variables that the contract does not update until the execution of other transactions terminates, (ii) no gas limit is required when handling interactions between external smart contracts. Prevention methods consists in (i) update the state variables before calling external contracts; (ii) introduce a `mutex lock` (Ethereum (2018b)) in the contract state so that only the owner can update such state and avoid reentrant calls to an external contract from a caller; (iii) use the `transfer` (Ethereum (2018b)) method to approve payment transactions to external contracts as this function avoids infinite loops because only provides 2300 *gas* to the callee. Similarly, Hyperledger Fabric suffers reentrancy since chaincodes-to-chaincodes are allowed with no limitations inter-channel. Fabric mitigates such issues through a timeout, however, it is important to note that reentrancy has a limited impact on private settings since no cryptocurrencies are involved. Conversely, Algorand does not suffer reentrancy since contract-to-contract calls are allowed one way only, thus if smart contract A calls a smart contract B, the latter cannot call back A.

(I_2) Integer overflow and underflow. This vulnerability occurs when a function computes an arithmetic operation that falls outside a specific datatype. Such vulnerability may affect both platform's software and smart contracts. It is unrealistic to state that a code cannot be affected by overflow, thus every platform and related smart contracts (if any) may be vulnerable. A prominent role is played by the programming language. Furthermore, some protections there exist both natively or through an external library. Ethereum does not provide native prevention for smart contracts, but some recommendation has been defined, such as (i) using *SafeMath* library (OpenZeppelin (2017)) to check on underflow/overflow, (ii) using *Mythril* library (ConsenSys (2020)) to check the security of EVM bytecode before its execution. Algorand does not need any prevention library as TEAL natively copes with under/overflow issues. Hyperledger Fabric, being based on golang makes us of the native under/overflow management or common libraries such as *overflow*.

(I_3) Frozen Token. This vulnerability causes users' funds deposited in the contract account to be locked and impossible to withdraw back, effectively freezing them into the contract. Both Ethereum and Algorand are vulnerable to such an issue. 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 a library, (ii) the callee contract function (`selfdestruct` for Ethereum, `ClearState` for Algorand) is executed without checks. Prevention method (Chen et al. (2020); Samreen and Alalfi (2021)): a deposit contract shall assure that the mission-critical functions or

money-spending functions are not outsourced to external contracts. Hyperledger Fabric is not vulnerable since no cryptocurrencies are involved.

(I_4) DoS with unexpected revert. This issue occurs if the execution of a transaction is reverted due to a thrown error or a malicious callee contract that deliberately interrupts the execution. Ethereum, Algorand and Hyperledger Fabric are vulnerable. For Ethereum, a best practice to mitigate the issue regards the transaction sender to provide to the callee a certain amount as a reward for the execution of a transaction so that the callee is not incentivized to revert. Algorand does not have mitigation in place since no reward fees are available. Fabric, similarly, does not have solutions to avoid it, due to the absence of cryptocurrencies.

(I_5) DoS with GasLimit. This vulnerability causes transactions to be aborted due to exceeding the gas limit. This affects only Ethereum due to the presence of gas. This vulnerability is caused by unbounded operations in a contract (Chen et al. (2020); Samreen and Alalfi (2021)). To mitigate this issue the contracts should not execute loops on EOA accessible data structures. Therefore loops should be controlled, such that the execution always terminates, even when transactions are aborted.

(I_6) 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 (Chen et al. (2020); Samreen and Alalfi (2021)). A proxy contract acts as a deposit that 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 to the withdrawal transaction and drain the proxy balance. Prevention method: The contract shall check the contract address and the nonce for each withdrawal transaction used with digitally signed messages. Both Ethereum and Algorand are vulnerable to this issue, while Hyperledger Fabric is not since it authenticates network nodes.

(I_7) Generating randomness. This vulnerability concerns smart contracts 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. Solidity's `block.number`, `block.difficulty`. A malicious miner can manipulate the PRNG to generate an output that is advantageous for itself. Ethereum, Algorand and Hyperledger Fabric are all affected by this issue. For mitigation, mining variables should not be used in control-flow decisions. Therefore, off-chain approaches to PRNG should be used, such as the use of oracles.

(I_8) Block Timestamp manipulation. This vulnerability affects smart contracts that use `timestamp` parameter in the control-flow (*e.g.*, timestamp used to schedule periodic payments) or as source of randomness (Chen et al. (2020)). 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. Ethereum is vulnerable to such issues and a prevention method consists in avoiding the parameter in any control-flow decision logic. Algorand is not vulnerable since it employs a maximum timestamp offset of 20 seconds between two blocks. Similarly, Hyperledger Fabric has no constant block time, but it depends on the transactions available to include in a block.

(I_9) 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 receives to execute a transaction. It is usually used to prioritise the execution of certain transactions over others (Wood et al. (2014)). However, malicious miners can alter this procedure and always prioritise their transactions regardless of the gas price, hence manipulating the global state of the blockchain in its favour (Samreen and Alalfi (2021)). Mitigation method: hide the gas price from transactions using cryptographic committees or guard conditions (Chen et al. (2020)). Algorand and Hyperldger Fabric are not affected by this issue since they do not use gas.

(I_{10}) Under-priced opcodes. This vulnerability is caused by under-priced opcodes that can be executed at low cost and that consume a large number of resources. Misuse of the opcodes, or a malicious actor, might trigger several invocations of these opcodes wasting the majority of resources. This vulnerability regards Ethereum and it is caused by misconfigured gas price parameters (Chen et al. (2020)). The Ethereum protocol has been upgraded to limit opcodes under-pricing. Algorand is not affected since the cost is set 1 to all opcodes with a limit of 700 per application. Hyperledger is not affected due to the nature of private blockchains' costless computation.

(I_{11}) Token lost to orphan address. This vulnerability is caused by a lack of validation checks on payment transactions. Ethereum only checks the recipient's address format but not if such an address is valid nor if it exists. If a user sends money to non-existing addresses, Ethereum automatically creates a new *orphan* address Atzei et al. (2017). An orphan address is neither an EOA nor a contract address, hence the user can't move the money which is indeed lost. Algorand has the very same issue, with a small client-side check of existence as mitigation implemented in all the official clients and SDKs. The only effective prevention method at the time of writing is to manually assure the correctness of the recipient's address (Chen et al. (2020)). Hyperledger nodes are instead authenticated, thus it is not affected.

(I_{12}) Short address. This vulnerability affects only Ethereum and it 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 fewer bytes, EVM auto-pads with zeros to generate the 32 bytes chunk. An attacker could manipulate this process to execute malicious actions. For instance, 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 to transfer (Chen et al. (2020)). To prevent that, write functions that validate the length of the transaction's inputs. Algorand has prevention by design, it does not add extra zeros as padding and the transaction fails in case of a short address. Hyperledger Fabric, as above, is not affected due to the authentication of nodes.

(I_{13}) 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 (Chen et al. (2020)). To avoid this, with Solidity 0.5.0 and above, it is mandatory to specify the function visibility. Algorand conversely has all functions public only. Hyperledger can hide data in several methods such as Trusted Execution Environment with Intel SGX, channels, or MPC (Benhamouda et al. (2019)).

(I_{14}) Unprotected suicide. In Ethereum, 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. The very same situation happens with Algorand through the `ClearState` function. Prevention method: enhance security checks with, permissions mechanisms, to assure that the `suicide/self-destruct` and `ClearState` calls are approved by different parties. Hyperledger Fabric is not affected.

(I_{15}) Unrequested Token. In Ethereum ERC-20 tokens can be sent to an arbitrary address. This is used as a common phishing technique where a malicious actor creates an ERC-20 token and sends some amount to random addresses. The token is designed with a sell smart contract function which drains the wallet. When a phished user attaches his wallet to the application controlling this contract, the user unintentionally authorises the smart contract to steal his funds. Algorand mitigates this issue via opt-ins. Hyperledger Fabric is not affected since no cryptocurrencies are involved.

3.5 Evaluation of security properties for Blockchain

In this section we introduce security and dependability attributes for blockchains and we evaluate them over three dimensions, i.e., *consensus*, *infrastructure* and *smart contracts*. The proposed analysis follows a qualitative evaluation that takes into account the descriptions of the platforms and protocols detailed in the previous sections. Therefore, the analysis considers how the identified properties are met in each system. We assume a deployment of n nodes responsible for consensus, which communicate over an eventually synchronous network (Dwork et al. (1988)), which realistically describes the Internet network, where messages can be arbitrarily delayed, but eventually delivered within a fixed time-bound.

A distributed protocol is considered secure if satisfies *safety* and *liveness* properties (Cachin and Vukolić (2017)). However, in a blockchain context, the traditional definition of such properties does not straightforwardly apply. For instance, transactions are asynchronously committed by the network after the execution of a consensus protocol. For this reason, safety and liveness must be refined in order to explicitly reflect the behaviour of a blockchain system. To cope with this limitation, we start from the traditional definitions of safety and liveness (Cachin and Vukolić (2017)) to introduce two novel properties, namely *persistency* and *termination*. We also provide a new metric for blockchains, i.e. the *fairness* property. Following seminal work by Francez (1986), we distinguish two aspects of the fairness: *validator fairness*, valid for consensus protcols and *client fairness*, valid for infrastructures. The security attributes for consensus can be summarised as follows:

1. *Persistency*: nothing wrong happens during the consensus execution; unwanted executions must be prevented. When an honest node accepts a transaction, then all the other honest nodes will make the same decision, which is irreversible. If persistency is violated after a certain threshold (i.e. confirmation time), it will never be satisfied again. Persistency is also referred to as *finality* (Vukolić (2015)).

2. *Termination*: ensures that a protocol makes progress towards an end, hence transactions correctly terminate, i.e. the block including those transactions reaches persistency.

3. *Validator fairness*: in a blockchain, the consensus 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.

Table 3.2 summarises the consensus resilience of the five algorithms acting in presence of possible adversaries. Firstly, we observe that PoW and CPoS enjoy strong termination. This property follows because the probabilistic leader election is based

	PoW	CPoS	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 3.2: Security evaluation of blockchain consensus protocols

on mining, and staking. Transactions are guaranteed to be added to the blockchain as soon as the mining proceeds, or there is a majority of stake-holders. Conversely, probability in leader election affects persistency of PoW and CPoS, because of 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 this reason, the persistency of such protocols must be classified as *eventual*. Conversely, the PPoS protocol does not allow forks, and blocks are instantly finalised, prioritising persistency over termination (Gilad et al. (2017)). Indeed, to ensure persistency, the PPoS allows stalls from malicious behaviours or network issues. However, PPoS' recovery mechanism ensures the protocol's 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. the richest users will be advantaged in the proposal of new blocks. This behaviour is reflected with the validator fairness property in Table 3.2. Differently in PoW, such property is strongly related to the hardware capabilities and miners with a lot of computational power will have more chances to solve new blocks.

Moving to permissioned blockchains, these systems 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, as 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. On the other hand, in PoA, persistency is only *eventually* guaranteed, because PoAs are open to forks. Termination is instead dependent on the number of honest validators, hence classified as eventual. As long as a majority of validators are 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 a leader as the others.

Turning to the security evaluation of blockchain platforms, we define six attributes based on the traditional definitions of security and dependability (Avizienis et al.

	Bitcoin	Ethereum 2.0	Algorand	Hyperledger	Ethereum-private
confidentiality	no	no	co-chains	channels	private txs
integrity	majority of hash power	majority of stake	majority of 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 3.3: Security evaluation of blockchain platforms

(2004)), i.e. the *CIA triad* and the user's *profiling*. We identify the relevance of the properties of *accountability* and *authorisation*. Such properties lead to constraints which can be used to detect misbehaving participants. Hence, our suggested attributes of security and dependability properties to assess blockchains' infrastructures and smart contracts are:

1. *Confidentiality*: the possibility to keep some transactions confidential; absence of unauthorised leaking of sensitive information owned by one or more nodes;
2. *Integrity*: absence of improper alterations of the blockchain data from unauthorised users;
3. *Availability*: the ability of the system to run correct services without interruptions;
4. *Authorisation*: the ability of the system to specify access rights and privileges to resources and to define permission roles for participants;
5. *Accountability*: the ability of the system to trace back the operations and the behaviour of a certain user identity/physical entity;
6. *Client fairness*: the willingness of the system to democratically accept transactions from any client without any preference.

Table 3.3, shows our infrastructures analysis understudy over the aforementioned six attributes. 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 already mentioned, hence maliciously injecting a fork with a subverted chain (Bonneau et al. (2015)). 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 due to

Issue ID	Security Issues	Native Resistance/Mitigation		
		Ethereum	*Algorand*	*HL Fabric*
I_1	CI + authorisation		✓	
I_2	CI + authorisation		✓	✓
I_3	A + authorisation			✓
I_4	A			
I_5	A		✓	✓
I_6	CI + authorisation			✓
I_7	I + authorisation			
I_8	IA		✓	✓
I_9	IA + accountability		✓	✓
I_{10}	A		✓	✓
I_{11}	I			✓
I_{12}	I + authorisation		✓	✓
I_{13}	CI + authorisation	✓		✓
I_{14}	all			✓
I_{15}	authorisation		✓	✓

TABLE 3.4: Taxonomy of security issues and native resistance/mitigation

the public nature of the information stored on these blockchains (Henry et al. (2018)). Indeed, each node in the network has access to the entire ledger of transactions, and it is possible with forensics activities to trace the behaviour of specific users (Karame et al. (2015)). However, if for Bitcoin and Ethereum 2.0 there is no way to mitigate this issue, Algorand recently designed a solution which combines the public, permissionless network with several private networks interconnected, a.k.a., *Co-Chains* (Micali (2020)). 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 (Herlihy and Moir (2016)). This is not so in public permissionless blockchains like Bitcoin, Ethereum 2.0, and Algorand where identities are pseudo-anonymous (Henry et al. (2018)) and users are not authenticated. However, although actions are difficultly attributable to specific entities, it is possible to analyse 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 (Karame et al. (2015); Möser (2013)). 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 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 transactions may be stalled forever waiting to be processed (Weber et al. (2017)). 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 transaction fees, which are thus distributed to rewards accounts for the users, and to ensure client fairness.

We conclude our analysis with smart contracts. Table 3.4 shows a classification of the *CIA triad* and *profiling* security properties against the issues described in Section 3.4. From the table emerges that most of the smart contract issues may cause violation of confidentiality, integrity, and authorisation properties. The platform that shows a better resilience results in Hyperledger Fabric and this is clearly due to its permissioned nature. Indeed, the authorization layer prevents security issues due to a trustless network. Among the public blockchain instead, Algorand outperforms Ethereum for many issues. This is due mainly to the usage of a constant fee for transactions and opcodes conversely to Ethereum which is based on gas with a different amount for opcodes. Also, the programming language used plays a key role. Both Hyperledger Fabric and Algorand use languages that give some primitive control to avoid issues, such as control against under/overflow concerning Ethereum. Finally, some design choice related to the management of asset and smart contract calls makes Algorand more secure than Ethereum. For instance, *reentrancy* (I_1) in Algorand cannot happen by design, and tokens require to be opted-in before they can be received (I_{15}). The Ethereum naive solutions and lack of controls make it the worst in terms of security. The only situation where Ethereum outperforms Algorand is for *erroneous visibility* (I_{13}), indeed it allows to build private functions within a smart contract, while Algorand does not.

3.6 Discussion

In this chapter we studied the security aspects of modern blockchain systems. We defined the security and dependability attributes which are relevant in the context of a blockchain. Thus, we analysed how five blockchain platforms, namely Bitcoin, Ethereum (with its variants Ethereum 2.0 and private chains), Algorand, and Hyperledger Fabric, guarantee those attributes. The analysis we proposed is divided into three dimensions, i.e. consensus, infrastructure, and smart contracts. Firstly, we highlighted the infrastructures' characteristics and how they differ in terms of

performance (infrastructure). Then, we described their built-in consensus protocols, respectively, PoW, Casper PoS, PoA, Pure PoS and PBFT, analysing the approaches they adopt to order transactions and create blocks in a byzantine, eventually synchronous, network model (consensus). Then, we listed smart contract issues evaluating whether blockchains are vulnerable and their native resistance/mitigations (smart contracts).

From our study emerged that permissioned blockchains like Hyperledger Fabric and Ethereum-private can guarantee fairness and confidentiality while providing accountability and authorisation. However, these platforms require strong assumptions on the underlying network and the number of possibly subverted nodes to also ensure integrity and availability. This is also reflected in the consensus protocol they adopt, specifically PBFT guarantees persistency at the cost of eventual termination, whereas in PoAs, both properties are ensured only eventually. Conversely, permissionless platforms such as Bitcoin, Ethereum 2.0, and Algorand offer better integrity and availability, despite failing on profiling, confidentiality, and client fairness properties. Finally, by studying smart contract issues we observed that Ethereum is the most vulnerable smart contract platform compared to Algorand and Hyperledger Fabric.

In the next chapters, we investigate how to enhance the blockchain security and performance, proposing novel solutions to respectively guarantee data privacy and increase the scalability. Chapters 4 and 5 describe two blockchain systems integrated with HE and MPC computing models, where data are encrypted on the ledger and smart contracts can still perform computations with these data despite being ciphertexts. Chapter 6 describes a blockchain system with a new architecture based on the sharding technique, which enables to scale out the nodes engaged in the consensus protocol and improve performance.

Considering the study made in this chapter, we decided to apply the solutions in Chapters 4, 5 and 6 to permissioned blockchains. There are two reasons behind this choice for Chapters 4 and 5. Firstly, nodes in the permissioned blockchain, both clients and validators, are assigned with verifiable cryptographic keys, which are used for authentication and authorisation. We leverage these keys to verify the HE computations performed by smart contracts and messages exchanged during the execution of a MPC protocol. Secondly, permissioned blockchains rely on voting-based consensus algorithms, which are more performant than those lottery-based of permissionless blockchains. This become a factor when we integrate HE and MPC into blockchain, as they add performance overheads. Indeed, the HE primitives of keys generation, encryption, decryption and evaluation have a computational cost. Smart contracts take longer to perform HE computations on ciphertexts than on plaintexts. On the other hand, the MPC protocols have a cost due to the exchange of messages between participants.

For Chapter 6, instead, the choice of a permissioned blockchain is a natural consequence of the work proposed in Chapters 4 and 5, to face the scalability issue in the same setting and system model. Indeed, the purpose of this thesis is to provide two novel, effective and viable solutions that can be both applied on the same permissioned system, simultaneously solving the blockchain problems of privacy and scalability. Notwithstanding, we believe that the solutions in Chapters 4, 5 and 6 can also be applied to permissionless blockchains with the appropriate adjustments. We leave this research path as future work.

Chapter 4

Applying Blockchain and Homomorphic Encryption to Smart Grid

The presence of IoT devices is considerable nowadays. They simplify and improve people's daily activities by monitoring the surrounding environment, gathering information, and taking automatic decisions. As an example, consider the smart meter: when installed in a property it records the actual consumptions and transmits the data in real-time, enabling the energy supplier to calculate the bills accurately and to have a more precise estimation of the energy usage. This brings a two-fold benefit, the landlord saves money and the energy supplier does not waste energy. Despite their great potential, IoT devices come with some downsides, mostly related to the lack of security measures. Adversaries can intercept the data in transit, eavesdrop their content and also tamper with them. Surveys on IoT security are presented in (Alaba et al. (2017); Vashi et al. (2017); Yang et al. (2017)), where the IoT vulnerabilities are analysed at different layers of its architecture, along with a classification of the possible attacks to which each layer can be exposed. The *Smart Grid* systems, relying on smart meters, suffer from a subset of these IoT attacks. Energy data can be altered causing erroneous bills, with money-losing implications for either energy companies or prosumers. If an energy trade or an auction is in place, prosumers can buy or sell energy to wrong prices. All these security concerns create scepticism about the Smart Grid adoption.

In this chapter we present a novel Smart Grid system featured by blockchain that we developed under the UKIERI research programme. A blockchain node is deployed for any prosumer of the Smart Grid. It is responsible for retrieving energy data from the smart meter, storing this data on the ledger and processing tasks via smart contracts. With its cryptographic functions and consensus protocol, the blockchain lends strong

integrity properties to energy data collected by smart meters: once appended to the ledger they can no longer be altered. Smart contracts, by consulting the ledger data, can automatically calculate bills or execute energy trading, and their correctness is verified by all nodes of the network during consensus. These blockchain qualities streamline the Smart Grid management and enhance its security, thus raising trust among the parties involved.

After the end of the UKIERI research project, we focused on how to improve the proposed Smart Grid blockchain system from the privacy perspective. Users may be wary of sharing their energy data, which disclose how much they consume, spend, or produce if equipped with energy generators. In addition, energy data may inadvertently reveal the habits of users. An ill-intentioned adversary, for example, can inspect weekly consumptions to understand when users are away from home. In order to resolve these privacy concerns and prevent possible threats, a typical solution is to encrypt data. However, this raises a problem for blockchain-based systems: smart contracts can no longer perform computations on the data, as they are encrypted. Hence, our previous blockchain system is no longer able to calculate energy bills, restricting its functionality to secure data storage. To cope with this hindrance we propose to integrate the Homomorphic Encryption (HE) model into blockchain, thus enabling smart contracts to carry out Smart Grid tasks while preserving users' privacy.

Contributions

The content of this chapter has been partially included in the UKIERI research project. Particularly, this work provides the following two back-to-back contributions:

- a novel blockchain-based Smart Grid system devised for the UKIERI project. My contributions to the project was to design the system architecture and develop the blockchain network on Hyperledger Fabric, i.e. the clients, peers, orderers and smart contracts.
- an improvement of the previous system by combining blockchain with HE. The two-fold advantage is (i) ensuring data privacy, i.e. system data are no longer in clear, and (ii) enabling private computing, i.e. computations can be performed on ciphertexts.

Chapter structure

Section 4.1 outlines the UKIERI research project, by detailing its objectives and our involvement. We describe the architecture that integrates blockchain into a Smart Grid

system, and its implementation on Hyperledger Fabric. In Section 4.2, we first highlight the privacy shortcomings of this blockchain-based solution. Then, we describe our proposal of integrating HE to fill this gap and whilst enabling smart contract computations on ciphertexts. Section 4.2.1 outlines the computational costs due to the introduction of HE. Section 4.2.2 and Section 4.2.3 describe how this HE-equipped version can be used for energy trading and applied to other IoT-based scenarios, respectively. Section 4.3 presents the state-of-the-art about the integration between Smart Grid and blockchain. Finally, Section 4.4 concludes and discusses the work.

4.1 Case study: the UKIERI research project

The *UK-India Education and Research Initiative (UKIERI)*[1] is a research and innovation programme that promotes the collaboration between UK and India. As for the UK side, we[2] participated to this programme from 2018 to 2020 in partnership with the India's Central Mechanical Engineering Research Institute (CMERI). *Robust Cyber Security Framework for Longitudinal Smart Energy Monitoring Device Data (CS-SED)* is the name given to the research project. The objective of CS-SED is to provide a resilient and secure infrastructure for IoT-based Smart Grid systems, addressing a variety of security threats and creating economic values. CS-SED aims at giving UK and India a strategic lead and a global competitive edge in this field of research, besides bringing about energy efficiency through an energy-aware neighbourhood reporting live data consumption.

The role of India's CMERI is to set up a Smart Grid environment. Hence, it concerns the development of low cost smart meter units and their subsequent installations in buildings, facilitating a smart IoT infrastructure for real time-stamped data collection reporting residential energy consumptions. Specifically, the Smart Grid system measures live power consumption by sampling voltage and current signals. On the other hand, our role is to create a blockchain network to enhance the security of the Smart Grid data and to support energy reporting and billing. Specifically, smart contracts convert the consumed KWH to Rupees after multiplying the former with constant power tariff. The result of these computations are securely stored on the ledger, to subsequently display online power consumption, both in terms of consumed KWH and balance Rupees. In the following, we present in Section 4.1.1 the blockchain-based Smart Grid architecture we developed for CS-SED, and in Section 4.1.2 its implementation on Hyperledger Fabric.

[1] http://www.ukieri.org

[2] A team of the Cyber Security group of the University of Southampton, to which I belong.

4.1.1 A Smart Grid system featured by Blockchain

In this section we present the high-level architecture that embeds a blockchain network into a Smart Grid system. Particularly, the architecture connects all the *prosumers*, i.e. energy consumers and/or producers, of a Smart Grid with an overlay blockchain network to securely manage the energy data. Each prosumer hosts a smart meter able to monitor energy status on the premises. It records the consumptions in real-time, specifically the readings collected are: voltage, current, power, energy frequency and power factor. Additionally, if the prosumer holds energy generators, e.g. solar panels or wind turbines, the smart meter records the amount of accumulated energy, which can be potentially traded. Smart meters are then able to communicate these collected data via wireless connections. It is assumed that these wireless communications are reliable, i.e. data are eventually delivered to the recipient, but they do not provide security, i.e. do not employ cryptographic measures.

We develop a software module, called *Secure Energy Transaction (SET)*, acting as an intermediary between prosumers and the blockchain network. The SET module includes an ad-hoc *client* application appointed to:

- retrieve energy data from the smart meter;
- send data to blockchain or call smart contract executions.

Hence, the SET client in its code embeds a smart meter listener and implements APIs to bilaterally interact with the blockchain. These API-requests to the blockchain are regulated by appropriate authorisations. Each prosumer of the Smart Grid connects with a SET client instance. SET clients are identified and assigned to prosumers by means of digital certificates released by a *Certificate Authority (CA)*. We indeed employ a permissioned blockchain platform, Hyperledger Fabric, that first authenticates the clients by checking their identity, and then approves or rejects their API-requests according to access control rules. In order to allow democratic control over the Smart Grid information, we configure the SET module to include a *blockchain node* for every prosumer, so that each of them holds a copy of the ledger and can actively participate in the consensus protocol. Blockchain nodes have smart contracts deployed, and upon a prosumer's demand they can perform Smart Grid functions, e.g. calculating bills. Being the blockchain setting permissioned, the blockchain nodes are identified and their operations are verified and endorsed by the entire network.

Figure 4.1 shows an example of the proposed blockchain-based Smart Grid network with 4 prosumers. Smart meters are physically connected to the energy supply (dotted orange lines in the figure) and can wirelessly communicate energy data to SET clients. SET blockchain nodes, forming the blockchain network, are connected to each other wirelessly (dotted blue lines in the figure). For the sake of readability, in Figure 4.1 we

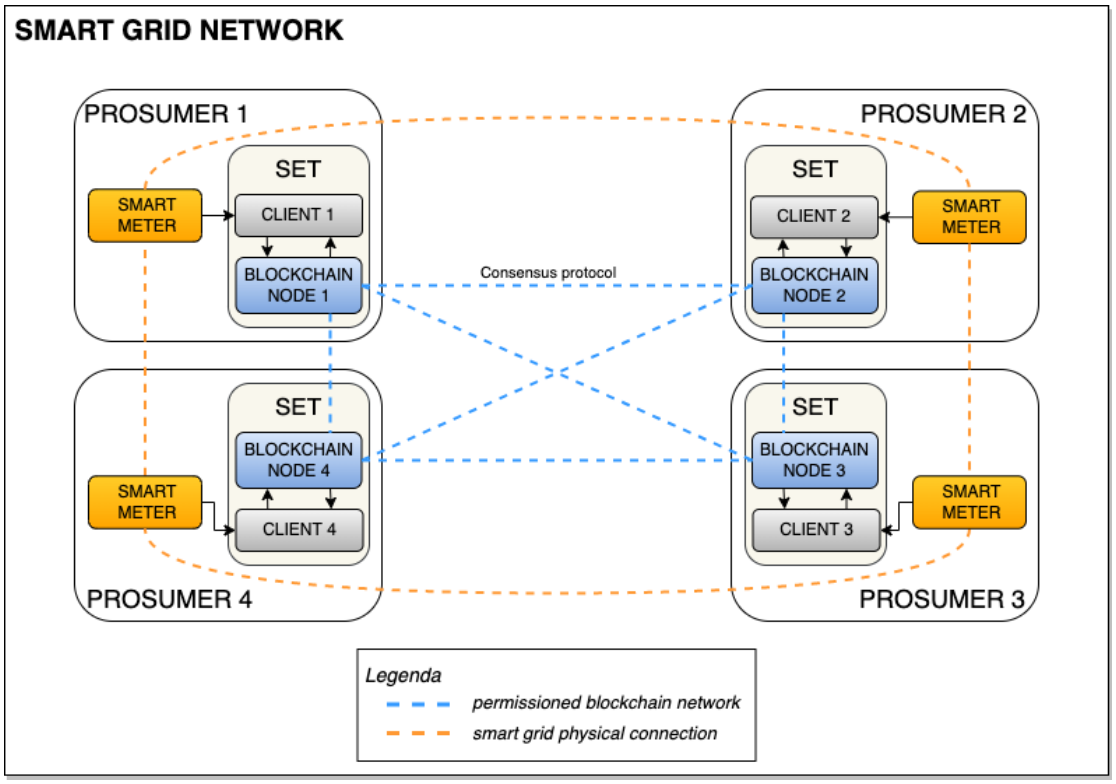


FIGURE 4.1: A blockchain-based Smart Grid network composed by four prosumers

omit to depict the CA and its interactions with SET to release digital certificates to both client and blockchain node. The novel Smart Grid system runs as follows. The SET client of a prosumer listens to the data that the smart meter continuously transmits, and as they arrive, it creates packets to be sent to the blockchain. The client includes data packets into a transaction, which it cryptographically signs with the private key of its digital certificate. Transaction is then sent to blockchain via the appropriate API. For example, as in Figure 4.1, *client-2* listens to the *prosumer-2*'s smart meter. Once collected data, *client-2* prepares a transaction *txn* signed with its private key sk_2. The *txn* is then sent to the blockchain *node-2*, used as a gateway for the blockchain network. As per standard execution of a permissioned blockchain, *node-2* firsts checks that sk_2 is correct and then initiates the consensus for *txn*. Note that the signature check prevents adversaries from impersonating a prosumer and injecting fake data on her/his behalf. Signature verification also applies to smart contracts calls, ensuring that only authorised prosumers can request Smart Grid functions (on certain data).

4.1.2 Prototype implementation on Hyperledger Fabric

The blockchain part of the architecture presented in Section 4.1.1 has been implemented with the Hyperledger Fabric platform. Particularly, we implemented different layers to represent the SET module on Fabric:

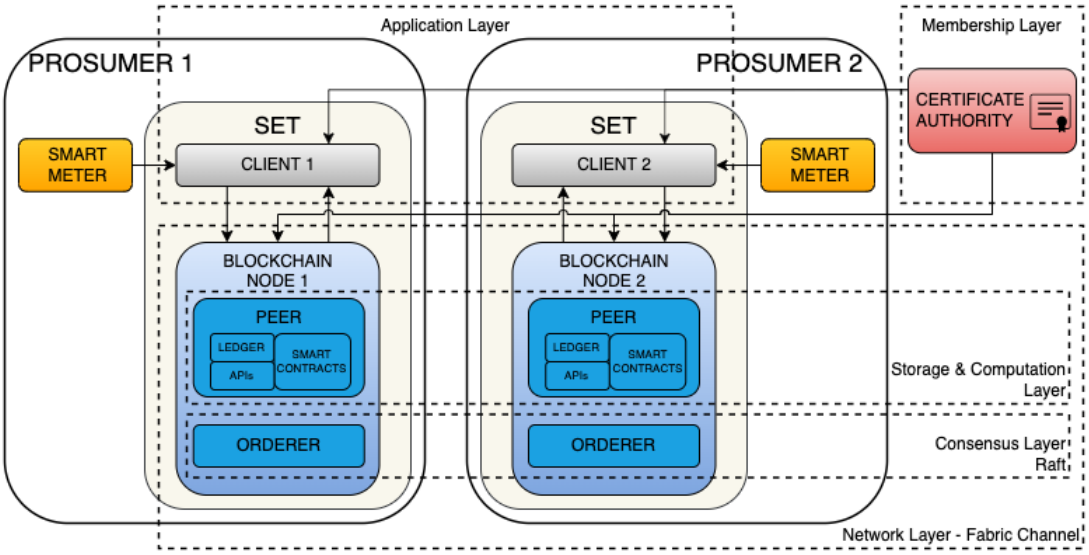


FIGURE 4.2: Hyperledger Fabric architecture layers for the SET module

- *membership layer*: it consists of a CA that issues credentials, in the form of cryptographic keys, to each participant to Fabric network, i.e. clients, peers and orderers. Clients use these credentials to authenticate their transactions, peers to authenticate smart contract executions and endorsements, orderers to authenticate their decisions during consensus. Furthermore, this layer enforces access control rules to authorise or reject client requests.
- *application layer*: It consists of clients that invoke API-requests to either read data from ledger, write to it, or execute functions via smart contract.
- *storage and computation layer* It consists of peers that hold the channel ledger and have smart contracts deployed. It exposes APIs for clients to respectively read/write data and run smart contracts.
- *consensus layer*: It consists of orderers that order transactions into blocks by running the consensus protocol, i.e., Raft (Ongaro and Ousterhout (2014)). The orderers start the consensus by generating the *genesis block*, which contains the channel configuration, i.e., the public credentials of all members involved, the client authorisation rules, and the transaction endorsement policies.
- *network layer*: it is the Fabric channel that connects peers and orderers.

Figure 4.2 shows how two prosumers and their SET modules are disposed over these layers. The membership layer is depicted outside the network because the CA is not a component of the blockchain. Indeed, in Fabric it is a pluggable service, and Fabric offers an in-house CA that we use to issue credentials to SET submodules, i.e. clients and blockchain nodes. The application layer contains the clients that act as brokers

between prosumers and the blockchain. A SET client connects to the network layer using its associated SET blockchain node as gateway. In particular, when a *prosumer-2* needs to send a transaction to the network layer, her SET *client-2* creates the transaction *txn* and sends it to the associated *node-2*. Then, the *node-2* performs two tasks: (i) executes *txn* producing txn_{out}; (ii) broadcasts both *txn* and txn_{out} to other blockchain nodes in the network (e.g., *node-1*, *node-3*, *node-4*). Upon receiving such message, each blockchain node in turn re-executes *txn* and broadcast its txn_{out}. In Fabric, this execute-and-broadcast phase is used to endorse that *txn* of *prosumer-2* is valid and that the majority of peers produce the same txn_{out} (Androulaki et al. (2018)).

We create a channel in Fabric and we activate inside it peers and orderers. As stated in Section 4.1.1, with the aim of delivering *fairness* between prosumers and *democratic control* of the system, we design the SET blockchain node to include both a *peer* and an *orderer*. This enables each prosumer to hold the ledger and smart contracts (as a peer) and to participate in the system's consensus protocol (as a orderer). Consequently, each of them possesses the Smart Grid information, can execute functions, and can validate-and-order the system transactions. Indeed, as Figure 4.2 shows, they both ideally belong to the same blockchain node component. Practically, in Fabric peers and orderers are separated because they have different roles. To solve the problem, we thus create in the Fabric channel as many organisations as prosumers, and each organisation owns only one peer and one orderer. That is, for example, *organisation-3* stands for *prosumer-3*, and owns *peer-3* and *orderer-3*. Then, peers and orderers act on two distinct layers. In the storage and computation layer the peers expose APIs to clients, which are implemented through Fabric built-in SDK. Considering that the blockchain ledger in Fabric is represented as a collection of key-value pairs, the APIs we expose are:

- `read-ledger(k)`: given a key `k`, it returns the associated value `v`. In this use case, a key `k` is a prosumer's unique id within the system, and the associated value `v` is a set of attributes: the prosumer's account balance [3], her list of energy consumptions and her list of bills. Note that the list of energy consumptions is updated by the prosumer by sending data to the blockchain, whereas the account balance and the bills are automatically updated by smart contracts.
- `store-energy(k, e)`: it sends the energy consumptions as a pair `(k, e)`, and it returns a confirmation whether the pair is inserted into the ledger or not. A prosumer use her unique id as `k` to send her energy consumptions in `e`. If the transaction is successful, the attributes of `k` are updated with `e` in the ledger.
- `sc-billing(k, t)`: it invokes a smart contract to calculate the energy bill of `k` for a time-frame `t`, it then stores the result into the ledger and returns a

[3]For the sake of simplicity, in this use case we consider the balance as an integer initialised to 0 and decremented after the bill calculation to show the amount due.

confirmation. A prosumer invokes this smart contract using her unique id and as result the smart contract updates the prosumer's bills list and balance.

As regards the `sc-billing(k, t)` API, we implemented the corresponding smart contract in Go [4] (Google (2009)). Specifically, the smart contract first ascertains who the requesting prosumer is by checking the client signature on the transaction. Then, it enforces access control policies and checks whether the requesting prosumer is the same as the one specified in `k`. If these checks are successful, the smart contract retrieves the energy consumptions of prosumer with id `k` from the ledger over a time-frame `t`. Finally, it calculates the bill by multiplying the consumptions with a pre-agreed tariff [5], hardcoded in the smart contract, and stores the result into ledger by updating the balance and the list of bills of `k`. If, at a later stage, there is a need to change the tariff, Fabric allows the smart contract(s) to be updated with the new tariff and re-installed in the peers.

Note that, we assume each prosumer's SET peer to have the billing smart contract installed, and according to Fabric implementation (Androulaki et al. (2018)), each of them runs the smart contract when an API request `sc-billing(k, t)` arrives. In Fabric, this is related to the endorsement process occurring between peers to verify the smart contract execution. For example, consider the Smart Grid network composed by four prosumers as in Figure 4.1. When the *client-1* calls the API `sc-billing(k, t)`, it is executed by all peers *peer-1*, *peer-2*, *peer-3*, *peer-4*. Once an API is endorsed, its corresponding transaction is handed from the peer to the orderer, to be later included in a block and appended to the ledger. In the consensus layer we use Raft, the default consensus algorithm in Fabric, by which orderers first validate and then order the received transactions. When a consensus is reached, the orderers notify the peers to either commit the block in the ledger or reject transactions. The peers in turn notify clients with the outcomes of API-requests.

4.2 Privacy improvement: combining Blockchain and HE

Privacy is an important requirement for any distributed system, especially for those based on IoT devices, which use their sensing capabilities to gather information from surrounding environment. Because of their limited computational power, IoT devices are not able to perform complex encryption and decryption quickly enough to transmit collected data securely in real-time. This is relevant in Smart Grid, where energy data may inadvertently reveal users daily activities. In Section 4.1 we

[4] Hyperledger Fabric SDK for Go available at: https://pkg.go.dev/github.com/hyperledger/fabric-contract-api-go

[5] In this use case, we assume that there is a single energy supplier who agrees the tariff with prosumers. In a more sophisticated scenario, where there are several energy suppliers, an oracle can be implemented in smart contracts to seek the best tariff on the market.

presented a Smart Grid system that uses a permissioned blockchain as underlying infrastructure to strengthen data security. However, that solution lacks of data privacy: energy data are transmitted in clear from the source, i.e. the smart meter, to the blockchain network where they are stored on the ledger. Besides being exposed to eavesdropping over communication links, energy data are replicated on all blockchain nodes. If one of these becomes compromised, an adversary can analyse the data of all prosumers to discover their habits.

Although on one hand encrypting data solves this privacy deficit, on the other it introduces another problem: smart contracts can no longer carry out data operations. After the conclusion of UKIERI project we deepened this aspect and investigated alternative strategies. We identify in the HE model as the approach that better fits the previous system, without the need to revolutionise it. Indeed, with HE embedded in the system, in the blockchain side, it is possible to simultaneously preserve data privacy and perform computations over encrypted data by means of smart contracts.

With reference to the architecture proposed in Section 4.1.1, we present an upgrade of the SET module that integrates the HE model. Figure 4.3 shows the HE-based SET module. Particularly, we include the HE algorithms *keygen*, *encryption*, *decryption* at SET client, whereas the HE *evaluation* algorithm is included as software library within smart contracts at SET blockchain node. Moreover, compared to previous architecture, we add a *Virtual Private Network (VPN)* tunnel between the smart meter and the client. The communications under VPN are encrypted with per session encryption keys: they are either changed any time a new connection is established or re-generated after a preset timeout. This guarantees the privacy of energy data in transit from the smart meter to the client. If an adversary, whether internal or external to the Smart Grid network, attempts to eavesdrop that communication links, it only obtains ciphertexts.

The *keygen* algorithm generates prosumer's HE keys [6]. The HE private key is held off-chain by the client in a separate (private) database, i.e. outside the blockchain boundaries. Hence, blockchain nodes, both peers and orderers, cannot decrypt data contained inside transactions. The *encryption* algorithm allows the client to encrypt the energy consumptions received by the smart meter. When the client prepares the transaction to be sent with the `store-energy` API, it attaches in its values the prosumer's HE public keys. The corresponding ledger record will then contain the encrypted energy and its related encryption and evaluation keys. This information empowers peers to perform homomorphic computations via smart contracts. The *decryption* algorithm allows the client to decrypt encrypted values it receives as output of a `read-ledger` request. On the peer side instead, the *evaluation* algorithm allows smart contracts to execute functions, e.g. calculate energy bills, with ciphertexts as input. Specifically, any arithmetic function a smart contract has to perform can be converted into *boolean circuits*: a model of computation using logic gates, such as *xor*

[6] Note that the HE keys are distinct from keys issued by the CA within identity certificates.

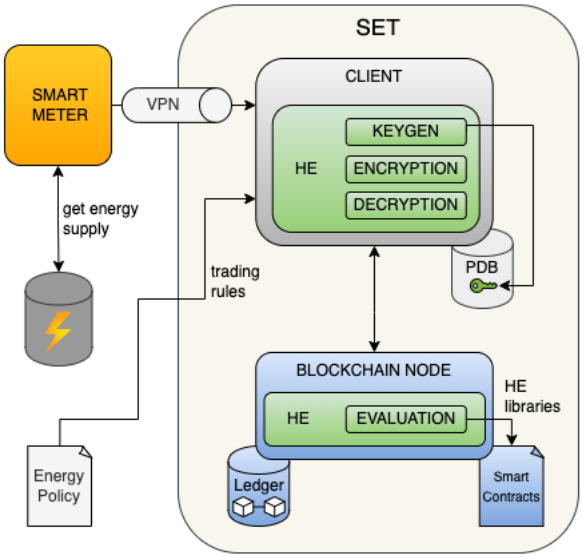


FIGURE 4.3: The SET blockchain module equipped with HE

(i.e., $\oplus$) and *and* (i.e., $\wedge$) on input bits (Vollmer (1999)). For example, the multiplication between the energy consumptions and a pre-agreed tariff to calculate the bill can be realised through $\wedge$ gates. The smart contract can then apply these circuit gates over a set of ciphertexts by relying on their associated pair of encryption and evaluation keys. As described in Section 2.3, we stress that the homomorphic property of these computations only holds for ciphertexts encrypted under the same key. That is, the smart contract cannot run the *evaluation* algorithm with input ciphertexts belonging to different prosumers, and thus encrypted with different encryption keys. As per HE formulation, the circuit output is an encrypted result that will be later stored by the smart contract to the ledger. The prosumer can finally find out its energy bills by requesting the client to retrieve-and-decrypt them from the blockchain ledger.

4.2.1 Overheads introduced by HE

In this section, we report experimental results from the implementation of the aforementioned HE-equipped SET in Hyperledger Fabric. These results show the computational and message overheads introduced by HE in SET. All experiments are performed on a machine with 1 CPU Intel(R) Core(TM) i9-9980HK @ 2.40GHz and 12GB RAM running Ubuntu 22.04 LTS. We use *Go* (Google (2009)) as programming language for developing the SET client and the smart contract of SET peer. As the HE scheme to be integrated in SET, we select the CKKS scheme described in Section 2.3.2. We stress that CKKS works with complex and real numbers, which perfectly suit the Smart Grid use case, and its HE computations return an approximate result. Also, CKKS is a FHE scheme providing both addition and multiplication on ciphertexts. We implement CKKS as a Go module by leveraging the *Lattigo library v2.3.0* Lat (2021).

	Keys generation	Encryption	Evaluation	Decryption
Time	832.87 ms	277.37 ms	21.36 ms	148.05 ms
Size	**sk** 8.90 MB **pk** 17.80 MB **ek** 213.66 MB	$\mathbf{c_f}$ 14.80 MB	$\mathbf{c_e}$ 14.80 MB	

TABLE 4.1: CKKS computational time and data size

This CKKS Go module provides the algorithms for keys generation, encryption, evaluation and decryption. In particular, as shown in Figure 4.3, we include keys generation, encryption and decryption algorithms in SET client, and the evaluation (both addition and multiplication) algorithms in the SET peer's smart contract. According to the homomorphic encryption standard Albrecht et al. (2018), for the CKKS scheme in SET client and SET peer's smart contract, we use the following parameters to achieve the security level 128 bits:

- $\log_2 n = 15$;
- $\log_2 q = 880$;
- $\Delta = 2^{40}$;
- $\sigma = 3.2$.

The key distribution $\mathcal{R}_3$ samples each coefficients from $\{0, \pm 1\}$ with probability 0.25 for each of -1 and 1 and with probability 0.5 for 0. The error distribution $\mathcal{X}_\sigma$ is a discrete Gaussian distribution of variance σ^2.

We simulate a scenario where the SET client collects energy consumptions in real number format, encrypts them with HE public key, and then submits the ciphertexts as a transaction to the SET peer (by calling the API `store-energy(k, e)`). Later on, the SET client invokes the SET peer's smart contract to compute the bill (by calling the API `sc-billing(k, t)`). The smart contract first adds homomorphically the encrypted energy consumptions and then multiplies by the tariff, generating an encrypted result which is sent back to the SET client. Upon the delivery, the SET client decrypts the evaluated ciphertext with the HE private key and finds out its bill.
Table 4.1 reports the computation time of each CKKS algorithm, i.e. keys generation, encryption, evaluation and decryption, and the size of each generated CKKS data, i.e. private key sk, public key pk, evaluation key ek, fresh ciphertext c_f and evaluated ciphertext c_e. Note that the evaluation is composed by addition and multiplication, i.e. $c_e = (\sum_{i=1}^{m} c_{f,i}) \cdot t$, where m is the number of the energy consumptions and t is the plaintext encoding the tariff. The experimental results show that the CKKS algorithms have a computational cost in the order of milliseconds. Specifically, the total overhead introduced by CKKS in the SET client is around 1.25s, whereas in the SET peer's smart

contract is around 21.36ms. The size of CKKS data is in the order of megabytes, among which *ek* is the largest, with a size of 255MB.

4.2.2 Privacy-preserving energy trading

Within the UKIERI project, we restricted the smart contracts only to calculate energy bills according to prosumer's consumptions. The introduction of HE in the blockchain architecture guarantees the privacy of these data, both when they are stored on the ledger and processed by smart contracts. Besides these benefits to energy billing, the HE model opens up the possibility for Smart Grid to privately realise another key functionality: the autonomous energy trading among prosumers. In this section we present how to enrich the Smart Grid with private energy trading by leveraging on the SET module previously presented in Figure 4.3. We propose to establish decentralised auctions via smart contract, where prosumers can buy/sell to each other directly, rather than dealing with other third parties, e.g. energy suppliers and their brokers.

In order to permit smart contracts to deal with bids made by different prosumers we need to make some adjustments to SET. Indeed, as depicted in Figure 4.3, we configure the SET client to include the HE *keygen* algorithm, such that each prosumer running an its instance possesses personal HE keys. This is useful for energy billing: each prosumer can encrypt its consumptions and delegate the computation to smart contracts. However, since the homomorphic property of the HE *evaluation* algorithm holds only for input ciphertexts encrypted under the same key, such approach does not work for energy auctions. If each prosumer encrypts its bid with its personal HE public key, then the smart contract is no longer able to compare bids and determine a winner. To solve the problem we introduce a sort of key-dissemination phase during an auction. Specifically, when a prosumer needs to sell exceeding energy it requests the blockchain (via smart contract) to create an auction and to disseminate its personal HE public key, which will be then used to encrypt the bids by other prosumers. This dissemination does not actually take place via message exchanges, but rather via the ledger. Indeed, the (seller) prosumer invokes a smart contract, which sets up the auction and stores its attributes to the ledger, including the seller's HE public key. In particular, we assume that the smart contract generates a unique id for the auction, and also stores in the ledger its amount of energy, its suggested price and the auction deadline. That is, we represent an energy auction in the ledger with the key-value pair `(k, v)`, where `k` is the auction unique id and $\texttt{v} = \{\texttt{q, p, pk, t, bs}\}$, where `q` is the energy quantity, `p` is the starting price, `pk` is the seller's HE public key, `t` is the deadline and `bs` is the list of bids collected until `t`. When a (bidder) prosumer is interested in joining an auction, she/he retrieves the associated HE key from the ledger, encrypts her bid and sends it to the blockchain. Upon the predefined timeout,

the auction is closed and the smart contract automatically calculate the winner. We therefore expose four new APIs at SET peer:

- `store-plus-energy(k, e)`: it sends the amount of exceeding energy as a pair `(k, e)`, and it returns a confirmation whether the pair is inserted into the ledger or not. A prosumer use her unique id as `k` to send her exceeding energy in `e`.
- `sc-create-auction(q, p, pk, t)`: it invokes a smart contract to create an auction for an energy quantity `q`, with a starting price `p`, a HE public key `pk` to encrypt bids, and a timer `t`. It creates a unique id `k` for the auction and a void list `bs` to collect the bids, stores the record (`k`, $\mathtt{v} = \{\mathtt{q, p, pk, t, bs}\}$) into the ledger and returns a confirmation containing the pair (`k`, `v`).
- `get-auctions()`: it returns a list of open auctions ($\mathtt{k}_i$, $\mathtt{v}_i$).
- `sc-bidding(k, p)`: it invokes a smart contract to make a bid with price `p` for an auction `k`. It stores the bid into `bs` of `k`, and returns a confirmation.

We assume that the smart meter sends the amount of exceeding energy `e` to the SET client, which in turn calls a `store-plus-energy(k, e)` API to store such quantity into the ledger. The API `sc-create-auction()` is called by a seller, while the APIs `get-auctions()` and `sc-bidding(k, p)` are called by possible buyers. The energy trading smart contract handles the APIs `sc-create-auction()` and `sc-bidding()` with two separate threads. Before running both threads, the smart contract verifies the requester identity and controls whether it is allowed to sell or buy. These latter checks are made against the ledger, and consist of ensuring that sellers and buyers have enough energy and money [7] respectively. We assume that for each prosumer the ledger records the balance of her account. Then, the smart contract concurrently runs the two threads. When it creates the auction, it stores the corresponding attributes into the ledger and starts the timer `t`. Upon the timeout, the smart contract retrieves from ledger the auction bids `bs`, finds the maximum bid with the HE *evaluation* algorithm, elects the associated prosumer as auction winner and finally stores the outcome on the ledger. Whereas, when it receives a bid from a prosumer, it retrieves from ledger the auction record and updates the list of its bids `bs` with the one received.

Moreover, as Figure 4.3 shows, the prosumer can specify her/his energy policy in an ad-hoc file, which defines the set of rules on the way energy has to be traded, e.g. which price constraints have to be complied with. These rules may vary during daily hours: set a cheap starting price for night auctions, when energy purchase has a lower

[7] Hyperledger Fabric does not provide a native token (i.e., a crypto-currency) for its platform, thus we do not refer to money as crypto-currencies and to account balance as traditional blockchain wallet containing crypto-currencies. However, Fabric allows an asset to be represented as either a fungible or non-fungible token. For simplicity, in this use case we consider transferred money and prosumer balance as integers, and we note that this implementation can be extended straightforwardly by creating a fungible token representing the application's money.

priority. The energy policy file is used by the SET client to tune the API parameters of both `sc-create-auction()` and `sc-bidding()`. Note that the prosumer's energy policy is stored off-chain. A standard energy trading proceeds as follows:

1. The prosumer's smart meter regularly monitors the on-site energy supply and sends to the SET client, via VPN connections, the quantity of energy available to be traded;
2. The SET client executes two listeners:
   2.1. A listener that receives data from the smart meter. Upon the delivery of exceeding energy, the client processes the prosumer's policy against the current conditions, and automatically decides the quantity of energy to sell `q`, its minimum sale price `p` and the auction deadline `t`. The client calls the `sc-create-auction(q, p, pk, t)` API by attaching as parameter its HE public key `pk`.
   2.2. A listener that constantly queries the ledger for open auctions. The client calls the `get-auctions()` API and displays to the prosumer the list of auctions and their associated parameters. The client processes the prosumer's policy against the list of auctions to automatically make a bid for an auction `k`. Alternatively, the prosumer can pick from the list. The client encrypts the bid with the auction key `pk` via HE *encryption*. Then, it calls the `sc-bidding(k, p)` API with the encrypted bid `p` as parameter.
3. When an auction `k` reaches its timeout `t`, the smart contract first calculates the maximum bid and the winning prosumer via HE *evaluation*. Then, it stores the auction result into the ledger.
4. Each SET client participating in an auction `k`, at the end of its deadline `t`, calls the `read-ledger(k)` API to find out who is the winner, and finally notifies the corresponding prosumer.

After winning the auction, the prosumer pays the bid amount to the energy seller, which in turn transfers the auctioned energy to it. We assume that these steps take place outside the blockchain. Nevertheless, this energy trading application can be expanded by enabling the blockchain to automatically carry out the payments with crypto-coins. Additionally, the SET client can further implement a function that queries the smart meter to ensure that energy has been transferred.

4.2.3 Generalisation for other IoT-based scenarios

As a fringe contribution, we present how the architecture combining blockchain and HE of Section 4.2 for Smart Grid can be generalised to other IoT-based scenarios. The

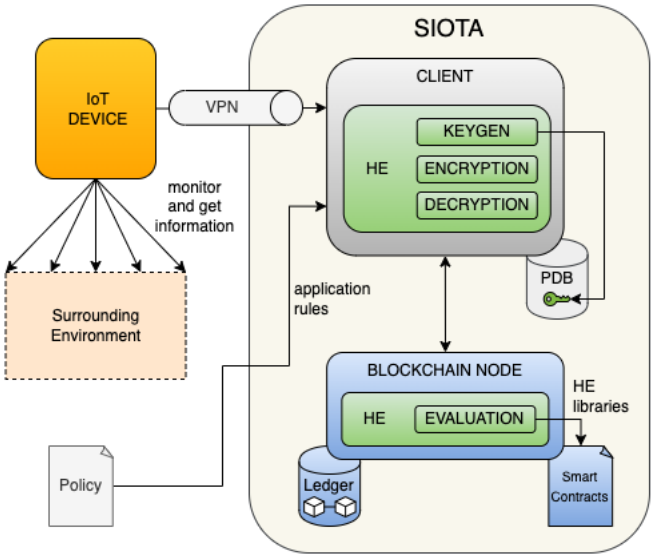


FIGURE 4.4: The SIOTA blockchain module equipped with HE

idea is to maintain the same backbone, with a HE-equipped software client working as intermediary between the on-site IoT device(s) and the blockchain entities. We consider a user hosting a generic IoT device, which automatically monitors and catches information from the surrounding environment. We envisage a general-purpose software module, called *Secure IoT Application (SIOTA)* composed by a client and blockchain node. They both have the same features and duties of those in SET, but are arranged for the IoT application they have to carry out. Indeed, the SET client is only responsible for collecting data from the smart meter, and sending these data to the blockchain to be later processed by smart contracts for billing. The SIOTA client, instead, can additionally query the IoT device to obtain further information from another IoT device nearby. For example, Smart Home systems typically employ various IoT appliances in different rooms, which can communicate between each other. In order to fulfil a specific task, the client may request to one of them to collect data from the others. Moreover, as in the energy auction setting (Section 4.2.2), the SIOTA client enforces user's policies to determine what operations to trigger. As regards the SIOTA blockchain node in contrast, it differs from SET only in terms of smart contract operations, which strictly depend on the IoT scenario.

Figure 4.4 shows the SIOTA module. Similarly to SET, both client and peer contain HE algorithms. Specifically, SIOTA client includes *keygen*, *encryption* and *decryption*, respectively used for generating HE keys, encrypting IoT sensitive data, decrypting ciphertexts. Whereas, the SIOTA peer includes the *evaluation* algorithm as software library within smart contracts code. Furthermore, as in SET, the communications between the IoT device and the client are encrypted inside a VPN tunnel.

4.3 Related work

The promising characteristics of blockchain have caused a lot of interest in energy sector, fostering the realisation of several research works focused on enhancing the security of the emerging Smart Grid and optimising its functions. Authors of papers (Noor et al. (2018); Mannaro et al. (2017); Pipattanasomporn et al. (2018); Hahn et al. (2017); Sabounchi and Wei (2017)), propose to leverage on smart contract to trade energy and create auctions among prosumers. Noor et al. (2018) present a blockchain-based model for micro-grid to align prosumers supply/demand and stretch capacity limits of the main-grid, so to reduce the peak-to-average ratio and to maximise benefits for both prosumers and energy supplier. The blockchain is in charge of verifying the identities of prosumers, handling the trading process via smart contracts and endorsing the energy payments. Mannaro et al. (2017) propose a system where robot-advisors suggest the best selling strategy to prosumers, and the smart contracts are used to automatically transact energy and cryptocurrency at suggested price. Pipattanasomporn et al. (2018) use Hyperledger Fabric to authenticate prosumers within an isolated micro-grid and to manage via smart contracts in place energy trading. Hahn et al. (2017), instead, propose a Smart Grid featured by Ethereum. Their proposed system supports the energy trading among prosumers by means of smart contracts implementing a Vickrey auction[8]. Likewise, Sabounchi and Wei (2017) use Ethereum to realise energy auctions via smart contracts for prosumers of a micro-grid. Authors of papers (Kounelis et al. (2017); Luo et al. (2018)) instead exploit the blockchain either as secure data storage or as parallel tool for payments between promusers. Kounelis et al. (2017) propose Helios, where solar energy can be produced by prosumers and stored in next-generation batteries. They use Ethereum smart contracts to distribute the exceeding generated energy and receive coins as reward. Luo et al. (2018) exploit blockchain in their proposed platform as a distributed database to securely store the energy trading transactions, which are formed in a multi-agent system designed to support prosumers network.

Although all the mentioned papers propose blockchain to streamline the energy trading process and to enhance the Smart Grid security, none of them considers the privacy aspect. Laszka et al. (2017) describe a solution to maintain prosumers anonymity and privacy while trading energy. They provide an anonymous communication service by combining the onion routing[9] technique with blockchain anonymous addresses assigned to prosumers. Micro-grid members then cannot identify by which prosumer the energy was produced or consumed, and in case of a trading who the stakeholders actually are. Likewise, Aitzhan and Svetinovic (2016) introduce a platform that uses blockchain along with multi-signatures and

[8] In Vickrey auction, the highest bidder wins, but the price paid by the winner is second-highest bid.
[9] Onion routing is a technique for anonymous communications, in which messages are encapsulated in bundles of encryption.

anonymous encrypted messaging streams. The system enables prosumers to anonymously negotiate energy prices and securely transact energy. Furthermore, three are two recent papers that propose to combine blockchain with HE for providing privacy. Singh et al. (2021) propose a Smart Grid where energy consumptions of users are encrypted and aggregated via HE, and the ciphertexts are stored in a blockchain deployed on a cloud. Instead, Li et al. (2022) propose a blockchain-based energy market where prosumers negotiate energy prices. The authors use HE, smart contract and ZKP the create a secure two-party protocol between two prosumers negotiating a price to withstand against the collusion attack.

Both papers (Laszka et al. (2017); Aitzhan and Svetinovic (2016)) propose solutions to deliver prosumer anonymity and also privacy of the energy data. However, the blockchain is merely used in their system to assign users with crypto-addresses and to securely store encrypted deals. Particularly, the trading negotiations, concerning the quantity of exchanged energy and its price, take place outside the blockchain network, without using smart contracts. Compared to this current state-of-the-art, although our solution does not provide users anonimity, it uses the blockchain to decentralise, automatise and streamline the Smart Grid functions of energy billing and trading, removing either energy brokers or consultants. Additionally, due to the integration of HE model, our solution executes these functions by guaranteeing the privacy of data while are processed by smart contracts. Conversely, Singh et al. (2021) and Li et al. (2022) actively use blockchain and smart contracts to perform operations in a Smart Grid, i.e. aggregating consumptions and negotiating prices respectively. Also, they both integrate HE in these processes. Nevertheless, our work present some differences with Singh et al. (2021) and Li et al. (2022). Differently from Singh et al. (2021), we use HE for energy billing and trading, so we focus on the prosumer and the benefits it can obtain from these decentralised functions. We note, however, that our approach is similar: prosumers sends data encrypted via HE to blockchain, which performs HE computations and store the result in the ledger. Speaking of approach, ours is completely different from that of Li et al. (2022). They do not implement an energy trading, but a two-party negotiation protocol for the energy tariff via smart contract, which is secured via HE and ZKP. The purpose of this protocol is to defend against the collusion attack of neighbour prosumers. To the best of our knowledge, we believe this one of the first work that integrates blockchain and HE into Smart Grid environment to perform decentralised and privacy-preserving energy billing and trading.

4.4 Discussion

In this chapter we presented a novel Smart Grid system featured by a permissioned blockchain that we developed for CS-SED project of the UKIERI research programme.

This integration is motivated by the several benefits that blockchain brings to Smart Grid. Firstly, blockchain used as underlying data storage guarantees strong integrity and non-repudiation properties to data collected by IoT devices, i.e. the smart meters. Secondly, its built-in smart contracts eliminate the need for a central authority, i.e. the energy supplier, to coordinate the Smart Grid and verify the accuracy of energy invoices. Thirdly, due to the permissioned setting, the blockchain enforces authentication and authorisation checks that enhance the trust among prosumers participating in the system. We therefore presented the architecture that connects the Smart Grid prosumers with an overlay network of blockchain nodes. We proposed a new software module, named SET, that acts as intermediary between the prosumer's smart meter and the blockchain. Specifically, we implemented the SET module in Hyperledger Fabric, a permissioned blockchain platform offering authentication and access control services. We designed SET to be composed by a client and a blockchain node. The former is used to collect energy consumptions from the smart meter and to invoke blockchain functions. Whereas the latter maintains a copy of the ledger and runs smart contracts that automatically calculate the energy bills.

Although this blockchain-based solution strengthens the security of Smart Grid and streamlines energy billing, it lacks privacy. Indeed, the energy consumptions, as well as energy bills, are both transmitted and saved on the ledger as plaintexts. This exposes them to be eavesdropped by an external adversary or maliciously inspected by a compromised blockchain node. Alongside it, the simplistic encryption of energy consumptions would prohibit smart contracts from calculating bills on them. Therefore, we upgraded the previous system by combining the blockchain with HE. The new HE-equipped SET module is then able to simultaneously encrypt the consumptions and calculate the bills on their ciphertexts using the blockchain. The results of our experimental evaluation show that the overhead introduced by HE is low in SET client (approximately 1s), and particularly low in SET peer's smart contract (approximately 20ms), demonstrating that the HE-equipped version performs as well as the one without HE.

Furthermore, we enriched the SET module with HE-based energy trading operations. In particular, we empowered SET to create decentralised privacy-preserving auctions via smart contracts when prosumers have surplus energy to sell. The auction is conducted over encrypted bids by relying on the HE properties. Summarising, the permissioned blockchain coupled with HE hence brings the following benefits to Smart Grid:

- It makes the overall Smart Grid system more cost-effective by enabling autonomous energy billing and decentralised energy trading. The energy supplier no longer has to deal with billing, while prosumers can trade energy among themselves without paying brokers or consultants.

- It enhances the Smart Grid security and prosumers trust, by relying on the security properties offered by the combination of blockchain and HE. Prosumers are authenticated and access control rules are in force. Energy consumptions, bills and auction details become both immutable and private. Energy billing and trading turn into privacy-preserving operations.

Considering these advantages, we then proposed SIOTA, a generalisation of HE-equipped SET applicable to other IoT-based scenarios. Notwithstanding, HE poses a limitation that hinders the application of SIOTA in IoT systems geared towards performing users' statistics. Schemes belonging to HE are typically referred to as *single-key*, because they only permit computations on data encrypted under the same key. The SIOTA module, as well as SET, is designed to equip the user with a single-key HE scheme. Each user can then generate personal HE keys, encrypt, decrypt or delegate the blockchain to evaluate on its encrypted data. As a result, the blockchain ledger contains ciphertexts encrypted with distinct HE keys. If the IoT system is formed with the objective of pooling and analysing users' data, the different single-key schemes at play prevent SIOTA from accomplishing it.

In the next chapter we present a solution to this problem. In particular, we present a novel permissioned blockchain integrated with *Multi-Key HE*, an innovative HE model that allows computations on data encrypted under different keys.

Chapter 5

PANTHER: A privacy-preserving permissioned Blockchain integrated with Multi-Key Homomorphic Encryption

The blockchain technology has brought significant advances in terms of integrity and availability of data, indeed an adversary would require extensive resources to tamper with or prevent access to the transactions stored in a blockchain. A blockchain can also perform computations through smart contracts, which define the code to execute as a transaction and, therefore, enjoy the same integrity and availability properties. Another important feature of blockchain concerns data transparency, indeed stored transactions can be read by any user having access to a node of a blockchain network. Permissionless blockchains such as Bitcoin and Ethereum, where any device is allowed to become a node of the network, provide the highest degree of transparency. Although desirable in some contexts, several scenarios require some level of privacy, for example when stored data is confidential. This aspect is partially addressed by permissioned blockchains, where only a fixed set of nodes can be part of the network, and access control rules can be enforced to regulate who can access what data. However, stored transactions are still replicated over all the blockchain nodes, and high privilege users of these machines can access this data anyway. For example, Hyperledger Fabric enforces policies [1] to govern user authorisations on ledger data, but among users, network admins always have read permissions. Fabric also introduces the concept of private data [2] to further restrict the data access to subgroups within the same network: the ledger contains only the hashes of the data, and the actual content its available to subgroups according to pre-agreed policies. Yet, all

[1] https://hlf.readthedocs.io/en/latest/policies/policies.html
[2] https://hlf.readthedocs.io/en/latest/private-data/private-data.html

users of a subgroup can access the data content. An effective solution to ensure confidentiality of stored transactions consists in encrypting them before submission to the blockchain. The downside of this approach relies in the inability to perform any smart contract computation on ciphertexts. For example, in a blockchain-based healthcare system where patients' data is stored encrypted, doctors could not rely on smart contracts to carry out required medical assessments, such as calculating the monthly average of blood glucose in patients suffering from diabetes.

That limitation can be overcome using *Homomorphic Encryption (HE)*. HE is a cryptographic tool to execute functions on cyphertexts and generate encrypted results, which, once decrypted, are equal to the results that would have been obtained by executing the functions on plaintexts. Integrating HE in a blockchain can ensure the privacy of stored transactions without preventing the possibility to carry out smart contract computations over them. However, traditional HE schemes are single-key, which means that computations are correct only if performed on data encrypted under the same key. Therefore, running smart contract computations on transactions submitted by different users and encrypted using different keys would produce incorrect results. With reference to the blockchain-based healthcare system example mentioned before, it would not be possible to run statistical analyses on medical data of a set of patients.

To solve this problem, in this chapter we propose to integrate a permissioned blockchain with *Multi-Key Homomorphic Encryption* (*MKHE*), a recent HE model that enables computations on data encrypted under different encryption keys. We detail how this integration can be realised by presenting the design of *PANTHER* (*Permissioned blockchAiN mulTikey Homomorphic EncRyption*), which allows a user to privately store her data on the blockchain, and yet take full advantage of smart contracts to execute functions even with data encrypted by others. To decrypt the result of a MKHE computation, a *Secure Multiparty Computation* (*MPC*) protocol is set up in PANTHER, involving only those parties that contributed to the computation with an input ciphertext. Specifically, during a MPC decryption protocol, each involved party sends its partial decryption to the others, and by combining them the result of the computation can be decrypted.

Contributions

The novel contribution of this work consists in proposing to integrate MKHE with a permissioned blockchain, since all existing related works are based on single-key HE. Furthermore, we present a novel MPC decryption protocol for MKHE. Compared to existing MKHE models, which rely on complex cryptographic procedures to deliver decryption correctness against a byzantine adversary, our protocol leverages

blockchain and its data integrity property. Indeed, in PANTHER, a MKHE computation is performed by blockchain nodes via smart contracts and, only if a majority of them compute it correctly, is the result added to the ledger. In our MPC protocol, the parties then verify that the ciphertext to be decrypted is consistent with the one stored in the ledger, which can be considered immutable and, thus, trusted to be correct. The other contributions provided are:

- a comprehensive discussion of the current blockchain solutions coupled with HE, where limitations are outlined;
- the architecture and protocols of PANTHER to enable data privacy and private operations via smart contracts;
- a qualitative message complexity comparison of the proposed MPC protocol with the state of the art;
- an extensive security analysis of PANTHER, where we prove that privacy, integrity and availability of PANTHER's data are preserved against a byzantine adversary;
- the implementation of MK-CKKS, a MKHE scheme based on CKKS (Cheon et al. (2017)) and written in Go that provides primitives to carry out MPC decryption protocols;
- a benchmark of PANTHER's performance, where experimental results show that MKHE introduces a small overhead into the blockchain.

Chapter structure

Section 5.1 presents a literature review on the integration between blockchain and HE, showing current advancements and open challenges in this research field. In Section 5.2 and Section 5.3, we define the blockchain system model and the adversary model, respectively. Section 5.4 introduces the requirements to meet to preserve the privacy of users' data. Then, in Section 5.5, we highlight the downsides of the current solutions that rely on single-key HE and threshold HE schemes (Section 5.5.1), and we propose our privacy-preserving blockchain architecture, which integrates MKHE into different layers of a permissioned blockchain (Section 5.5.2). Section 5.6 details the protocols run by clients and blockchain nodes. A qualitative performance analysis of the proposed MPC decryption protocol is discussed in Section 5.7. A rigorous security analyses of PANTHER is provided in Section 5.8. The implementation of PANTHER is outlined in Section 5.9, along with its benchmark. Finally, Section 5.10 concludes and discusses the work.

5.1 Related work

According to recent surveys (Feng et al. (2019); Bernal Bernabe et al. (2019); Zhang et al. (2019)), the HE model has been identified as one of the main methodologies to cope with the lack of data privacy of current blockchain systems. As claimed by Feng et al. (2019), the application of HE within blockchain offers the opportunity to perform privacy-preserving operations over data recorded on ledger, even though they have previously been encrypted. This peculiarity prompted the realisation of several academic works based on HE and blockchain in heterogeneous scenarios, e.g. e-voting, supply-chain, IoT systems.

Du et al. (2020a) propose a new supply-chain platform on Hyperledger Fabric which embeds the Paillier cryptosystem into smart contracts to encrypt trade data (e.g., prices, quantities) and to perform on them addition operations (e.g., add/remove goods quantities). She et al. (2019) propose a framework on Hyperledger Fabric to manage the users' data collected by the IoT devices in smart home environments. In this blockchain system the authors use the Paillier cryptosystem to encrypt the sensitive data of users and perform on them analysis. Similarly, Shen et al. (2019) propose a framework for performing machine learning analysis on data collected by IoT devices in smart city environments. Specifically, they employ the Paillier cryptosystem as privacy-preserving building block of the machine learning compiler, and a blockchain ledger as tamper-proof data storage.

Moreover, in literature can be found some work where HE comes with *Zero-Knowledge-Proof (ZKP)* mechanism. Formulated in 1989 by Goldwasser et al. (1989), the ZKP allows a party possessing a secret information to prove other parties, using a statement, that he/she knows it without revealing any knowledge of the information itself. Such functionality can be useful to enforce the verifiability of HE-based operations in contexts where a certain level of transparency is desirable alongside confidentiality. Ma et al. (2021) propose an abstract framework based on Ethereum for supply-chain systems, where a PHE cryptosystem is used to encrypt the transactions content and to update the accounts balances. In addition, they create two statements with ZKP to show that encrypted buyer's balance is greater than dealt encrypted amount, and that the encrypted balances of both buyer and seller are updated accordingly to transferred money amount. Killer et al. (2020) propose Provotum, an e-voting Ethereum framework that employs the ElGamal cryptosystem to encrypt users' votes and execute the tallying via smart contract. The ZKP is used to demonstrate the correctness of both voting and tallying phases. Notably, the authors use ElGamal under the *Threshold HE (THE)* model, first-time defined by Cramer et al. (2001), and combine via a MPC protocol the public keys of users to produce a system public key. As in MKHE, also in THE a MPC protocol occurs among users to decrypt. Yu et al. (2018) implement an e-voting system on Hyperledger Fabric where the

Paillier cryptosystem is embedded inside smart contract logics as software libraries. Indeed, they perform the tallying of encrypted votes via smart contract and, in addition, they employ the ZKP to demonstrate the correctness of voting procedure. Similarly, Yang et al. (2020) propose an approach that merges ElGamal with the group-based encryption to encrypt users' votes of an e-voting system. Differently from Killer et al. (2020) and Yu et al. (2018), Yang et al. (2020) use the blockchain as database to store the votes and perform the tallying phase without leveraging on smart contract.

Differently from papers mentioned so far, Mitani and Otsuka (2020) propose a protocol featured by a FHE scheme, which enables users of a permissioned blockchain to encrypt their transactions and update their balances according to the encrypted assets exchanged. The authors use ZKP to prove that the correctness and traceability of these encrypted data can be verified from users of a permissionless blockchain. Table 5.1 shows a comparison of the aforementioned papers, conducted in terms of:

- Which kind of HE model is used;
- Whether or not smart contracts implement HE-based computations;
- Whether or not the ZKP is used along with HE to strengthen the truthfulness of encrypted data;
- Which kind of blockchain setting is considered, i.e. permissionless or permissioned.

The first aspect to note in Table 5.1 is that all papers adopt single-key HE schemes within their respective blockchain environment. In the single-key PHE and FHE models the correctness property 2.5, and its inherent homomorphism, does not hold for data encrypted with different unrelated keys. This is the case of papers (Ma et al. (2021); Mitani and Otsuka (2020)). In both systems a user is equipped with a personal pair of HE-keys. Although she can execute the HE-based computations on her own data, encrypted with her public key, she cannot execute it on data of different users. Thus, users are not allowed to combine their data and perform functions on them, restricting the advantages of HE. Attempting to overcome such limitation, and enable computations on joint data of different users, the other cited papers propose various approaches. Papers (Du et al. (2020a); She et al. (2019); Shen et al. (2019); Yu et al. (2018)) employ only one public key for the entire system, while the matching private key is retained by an administrator in charge of evaluating data. Similarly, in Yang et al. (2020), system admins distribute to users a small set of public keys associated with candidates of an election, and retain the private keys. Users encrypt their votes and the admins perform HE-based computations to count the votes of each candidate. However, with this approach users have to trust the admin distributing the system

Papers	HE model used	HE private computations into smart contract	HE with ZKP	Blockchain setting
Du et al. (2020a)	PHE	Yes	No	Permissioned
Killer et al. (2020)	PHE - THE	Yes	Yes	Permissioned
Ma et al. (2021)	PHE	No	Yes	Permissionless
Mitani and Otsuka (2020)	FHE	No	Yes	Both
She et al. (2019)	PHE	No	No	Permissioned
Shen et al. (2019)	PHE	No	No	Permissionless
Yang et al. (2020)	PHE	No	Yes	Permissionless
Yu et al. (2018)	PHE	Yes	Yes	Permissioned
This work	MKHE	Yes	No	Permissioned

TABLE 5.1: Work combining blockchain and HE

public key, who may behave dishonestly. Moreover, users are passive. Not having the decryption key, they cannot autonomously execute HE-based computations and are merely restricted to feed the system with their data. Another interesting approach is applied by Killer et al. (2020). The authors employ THE model, and aggregate the individual public keys of users into a single system public key. However, as detailed and further elaborated in Section 5.5.1, THE model requires users to run a MPC protocol to create the common public key, which implies that all of them need to be always online. In the light of these considerations we decided to adopt the MKHE model as a solution to these drawbacks.

The second aspect to note is that, among the cited work, only the papers Du et al. (2020a); Killer et al. (2020) and Yu et al. (2018) provide smart contracts executing privacy-preserving computations via HE. By contrast, all other papers opt to run the HE computations outside the blockchain, and then to just store the results of its executions into the ledger. This limits the blockchain to merely operate as a database and turns out to not be a true integration between blockchain and HE. Thus, similarly to Du et al. (2020a); Killer et al. (2020) and Yu et al. (2018), we conceive an architecture in which the HE computations are implemented straight through smart contracts, enabling users to delegate their desired computations to blockchain while preserving the privacy of involved data.

Another aspect emerged from this literature review is the exploitation of ZKP along with HE. The ZKP has shown to be very useful in different use cases (Ma et al. (2021); Killer et al. (2020); Yu et al. (2018); Yang et al. (2020); Mitani and Otsuka (2020)) to prove some statement about the data encrypted via HE. For instance, Ma et al. (2021) use ZKP to allow a buyer to prove that her encrypted account balance is greater than a certain purchase price. Considering ZKP characteristics and drawbacks, we opt to not include it in our architecture for two reasons. Firstly, the functionality of ZKP is worthwhile just in particular use cases, where some transparency about secret data is needed. Recalling the paper Ma et al. (2021), the statement created via ZKP releases an extra detail about the account balance, i.e. that it is above a certain amount. Conversely, our aim is to provide privacy for blockchain data without releasing any

additional information about them, i.e. without any degree of transparency. Secondly, relying on complex cryptographic procedures to create the proofs, ZKP requires a significant computing power and it is time-consuming. Thus, employing ZKP to prove the correctness of computations performed via HE, as in papers Killer et al. (2020); Yang et al. (2020); Yu et al. (2018), means further contributing to slightly worsen the system performance. Conversely, we rely on the dependability and non-repudiability of smart contracts executed on the blockchain to prove the correctness of HE computations. Indeed, when a HE computation is performed by blockchain, it is only approved and committed into the ledger only if a majority of network nodes (tipically 2/3) validate its execution of the smart contract. Notwithstanding, we leave the door open to apply in future ZKP on top of our architecture in order to release some degree of transparency.

Table 5.1 reports also which kind of blockchain setting is chosen in each cited work. Papers (Shen et al. (2019); Ma et al. (2021); Yang et al. (2020)) employ a permissionless blockchain, whereas papers (Du et al. (2020a); She et al. (2019); Killer et al. (2020); Yu et al. (2018)) employ a permissioned. It is worth noting the work of Mitani and Otsuka (2020) where both settings are employed, i.e. the transactions of a permissioned blockchain are privately recorded in a permissionless blockchain.

To sum up, the integration between blockchain and the HE model is at early stage, and current academic work seek to accomplish this by focusing on single-key HE schemes. To the best of our knowledge, we believe this is the first work that integrates the blockchain with a multi-key HE scheme, capable of performing computations even on data encrypted with different keys. Furthermore, this work is one of the few that proposes to implement a HE scheme via smart contracts, preserving the privacy of processed data and enhancing the blockchain security.

5.2 System model

We consider a *permissioned* blockchain system composed by multiple organizations, each of which supplies at least one node to set up a private network. We refer to the network nodes as *peers*. In permissioned blockchain context, peers are identified and each of them has assigned a publicly available *digital certificate*, which is released by a trusted *Certificate Authority (CA)* and contains a set of verifiable attributes relating to its holder, e.g. its hostname and the organization it belongs to. Additionally, each ith peer in the network has a set of cryptographic credentials, namely a pair of public-private keys (Cpk_i, Csk_i). The public key is included as attribute in peer's digital certificate. The private key is used to digitally sign transactions processed by a peer. We also consider that each peer holds a local copy of the *ledger*, where are recorded all transactions occurred in the system, and is endowed with smart contracts

to perform computations on data. We refer to peers that execute smart contracts as *computing peers*. We assume asynchronous communication between peers to validate a transaction, instead we rely on partially synchronous communication between those peers engaged in the consensus protocol, as per PBFT system model Castro and Liskov (1999), to guarantee its liveness property.

We refer to members of organizations as *users*. Analogously to peers, each jth user owns a certificate and a pair of cryptographic keys (Cpk_j, Csk_j), which she can use to digitally sign the transactions she sends to the blockchain. Depending on the role a user covers within her organization, she can have different rights and permissions for carrying out operations with the blockchain. These authorisations are associated with the user's certificate and are stated in the blockchain configuration. Each user can interact with the blockchain network through an application software (e.g., web or mobile app), for submitting transactions and retrieving data from the ledger. We refer to it as *client*. We assume asynchronous communication between clients. Also, we assume that a client sends asynchronous and uniquely indexed requests to peers, and that peers eventually reply to a request (i.e., partially synchronously).

We assume that every pair of system parties is connected by a bidirectional link, and that these point-to-point links are reliable in the face of crashing parties, i.e. a correct party eventually delivers a message sent to it by another correct party (Cachin et al. (2011)). Links reliability implies that messages are continuously retransmitted until reach their destination, and network partitions will be eventually solved. Thus, even in the case of loss of some messages due to a temporary link failure, the probability for a message to reach its destination is non zero. We assume parties use reliable broadcast primitives for one-to-many communications, i.e. if a correct party delivers a message then eventually all correct parties deliver it (Cachin et al. (2011)). Furthermore, we assume that the communication channels per se do not provide privacy, i.e. they do not apply cryptographic methods over carried data.

5.3 Adversary model

Bearing in mind the system model described in Section 5.2, there exist multiple potential attacks to which participating actors might be exposed. The aim of our system is to defend by-design against these threats by leveraging both blockchain and HE security strengths. We thus define the capabilities of an adversary willing to undermine the privacy, integrity or availability of the system data. We consider computationally bounded adversaries, i.e. PPT adversaries. According to the network assumptions made in Section 5.2, we consider that even if an adversary drops some retransmissions of a message, it cannot drop them all and eventually the intended recipient delivers the message. Formally:

Definition 5.1 (Byzantine blockchain adversary)**.** Let S be a permissioned blockchain system, composed by N clients and M peers. Each jth client holds a MKHE scheme $\mathcal{E}_j$ and runs MPC protocols with other clients to decrypt evaluated ciphertexts. Peers are equipped with an EVAL algorithm by which they can perform computations on ciphertexts via smart contract. Then, a PPT adversary $\mathcal{A}$ attacking S can corrupt either clients or peers, and control them with a behaviour byzantine (a.k.a. malicious or active), i.e. the corrupted parties can deviate from their prescribed protocol. If $\mathcal{A}$ is in control of a party, whether client or peer, can send incorrect data. If $\mathcal{A}$ controls a client can collude with other corrupted clients to subvert a MPC decryption protocol. Whereas, if $\mathcal{A}$ controls a peer (i) can tamper with data of received transactions or stored in its ledger, (ii) can alter a smart contract or snoop on its processed data, (iii) can avoid to process transactions and reply to clients. In addition, $\mathcal{A}$ can attack the communication channels of S by:

- eavesdropping messages among clients and peers;
- tampering with data contained inside messages;
- discarding some messages, but not all.

We assume that the number f_p of byzantine peers is $f_p < |M|/3$, as per PBFT implementation Castro and Liskov (1999). As detailed in Cachin et al. (2011), any set of $\left\lceil \frac{|M|+f_p+1}{2} \right\rceil$ or more peers is a quorum tolerating f_p byzantine peers, and among them there is at least one quorum $\tilde{Q}_p$ entirely composed by honest peers. Furthermore, to guarantee decryption correctness, we require that, for any MPC decryption protocol where $T \subseteq N$ clients participate, the number f_t of byzantine clients is $f_t < \frac{|T|}{3}$. We consider that the set N of clients and the set M of peers are disjoint, and so any set T is disjoint from M. Hence, f_t and f_p are not related. This implies that compromising f_p peers does not give any advantage to an adversary willing to undermine the MPC decryption protocol executed by T. Viceversa, compromising f_t clients gives no advantage to an adversary willing to undermine the blockchain consensus.

5.4 Requirements

In this section we point out the requirements that a permissioned blockchain system should meet in order to enforce the privacy of users' data. We consider that users want to keep confidential their transactions, such that only the stakeholder of a transaction can know the corresponding content. Then, users should employ cryptographic tools to conceal their data in the ledger, which actually by default allows any participant to read all the transactions recorded inside it. We consider also

that a user may outsource some computations to the blockchain on her own confidential data (e.g., compute the average of its monthly expenses), or alternatively with confidential data belonging to other participants (e.g., compare its expenses with those of another user to discover who spends more). Then, the blockchain should perform these computations in a privacy-preserving fashion, which means that only the appropriate users can find out the corresponding result, and also no computing peer can know the actual value of data used in the computation. Hence:

- **R$_1$** No user can find out the content of other users' transactions retrieving data from the ledger;
- **R$_2$** A user can delegate the blockchain, via smart contracts, to perform privacy-preserving computations on her own confidential data;
- **R$_3$** A user can delegate the blockchain, via smart contracts, to perform privacy-preserving computations on joint confidential data, owned by different users.

5.5 PANTHER architecture

This section presents the architecture of PANTHER by first justifying the choice of using an MKHE scheme rather than a single-key one (Section 5.5.1) and, then, detailing the system components and their interfaces (Section 5.5.2).

5.5.1 Design choices

As explained in Section 2.3, the homomorphism property of traditional single-key HE schemes only holds over ciphertexts encrypted under the same key, which makes it hard to meet Requirement R_3. This limitation has been addressed in different ways by the related works discussed in Section 5.1. The majority of works (Du et al. (2020a); She et al. (2019); Shen et al. (2019); Yu et al. (2018); Yang et al. (2020)) provide all users with the same public key without sharing the corresponding private key. However, with this approach users cannot autonomously decrypt the evaluated ciphertexts computed on their data.

Another method consists in applying the THE model, as proposed by Killer et al. (2020), where all users combine their public keys to create a single '*system-level*' public key. The downside of THE is that all the users take part to the initial MPC protocol to generate the system-level public key, therefore all of them are required to be involved for any MPC decryption. Also, issues arise whenever the set of users changes. For example, if a new user joins the system, then it is necessary to run again the MPC

protocol to generate a new system-level public key, re-encrypt the data and perform the computation again. As another example, if a user leaves the system, then no MPC decryption can terminate and, again, a new system-level public key needs to be generated. These restrictions can be mitigated by setting a lower threshold on the number of users required to complete an MPC decryption. The threshold can be decreased at most up to the majority of users, or up to 2/3 of them to tolerate byzantine adversaries. Nevertheless, depending on the total number of users, their majority may still be too large a number, especially in the situation where HE computation is carried out with input data belonging to a small subset of users.

An alternative THE model is to have subset of users who need to execute a function on their data generate a '*function-level*' public key by running an MPC protocol. There are two downsides to this approach. Firstly, an MPC protocol needs to be executed whenever a new set of users want to perform a computation on their data. Secondly, the same data of a user may need to be stored in the blockchain several times, each time encrypted with a different function-level public key.

A dual approach is to split a system-level private key among users via secret sharing, as originally proposed by Cramer et al. (2001), which avoids users running an MPC protocol for generating a shared public key. However, this strategy has three disadvantages. Firstly, all the users need to be involved in the decryption, even if a user just wants to run computations on her own data, as per Requirement R_2. Secondly, a THE scheme like this requires a specific entity to generate the system-level public key, which entails centralisation and, therefore, introduces a single-point-of-failure. Thirdly, issues arise whenever the set of users changes, because the system-level private key needs to be re-shared among the new set of participants.

We decide to adopt the MKHE scheme in PANTHER to overcome the limitations described above and meet R_1, R_2 and R_3. PANTHER differs from the existing approaches in the following aspects:

- each user has their own pair ($HEpk_j$, $HEsk_j$) to decrypt autonomously, as per Requirement R_2, and collectively via MPC protocols, as per R_3;
- the users involved in an MPC decryption for some ciphertext $\tilde{c}$ are only those which contributed their data as input for the computation of $\tilde{c}$;
- users do not have to run MPC generation protocols;
- a user encrypts their data only once using their own $HEpk_j$, thus avoiding redundancy in the ledger;
- whenever a user joins or leaves the system, no new key needs to be generated and distributed.

Adopting MKHE enables having the blockchain peers generate the computation public keys, instead of the users, which streamlines key management and reduces significantly the effort on users. We achieve this in MKHE by employing the *key-switching* tool, which allows extending a ciphertext with another public key. In particular, when peers are requested to run an HE computation on t ciphertexts encrypted with t different public keys, they first combine all the t keys into a single $\tilde{pk}$, then extend each of the t ciphertexts with $\tilde{pk}$. After this pre-processing stage, the peers run the EVAL algorithm on the t ciphertexts to produce a single ciphertext $\tilde{c}$. Later on, the corresponding t users can run an MPC decryption on $\tilde{c}$ to find out the function result.

Moreover, existing HE models with MPC decryption, as in papers Asharov et al. (2012); López-Alt et al. (2012); Mukherjee and Wichs (2016), use ZKP or succinct arguments Bitansky et al. (2012) to prove decryption correctness and achieve security against a byzantine adversary. Specifically, considering a MPC decryption protocol among T parties for decrypting an evaluated ciphertext $\tilde{c}$ produced by a server S, papers Asharov et al. (2012); López-Alt et al. (2012); Mukherjee and Wichs (2016) use such techniques to prove that:

1. the ciphertext $\tilde{c}$ is correct and valid. The server S needs to generate a proof that $\tilde{c}$ is the output of EVAL $= (\mathcal{F}, \vec{c}, pk, ek)$, where $\vec{c}$ are the input ciphertexts of T.
2. partial decryptions of T are well-formed. Each ith party in T needs to generate a proof that the private key sk_i used in its partial decryption of $\tilde{c}$ is consistent with the public key pk encrypting its input c_i for $\tilde{c}$ computation.

For example, the MKHE scheme proposed in López-Alt et al. (2012) uses (i) SNARG and SNARK to generate a proof for the correct execution of EVAL, and (ii) ZKP to generate proofs for the consistency between private and public keys of each party in T. Note that in López-Alt et al. (2012) (as well as Asharov et al. (2012); Mukherjee and Wichs (2016)), each party in T needs to generate a ZKP proof for its private key because all partial decryptions of T are required to decrypt.

Differently from existing HE models, in PANTHER we leverage the blockchain to deliver correctness of MPC decryption in the face of byzantine adversary. Indeed, considering a MPC decryption occurring in PANTHER among T for decrypting $\tilde{c}$, we rely on blockchain properties of data integrity and trustworthiness to ensure that:

1. the evaluated ciphertext $\tilde{c}$, stored on the ledger, has been generated correctly by computing peers.
2. each honest party in T partially decrypts the correct $\tilde{c}$ and broadcasts the result to the correct set T.

Firstly, in PANTHER each peer executes $\text{EVAL} = (\mathcal{F}, \vec{c}, \vec{pk}, \vec{ek})$ via smart contract, where $\vec{c}$, $\vec{pk}$ and $\vec{ek}$ are the input ciphertexts, public and evaluation keys of T respectively. Then, each peer broadcasts to others the result of this computation, i.e., the evaluated ciphertext $\tilde{c}$, for endorsement. If a $2/3$ majority of peers send the same value of $\tilde{c}$, it is deemed correct and thus included in the set of transactions to be ordered in the next consensus round. Once the consensus round completes, $\tilde{c}$ is permanently saved in the ledger. Therefore, parties in T can be assured that $\tilde{c}$ is correctly produced and not altered.

Secondly, unlike López-Alt et al. (2012), PANTHER requires a threshold of partial decryptions greater than $\frac{|T|+f_t}{2}$ to decrypt $\tilde{c}$. As long as the number of byzantine parties in T is $f_t < \frac{|T|}{3}$, there exists a quorum of honest parties able to decrypt. A party starts the MPC protocol by broadcasting to T a message containing its partial decryption and the associated pair $(\tilde{c}, T)$. Upon delivery of such message, the parties check that $(\tilde{c}, T)$ are consistent with the trustworthy ones stored in the ledger, and if it fails, they decide not to join the protocol. This forces a byzantine initiator to broadcast the correct $(\tilde{c}, T)$, which ensures that each honest party generates a correct partial decryption of $\tilde{c}$ and broadcasts it in turn to correct set T. All honest parties eventually receive a quorum of correct partial decryptions that meets the threshold and successfully decrypts.

Using the blockchain for the aforementioned verifications is simpler and less computationally expensive than using ZKP or succinct arguments, as it just requires a standard interaction with blockchain peers to check data correctness.

5.5.2 MKHE-equipped permissioned Blockchain

Figure 5.1 shows the architecture of PANTHER, where the MKHE scheme is integrated with a permissioned blockchain. Each participating user relies on a client to interact with the blockchain network, and holds a private database (*PDB*) to securely store their private keys. We refer to N as the set of PANTHER's clients. The blockchain network comprises M computing peers that process transactions and execute user-selected functions via smart contracts. Each peer stores the same version of the ledger. We stress that N and M are two disjoint sets.

As any permissioned blockchain, PANTHER includes a CA that enables the authentication of all system parties by issuing verifiable digital certificates. Each client $BC \in N$ and each peer $CP \in M$ uses its certificate private key Csk [3] to sign the messages it sends within PANTHER. The authentication enabled by the CA also allows enforcing read permissions on the data stored in the ledger. When a peer

[3]Note that there is no relationship between the keys associated with the CA digital certificate and the HE keys generated by the KEYGEN algorithm.

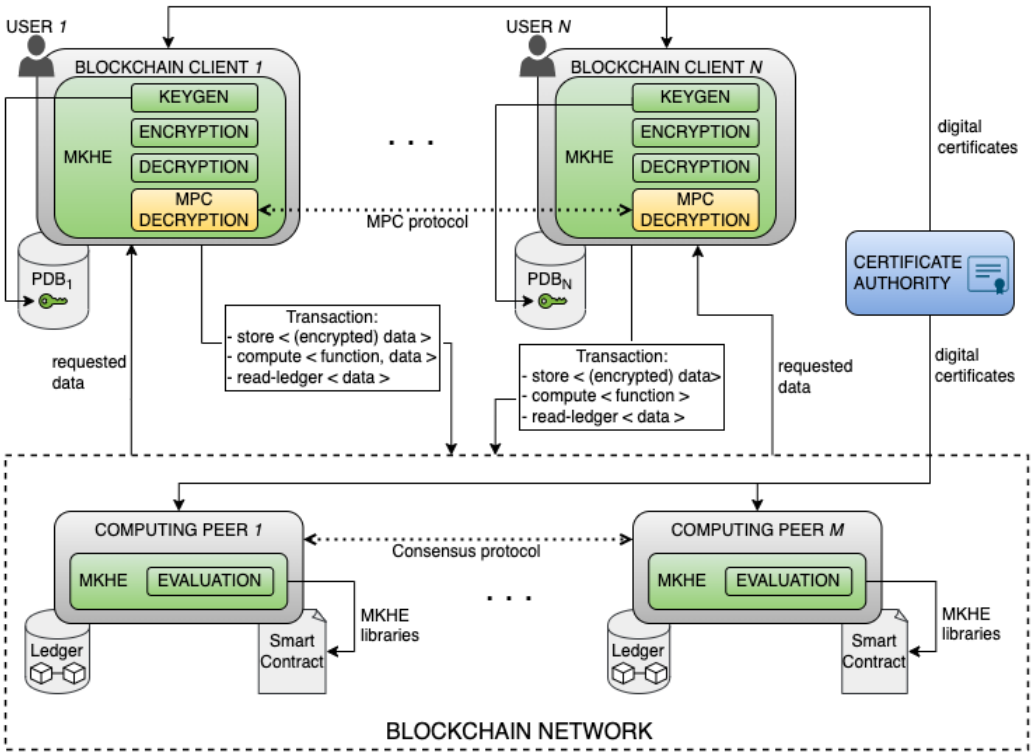


FIGURE 5.1: PANTHER architecture

receives a read request, it first verifies that the requester's digital certificate matches the set of authorised clients. Then, data are handed back to the requester only if a 2/3 majority of peers approves the request. However, if the data is encrypted then its actual value remains concealed. Note that this authorisation mechanism is also enforced when a client requests data owned by another client as input for a computation.

A client submits transactions to the blockchain network and retrieves data stored in the ledger. There are three types of transactions:

- STORE: to record user's data in the ledger. If confidential, data can be stored encrypted;
- COMPUTE: to request the peers to compute a desired function on some data recorded in the ledger;
- READ-LEDGER: to read desired data from the ledger.

As a distinguishing characteristic of PANTHER, each BC_j implements 4 out of the 5 MKHE scheme algorithms, namely KEYGEN, ENC, DEC and MPC-DEC.

The KEYGEN algorithm allows a client to generate the user's personal HE keys ($HEpk_j$, $HEsk_j$, $HEek_j$). The user keeps $HEsk_j$ secret in their personal PDB_j, separately from the

blockchain network. The ENC and DEC algorithms are executed when the client needs to encrypt user's data before submitting a STORE request, or to decrypt ciphertexts upon receiving a READ-LEDGER response, respectively.

A client starts an MPC-DEC protocol when it wants to decrypt an evaluated ciphertext to which other clients too contributed with their own data. Specifically, when a client receives an evaluated ciphertext $\tilde{c}$ in response to a READ-LEDGER request, it starts an MPC decryption protocol with the T clients that contributed to $\tilde{c}$, i.e., those clients that provided their own data as inputs to the EVAL algorithm to produce $\tilde{c}$. The MPC decryption protocol proceeds in steps. Firstly, the jth client decrypts $\tilde{c}$ with $HEsk_j$, producing a partial decryption pc_j that is then broadcast to the clients T. Once received pc_j, each client in T decrypts $\tilde{c}$ and produces its partial decryption that is then broadcast to the others. When a client in T receives enough partial decryptions, depending on the preset threshold, it can combine them to extract the value of $\tilde{c}$. In PANTHER, we set the decryption threshold as equal to the size of quorum tolerating byzantine clients, i.e., greater than $\frac{|T|+f_t}{2}$. Hence, any quorum of honest clients in T can decrypt $\tilde{c}$.

Peers process the transactions submitted by the clients. Each peer in M executes smart contracts to process COMPUTE transactions. In PANTHER, smart contracts implement the MKHE EVAL algorithm to perform functions on ciphertexts. We consider the MKHE integrated in PANTHER to be a fully HE scheme, i.e. MKFHE, where is possible to perform both addition and multiplication (and any combination of them). Note that when peers execute a EVAL $= (\mathcal{F}, \vec{c}, \vec{pk}, \vec{ek})$ computation, they only know what function $\mathcal{F}$ is requested. Peers do not know the actual value of either the inputs or the output, because these data are encrypted. Only the clients in T will discover the output of EVAL (by running a MPC-DEC protocol), and they are entitled to discover it. Hence, there is no leakage of information during the execution of a smart contract, apart from the function it performs.

Furthermore, each peer in M participates in the blockchain consensus. We assume that peers run the PBFT consensus algorithm Castro and Liskov (1999) to withstand against a byzantine adversary. PBFT proceeds in rounds, for each round there is a leader and a set of backups that order incoming transactions and enclose them in a block. At the end of a round, each peer deterministically appends the new block to its local copy of the ledger.

5.6 PANTHER protocols

This section presents the protocols executed by clients and peers in PANTHER. We first introduce the notation used to define the protocols. We use round brackets ( ) to denote a tuple and curly brackets { } to denote a set. We use the symbol $\perp$ to denote

the undefined value, $\emptyset$ to denote the empty set, and $\|$ for string concatenation. We use the notation $name[tag] = values$ to denote an instance of a map called *name*, in which *values* are mapped to a singular *tag*. The type of *tag* is integer, whereas *values* can be of any type, including a tuple, a set or even a map. We use the function SHOW to display string messages to users, which comprise the peers' replies if in plaintext, and the decryption outputs if in ciphertext. We use the function ONLEDGER to verify whether some data received as input corresponds to that stored on the ledger. To ease readability, we omit to describe the steps that involve interactions with the CA. Nevertheless, we use the functions CA-RELEASE to obtain digital certificates from a CA, CREATESIG to digitally sign a string with the certificate private key, and VERIFYSIG to verify whether a given string is properly signed and has not been tampered with. We use the functions RETRIEVEDATA, RETRIEVEKEYS, and RETRIEVEOWNERS to let a peer to retrieve data details from its local copy of the ledger. We use the function PBFT to let a peer start a new round of consensus, and UPDATELEDGER to append the block produced by the last consensus round to the ledger.

The Protocols 5.1 and 5.2 are executed by each blockchain client BC_j. In particular, Protocol 5.2 is the continuation of Protocol 5.1. At first, BC_j sets up the maps *decs* and *replies*, initialising them to empty maps.

- The *decs* map handles the MPC decryption protocol instances; a tag represents a transaction id and a value represents a tuple (*info*, *partial*). The field *info* is itself a tuple storing (i) the ciphertext to be decrypted in the *enctxt* field and (ii) the set of clients to be involved in the decryption in the *parties* field. The field *partial* instead is the set where partial decryptions are collected.
- The *replies* map collects peers' responses for a requested transaction; a tag represents a transaction id and a value represents a map *peers*, which in turn has a peer id as a tag and a tuple (*res*, *bcs*) as a value. The field *res* records the data received, while the field *bcs* represents the set of the owners of that data.

Each BC_j receives a certificate from the CA for the keys (Cpk_j, Csk_j) and generates its HE cryptographic keys $(HEpk_j, HEek_j, HEsk_j)$ using the KEYGEN algorithm.

When a user intends to submit transactions to the blockchain network, they can choose between STORE, COMPUTE and READ-LEDGER types. Before submission, each transaction is signed to enable verification by the peers. If a user wants to submit confidential data, they can encrypt it using the ENC algorithm and send the resulting ciphertext into a STORE transaction. As a result of a READ-LEDGER request, the client BC_j receives from peers replies that contain the requested object d and the set T of clients associated to d. Upon the delivery of a reply from peer CP_x, BC_j verifies its signature and, if valid, updates the maps *replies* with the values received from CP_x.

Once BC_j has received a tuple (d, T) from a quorum $\frac{|M|+f_p}{2}$ of peers, it then processes the reply:

1. if d is only encrypted under $HEpk_j$, then it is simply decrypted using $HEsk_j$
2. if d is encrypted under multiple keys, then the BC_j starts an MPC decryption protocol among all the clients in T by calling the function MPC-DEC (see Protocol 5.2)
3. if BC_j is not included in T, then it is not allowed to decrypt because d results from an EVAL in which no input data was provided by BC_j.

We formally define the properties of the client protocols.

Definition 5.2 (Properties of the client protocol)**.** Let BC_j be a PANTHER's blockchain client ($j \in N$), equipped with an MKHE scheme $\mathcal{E}_j$ = (KEYGEN, ENC, DEC, MPC-DEC). We say that PANTHER satisfies the security properties for a client BC_j with respect to an adversary $\mathcal{A}$ if:

(A) *Privacy*: BC_j cannot learn either the content of transactions in transit or the data of other clients stored in the ledger. This includes the outputs of EVAL executions for which BC_j is not involved.

(B) *Integrity*: if $\mathcal{A}$ intercepts a message in transit intended for BC_j and tampers with its contents, then BC_j notices the modifications and discards the tampered message.

(C) *Availability*: even if $\mathcal{A}$ drops a message in transit intended for BC_j, eventually BC_j delivers the message.

In Protocol 5.2, a client BC_j starts an MPC protocol to decrypt an evaluated ciphertext d. It firstly decrypts d using its $HEsk_j$ to produce a partial decryption pc_j. Then, via an SD1 message, it broadcasts pc_j, along with a signature of the tag $id \parallel d \parallel pc_j$, to the group of clients T that were involved in the creation of d. This initiates the phase one of the MPC decryption protocol. Once a client in T delivers SD1, it firstly verifies that the signature is valid by comparing the attributes received with the signed tag, and checks the integrity of d and T by ensuring that they match the versions stored on the ledger. In addition, it checks that an MPC decryption for that specific *id* does not already exist. If these conditions are satisfied, each client in T creates a record for this MPC protocol instance in its *decs* map, by populating the sets *info* and *partial*, and proceeds to partially decrypt d. At this point, each client broadcasts to all the others its partial decryption and the corresponding signature in an SD2 message, which starts the second and last phase of the protocol. Note that, since we rely on asynchronous

Protocol 5.1 Blockchain Client BC_j

Init:
  $N = \{BC_1, \ldots, BC_n\}$; $M = \{CP_1, \ldots, CP_m\}$;
  $(HEpk_j, HEek_j, HEsk_j) = \text{KEYGEN}(1^\lambda)$;
  $(Cpk_j, Csk_j) = \text{CA-RELEASE}()$;
  initialise *decs*, *replies*;

upon request to create transaction **do**
  $id = \text{CREATETXNID}()$;
  select $txn \in \{\text{STORE}, \text{COMPUTE}, \text{READ-LEDGER}\}$
  if $txn == \text{STORE} \wedge d$ is confidential **then**
    $ciphertext = \text{ENC}(HEpk_j, d)$;
  $signature = \text{CREATESIG}(Csk_j, id \parallel txn)$;
  broadcast SUBMIT(id, txn, $signature$) **to** M;

upon delivery REPLY(id, d, T, sig) **from** CP_x **do**
  if VERIFYSIG($CP_x, Cpk_x, id \parallel d \parallel T, sig$) **then**
    $replies[id].peers[CP_x] = (d, T)$;

upon finding (id, d, T) **such that**
$|\{CP_i \in replies[id].peers$ **with** $replies[id].peers[i].res = d$
$\wedge\, replies[id].peers[i].bcs = T\}| > \frac{|M|+f_p}{2}$ **do**
  if $BC_j \in T$ **then**
    if $|T| > 1$ **then**
      MPC-DEC(id, d, T);
      SHOW("decrypting with T clients");
    else
      SHOW(id, DEC($HEsk_j$, d));
  else
    SHOW("not allowed to decrypt, ask to T");

communication between clients, all SD1 and SD2 messages exchanged during an execution of Protocol 5.2 are asynchronous.

We denote with f_t the number of byzantine clients in T. Each client in T waits that the size of the set *partial* is larger than $\frac{|T|+f_t}{2}$, i.e. the threshold required to decrypt. Then it tries to locally complete the decryption of d by combining (i.e., summing) the partial decryptions received. This operation is attempted for all the quorums Q_z that are subsets of *partial*, where the size q of each Q_z is $\left\lceil \frac{|T|+f_t+1}{2} \right\rceil$ and z ranges from 1 to binomial $\binom{|T|}{q}$. Note that the decryption threshold is equal to the size of a quorum of clients tolerating f_t.

We refer to $\sum Q_z$ as the sum of all partial decryptions in Q_z, saved in the temporary variable *merge*. If the partial decryptions in Q_z are provided by honest clients, then *merge* contains the correct plaintext result of the requested computation. We assume

Protocol 5.2 MPC decryption of BC_j (follows Protocol 5.1)

1: **function** MPC-DEC(id, d, T) ▷ with $T \subseteq N$
2: $pc_j = \text{DEC}(HEsk_j, d)$;
3: $sig = \text{CREATESIG}(Csk_j, id \parallel d \parallel pc_j)$;
4: **broadcast** SD1(id, d, T, pc_j, sig) **to** T;
5:
6: **upon** delivery SD1(id, d, T, pc_x, sig) **from** BC_x **do**
7: **if** $\text{VERIFYSIG}(BC_x, Cpk_x, id \parallel d \parallel pc_x, sig)$
$\wedge\, \text{ONLEDGER}(d, T) \wedge BC_x \in T \wedge decs[id] == \perp$ **then**
8: $decs[id].info = (d, T)$;
9: $decs[id].partial = pc_x$;
10: **if** $BC_j \neq BC_x$ **then**
11: $pc_j = \text{DEC}(HEsk_j, d)$;
12: $sig = \text{CREATESIG}(Csk_j, id \parallel d \parallel pc_j)$;
13: **broadcast** SD2(id, d, T, pc_j, sig) **to** T;
14:
15: **upon** delivery SD2(id, d, T, pc_x, sig) **from** BC_x **do**
16: **if** $\text{VERIFYSIG}(BC_x, Cpk_x, id \parallel d \parallel pc_x, sig)$ **then**
17: **if** $\text{ONLEDGER}(d, T)$ **then**
18: **if** $decs[id] == \perp \wedge BC_x \in T$ **then**
19: **do** lines 8-13;
20: **else if** first SD2(id,...) from BC_x
$\wedge\, BC_x \in decs[id].info.parties$
$\wedge\, pc_x \notin decs[id].partial$ **then**
21: $decs[id].partial = decs[id].partial \cup \{pc_x\}$;
22:
23: **upon** finding $decs[id]$ **such that**
$|decs[id].partial| > \frac{|decs[id].info.parties| + f_t}{2}$ **do**
24: **for all** $Q_z \subseteq decs[id].partial$
$\wedge\, |Q_z| == \left\lceil \frac{|decs[id].info.parties| + f_t + 1}{2} \right\rceil$ **do**
25: $merge = \sum Q_z$;
26: **if** $merge$ is in plaintext **then**
27: SHOW(id, $merge$);
28: **break**;

that any plaintext result of this kind has a predefined structure that can be verified. If instead some partial decryption in Q_z is provided by a byzantine client, then *merge* does not contain the correct plaintext result of the requested computation. Although in this latter case the probability ϵ that *merge* is compliant anyway with the predefined structure is not zero, we assume it to be small enough to safely rely on *merge* structure validation to determine whether the obtained result is in plaintext (see line 26 in Protocol 5.2). As we prove in Section 5.8, eventually every honest client finds a $\tilde{Q}_z$ that decrypts correctly, as long as $f_t < t/3$.

Figure 5.2 shows an execution of the MPC decryption protocol. BC_1 receives from

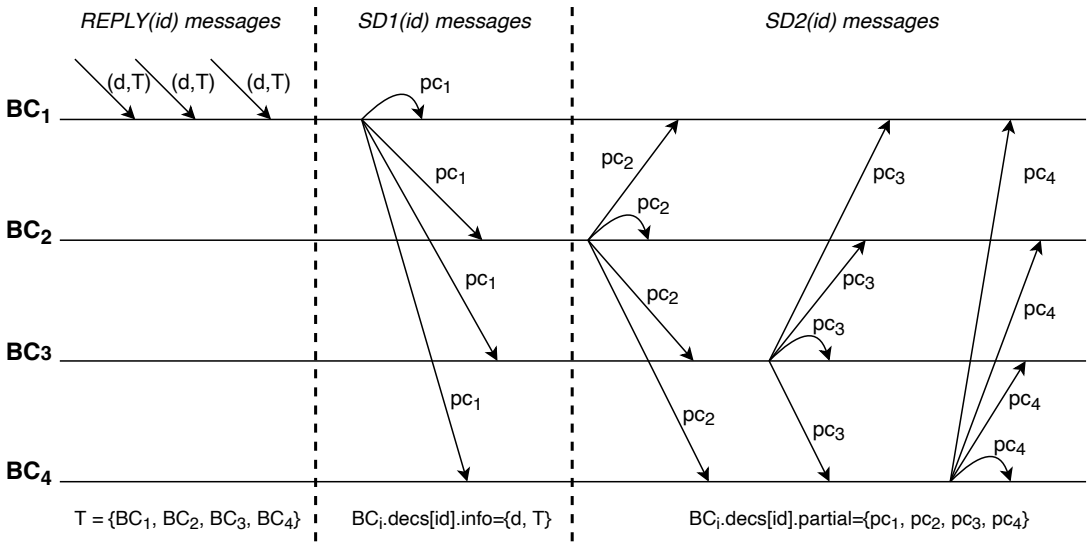


FIGURE 5.2: An execution of the MPC decryption protocol

peers replies indexed by *id*. After collecting the same tuple (d, T) for *id* from more than $\frac{|M|+f_p}{2}$ peers, BC_1 internally calls the MPC-DEC function for decrypting d with T. The set T is composed by BC_1, BC_2, BC_3 and BC_4. BC_1 creates the partial decryption pc_1 and broadcasts it to T in SD1 messages indexed with *id*. Upon receiving this message, BC_2, BC_3 and BC_4 update their local map *decs*[*id*].*info* with the tuple (d, T). Then, they create their respectively partial decryptions pc_2, pc_3, pc_4 and broadcast them to T in SD2 messages indexed with *id*. Upon the delivery of an SD2 message, each BC_i updates its map *decs*[*id*].*partial* with the received partial decryption. Finally, each BC_i decrypts d once its set *partial* is greater than $\frac{|T|+f_t}{2}$. In the example shown in Figure 5.2, this happens after the delivery of pc_3 sent by BC_3.

We formally define the properties for a MPC decryption protocol, i.e. the MPC-DEC of the MKHE model:

Definition 5.3 (Properties of MPC decryption)**.** Let $T = \{BC_1, \ldots, BC_t\}$ be a set of PANTHER's clients, where $t{=}|T|$ and each BC_j holds an MKHE scheme $\mathcal{E}_j$. Let $\tilde{c} =$ EVAL$(\mathcal{F}, (c_1, pk_1, ek_1), \ldots, (c_t, pk_t, ek_t))$ be an evaluated ciphertext, where the jth key tuple (pk_j, ek_j) belongs to BC_j and it is in the support of KEYGEN(1^λ), and each $c_j{=}$ENC(pk_j, m_j). We say that PANTHER satisfies the security properties for a MPC-DEC$(BC_1, \ldots, BC_t, \tilde{c}) = \tilde{m}$ if:

(A) *Privacy*: the only information a client BC_j can learn is the output $\tilde{m}$, i.e., the decryption of $\tilde{c}$.

(B) *Correctness*: the output $\tilde{m}$ is always the correct plaintext of $\tilde{c}$, i.e., $\tilde{m} = \mathcal{F}(m_1, \ldots, m_t)$.

(C) *Fairness*: corrupted clients obtain the output $\tilde{m}$ if and only if the honest clients also do.

(D) *Validity*: any operation executed by honest clients during the protocol refers only to $\tilde{c}$ and involve only the T clients that contributed to $\tilde{c}$.

(E) *Agreement*: if a honest client correctly decrypts $\tilde{c}$, then all the other honest clients eventually decrypt $\tilde{c}$.

(F) *Termination*: if a BC_j starts the protocol then, all the honest clients eventually complete it.

The Protocol 5.3 is executed by each computing peer CP_i. At first, CP_i starts the variable *round* to serialise the PBFT consensus in rounds and sets up the maps *endorseTxns* and *consensusTxns*, initialising them to empty maps.

- Map *endorseTxns* pools transactions for endorsement before consensus; a tag is a transaction id and a value is a tuple (*info*, *peers*). The field *info* is a tuple (*type*, *req*, *owners*), and the field *peers* is a map indexed by peer id and with a tuple (*data*, *keys*) as a value.
- Map *consensusTxns* pools (endorsed) transactions for ordering in the next consensus round; a tag is a transaction id and a value is a tuple (*type*, *req*, *owners*, *data*, *keys*).

Variable *type* records the transaction type, *req* the requester client, *data* the transaction data, and the two sets *keys* and *owners* collect the keys and the list of the clients associated with *data*, respectively. When CP_i receives a transaction indexed by *id* from some BC_x, its signature is firstly verified and then it is handled according to its type. In case of a STORE transaction, CP_i updates its local map *endorseTxns*[*id*] with all submitted attributes (e.g., data and keys) and broadcasts these values to M for endorsement. In case of a COMPUTE transaction, CP_i performs the requested function via smart contract by calling the EVAL algorithm. Then, CP_i updates *endorseTxns*[*id*] with the computation result, i.e. the evaluated ciphertext and its associated HE public keys, and broadcasts these values to M for endorsement. In case of a READ-LEDGER transaction, CP_i retrieves the requested value and its owners from the ledger, then sends them back to the requester client BC_x. [4] The endorsement phase takes place whenever a peer handles a STORE or COMPUTE transaction. During this phase, peers have to verify that all honest among them are working with the same transaction details before entering the next consensus round. For transaction STORE this means that honest peers have received the same data to store, whereas for COMPUTE means that honest have performed the computation correctly and thus produced the same result (i.e., the same evaluated ciphertext). Therefore, each peer broadcasts to others

[4] Note that, being permissioned, the blockchain system enforces access control policies on data stored in the ledger. Thus, a client can (i) request to read a value or (ii) perform computations on data owned by other clients, if and only if it has read permissions on such data.

Protocol 5.3 Computing Peer CP_i

Init:
  $N = \{BC_1, \ldots, BC_n\}$; $M = \{CP_1, \ldots, CP_m\}$;
  $(Cpk_i, Csk_i) =$ CA-RELEASE( );
  initialise *round*, *endorseTxns*, *consensusTxns*;

upon delivery SUBMIT(id, txn, sig) **from** BC_x **do**
  if VERIFYSIG(BC_x, Cpk_x, $id \parallel txn$, sig) **then**
    case txn is STORE(d, $HEpk_x$, $HEek_x$) **then**
      $keys = (HEpk_x, HEek_x)$;
      $endorseTxns[id].info = (txn, BC_x, BC_x)$;
      $endorseTxns[id].peers[CP_i] = (d, keys)$;
      sig = CREATESIG($Csk_i, id \parallel d \parallel keys$);
      broadcast ENDORSE(id, d, $keys$, sig) **to** M;
    case txn is COMPUTE($\mathcal{F}$, $d_1, \ldots, d_t$) **then**
      $res = \perp$; $keys = \emptyset$; $bcs = \emptyset$;
      for $k = 1$ **to** t **do**
        $keys = keys \cup$ RETRIEVEKEYS(d_k);
        $bcs = bcs \cup$ RETRIEVEOWNERS(d_k);
      $res =$ EVAL($\mathcal{F}, d_1, \ldots, d_t, keys$);
      $endorseTxns[id].info = (txn, BC_x, bcs)$;
      $endorseTxns[id].peers[CP_i] = (res, keys)$;
      sig = CREATESIG($Csk_i, id \parallel res \parallel keys$);
      broadcast ENDORSE(id, res, $keys$, sig) **to** M;
    case txn is READ-LEDGER($request$) **then**
      $d =$ RETRIEVEDATA($request$);
      $owners =$ RETRIEVEOWNERS($request$);
      $sig =$ CREATESIG($Csk_i, id \parallel d \parallel owners$);
      trigger REPLY(id, d, $owners$, sig) **to** BC_x;

upon delivery ENDORSE(id, d, $keys$, sig) **from** CP_x **do**
  if VERIFYSIG($CP_x, Cpk_x, id \parallel d \parallel keys, sig$) **then**
    $endorseTxns[id].peers[CP_x] = (d, keys)$;

upon finding $(id, d, keys)$ **such that**
$|\{CP_z \in endorseTxns[id].peers$ **with**
$endorseTxns[id].peers[z].data = d$
$\wedge\, endorseTxn[id].peers[z].keys = keys\}| > \frac{|M|+f_p}{2}$ **do**
  $consensusTxns[id] = (endorseTxns[id].info, d, keys)$;

upon delivery PBFT-COMMIT($block$) **do**
  UPDATELEDGER($block$);
  $round = round + 1$;
  PBFT($round$, $consensusTxns$);
  empty $consensusTxns$;

the id, data and keys of the handled transaction. Once CP_i receives the same transaction values from a quorum $\frac{|M|+f_p}{2}$ of honest peers, it then endorses the transaction and updates its local map $consensusTxns[id]$ with the transaction attributes.

The PBFT consensus of PANTHER proceeds in rounds. When the current round ends, CP_i receives the message PBFT-COMMIT containing the block with the transactions ordered. Then, CP_i appends the block to its local copy of the ledger and enters the next round by proposing the endorsed transactions it collected. We omit to describe the details of the PBFT protocol executed by the peers in each round Castro and Liskov (1999).

We formally define the properties of Protocol 5.3 as follows:

Definition 5.4 (Properties of peer protocol)**.** Let CP_i be a PANTHER's computing peer ($i \in M$), equipped with a smart contract that implements the EVAL algorithm of MKHE. We say that PANTHER satisfies the security properties for a peer CP_i with respect to an adversary $\mathcal{A}$ if:

(A) *Privacy*: CP_i cannot learn either the content of received transactions or the clients' data stored in its ledger. Moreover, CP_i cannot deduce secret information from an execution of the EVAL algorithm.

(B) *Integrity*: if $\mathcal{A}$ intercepts a message in transit intended for CP_i and tampers with its content, then CP_i notices the modifications and discards the tampered message. If CP_i is corrupted by $\mathcal{A}$, although it can tamper with received transactions or alter its smart contracts, CP_i will not be able to append tampered data onto the ledger.

(C) *Availability*: even if $\mathcal{A}$ drops a message in transit intended for CP_i, eventually CP_i delivers the message. If CP_i is corrupted by $\mathcal{A}$, although it can avoid to process transactions or reply to clients, eventually honest peers handle these tasks.

5.7 Efficiency of MPC protocol

We analyse the efficiency of the MPC decryption protocol (Protocol 5.2) in terms of number of messages exchanged between involved clients. Then, we compare it qualitatively with other two representative MPC protocols proposed in literature, namely Mukherjee and Wichs (2016) and Asharov et al. (2012), to show that they have the same message complexity.

An instance of the MPC decryption protocol in PANTHER starts when a client $BC_j \in T$ receives an evaluated ciphertext encrypted under $t = |T|$ unrelated keys. This starts a 2-round of interactions among clients in T:

1. BC_j produces its partial decryption pc_j and broadcasts it to all the clients in T;
2. After receiving pc_j, each client in T (but BC_j) produces its own partial decryption and broadcasts it to all the others. Once a client collects a quorum of partial decryptions, it can locally combine them to obtain the plaintext result.

Note that Protocol 5.2 does not include other steps that MPC protocols usually encompass, such as providing encrypted inputs and computing the EVAL function. These steps are asynchronous in PANTHER with respect to the MPC decryption protocol execution, since they are carried out at different times and with different input data. For instance, a group T of clients can initially just store their data and, later on, one of them can decide to request a computation on a subset $T' \subset T$. In terms of communication complexity, the first round of MPC decryption has an asymptotic cost of $O(t)$ because the initiator sends a message to T, and the second round has an asymptotic cost of $O\left((t-1)t\right)$ because each client in T but the initiator sends a message to all the others. Therefore, the overall message complexity of Protocol 5.2 is $O(t^2)$.

Mukherjee and Wichs (2016) propose a 2-round MPC protocol that works as follows:

1. Each party encrypts its own input under its own public key and broadcasts the ciphertext. After receiving other parties ciphertexts, each party locally produces an evaluated ciphertext by running the EVAL algorithm on the desired function;
2. Each party broadcasts its partial decryption of the evaluated ciphertext. After collecting a threshold of partial decryptions, each party can combine them to recover the function output in plaintext.

Only round 2 of this protocol can be compared with Protocol 5.2, because round 1 covers steps that PANTHER performs in other protocols (as explained above). Considering t parties are involved in Mukherjee's and Wichs's MPC decryption protocol, its message complexity is $O(t^2)$ because each party sends a messages to all the other parties.

Asharov et al. (2012) propose instead a 3-round MPC protocol based on the THE model:

1. Each party broadcasts its public key. When a client receives the public keys of all the others, it combines them to generate a common public key;
2. Each party broadcasts an encryption of its input under the common public key. At the end of this round, each party locally produces an evaluated ciphertext by running the EVAL algorithm on the desired function;

3. Same as round 2 of Mukherjee and Wichs (2016).

The same considerations made for Mukherjee and Wichs (2016) apply to Asharov et al. (2012), therefore the message complexity of this protocol is again $O(t^2)$.

In conclusion, the message complexity of Protocol 5.2 matches that of state-of-the-art MPC decryption protocols.

5.8 Security analysis of PANTHER

We analyse the security of PANTHER and its inherent operating protocols, in the face of a byzantine adversary attacking the system as per Definition 5.1.

5.8.1 Client security

Theorem 5.5 (Data privacy for clients in PANTHER)**.** *The data of a client in PANTHER remains private since they are submitted via transactions until they are stored into the ledger, so meeting Property (A) of Definition 5.2.*

Proof. The *privacy* property (A) of Definition 5.2 follows from the *semantic security* property (Definition 2.4) of the MKHE schemes implemented in PANTHER. We consider that a PPT adversary $\mathcal{A}$ attacking PANTHER knows the HE and CA public keys of all system clients, i.e., each $HEpk_j$, $HEek_j$, Cpk_j for $j \in N$. Assume the adversary $\mathcal{A}$ corrupts a client BC_x. The aim of $\mathcal{A}$ is to collect ciphertexts of honest clients and try to deduce from them secret information, such as the corresponding private keys. Then, $\mathcal{A}$ can eavesdrop transactions in transit or read ciphertexts from the ledger. Assume that $\mathcal{A}$, knowing the context, can guess the plaintexts an honest client plans to encrypt. For instance, if the system implements an auction, $\mathcal{A}$ knows that the transactions most likely contain encrypted bids, which can be estimated to a certain range. Let $\bar{m}$ be a guessed plaintext and $\tilde{c}_y$ be a fresh ciphertext that BC_x collects from BC_y, then BC_x can produce $\bar{c} = \text{ENC}(HEpk_y, \bar{m})$, and compare $\bar{c}$ with $\tilde{c}_y$. However, even if BC_x guessed the plaintext correctly, since the MKHE scheme is semantically secure, $\bar{c}$ will differ from $\tilde{c}_y$, and so BC_x cannot gain any useful insight from the comparison. This is because ENC algorithm of BC_y includes a randomness component in $\tilde{c}_y$, which makes it one of many valid ciphertexts for $\tilde{m}_y$.

Similarly to fresh ciphertexts, $\mathcal{A}$ can collect evaluated ciphertexts, either in transit or from the ledger, and their related inputs. Let $\tilde{c}$ be an evaluated ciphertext that BC_x collects, and its input ciphertexts be $\vec{c} = \{c_1, \ldots, c_t\}$. Then, guessing the function $\mathcal{F}$ involved in producing $\tilde{c}$, BC_x can compute $\bar{c} = \text{EVAL}(\mathcal{F}, H\vec{E}pk, H\vec{E}ek, \vec{c})$, where $H\vec{E}pk$

and $H\vec{E}ek$ are the t public keys related to $\vec{c}$, and can compare $\bar{c}$ with $\tilde{c}$. However, the EVAL is a public algorithm with no secrets, thus semantic security holds in the presence of evaluation keys $H\vec{E}ek$. Also, the EVAL algorithm comprises a randomness component, thus $\bar{c}$ will be different from $\tilde{c}$. Hence, the comparison does not bring any useful insight to the adversary $\mathcal{A}$.

Furthermore, knowing the context, $\mathcal{A}$ can guess the decrypted result of an evaluated ciphertexts. Continuing with the auction example, $\mathcal{A}$ can guess the winning bid. Let $\hat{m}$ be a guessed decrypted plaintext of an evaluated ciphertext $\tilde{c}$ that BC_x collects, and let $\mathcal{F}$ be the guessed function computed in $\tilde{c}$, such that ideally $\hat{m} = \mathcal{F}(m_1, \ldots, m_t)$. Then, BC_x can produce $\hat{c} = \text{ENC}(H\vec{E}pk, \hat{m})$, and compare it with $\tilde{c}$. However, according to Definition 2.7, if the HE scheme is circuit private then the distribution of EVAL algorithm is (statistically) indistinguishable from the distribution of ENC algorithm. This means that a ciphertext produced by EVAL does not reveal its inner function, and thus BC_x cannot determine whether $\tilde{c}$ is actually computed by the guessed function $\mathcal{F}$. Hence, the comparison between $\hat{c}$ with $\tilde{c}$ does not bring any useful insight to $\mathcal{A}$. □

Theorem 5.6 (Data integrity for clients in PANTHER)**.** *The replies that a blockchain client receives from the peers in PANTHER are tamper-proof, so meeting Property (B) of Definition 5.2.*

Proof. The *integrity* property (B) of Definition 5.2 follows from the permissioned nature of the blockchain infrastructure implemented in PANTHER, and from the authentication checks carried out in Protocol 5.1. Assume the adversary $\mathcal{A}$ intercepts a reply message m_{xy} sent from a peer CP_x to a client BC_y and tampers with its contents. Upon the message delivery, BC_y firstly verifies that m_{xy} is correctly signed by the sender (line 16), i.e., it applies the CA key Cpk_x of CP_x to the signature in m_{xy} and checks that the result matches with m_{xy} contents. This check will fail if $\mathcal{A}$ tampers with some parameters of m_{xy} or if it replaces the original signature. Consequently, BC_y discards the tampered m_{xy}. □

Theorem 5.7 (Data availability for clients in PANTHER)**.** *If an honest peer CP_y sends a message m_{yx} to an honest client BC_x in PANTHER, then eventually BC_x delivers m_{yx}, so meeting the availability property (C) of Definition 5.2.*

Proof. The *availability* property (C) of Definition 5.2 follows from the assumptions that system parties are fully interconnected and the communication channels are reliable. Assume the adversary $\mathcal{A}$ discards a reply message m_{yx} sent from an honest CP_y to an honest BC_x. Relying on reliable links, CP_y will keep relay m_{yx} until BC_x delivers it Cachin et al. (2011). Note the case where CP_y is byzantine: CP_y can then send an incorrect reply to BC_x, say m''_{yx} instead of m'_{yx}. From Protocol 5.1 (in line 19) it follows that, before processing a reply message, BC_x waits to see a quorum $\frac{M+f_p}{2}$ of replies

with the same content. Since honest peers store the correct m'_{yx} in their ledger, m''_{yx} will not reach the quorum and BC_x eventually process m'_{yx}. □

5.8.2 MPC-Decryption security

Before proving the security of the MPC decryption protocol, we give the definition of the ideal/real paradigm (Canetti (2000); Goldreich (2004)), which we will use to prove the privacy property.

Definition 5.8 (Ideal/Real paradigm)**.** Let $\text{IDEAL}_{\mathcal{D},\mathcal{S}}(\vec{x})$ be the joint output of an ideal-world adversary $\mathcal{S}$ and parties $T=\{P_1,\ldots,P_t\}$ in an ideal execution with functionality $\mathcal{D}$ and inputs $\vec{x} = \{x_1,\ldots,x_t\}$. Similarly, let $\text{REAL}_{\Pi,\mathcal{A}}(\vec{x})$ be the joint output of a real-world adversary $\mathcal{A}$ and parties T in an execution of protocol Π with inputs $\vec{x}$. We say that protocol Π securely realises $\mathcal{D}$ if for every real-world adversary $\mathcal{A}$, there exists a ideal-world adversary $\mathcal{S}$ with black-box access to $\mathcal{A}$ such that for all input vectors $\vec{x}$,

$$\text{IDEAL}_{\mathcal{D},\mathcal{S}}(\vec{x}) \approx \text{REAL}_{\Pi,\mathcal{A}}(\vec{x}).$$

This definition asserts that the output distribution of the adversaries $\mathcal{A}$ and $\mathcal{S}$ [5] are indistinguishable in the real and ideal executions.

Theorem 5.9 (Security of MPC decryptions in PANTHER)**.** *Any instance of the MPC decryption Protocol 5.2, indexed by a unique id, occurring in PANTHER among a subset of clients $T = \{BC_1,\ldots,BC_t\}$, where $T \subseteq N$ and $t = |T|$, to decrypt an evaluated ciphertext $\tilde{c}$, meets the properties of Definition 5.3, as long as the byzantine clients are $f_t < t/3$.*

Proof. Recall that a byzantine quorum tolerating f_t is a set of more than $\frac{t+f_t}{2}$, and that the MPC-threshold set to reconstruct is $thd \geq \left\lceil \frac{t+f_t+1}{2} \right\rceil$.

We prove the *privacy* property (A) of Definition 5.3 by relying to the ideal/real paradigm, such that Protocol 5.2 emulates an ideal world where the decryption is carried out by an external trusted party: each client sends its partial decryption to the trusted party, which decrypts and sends back the result to clients. Protocol 5.2 is deemed secure if whatever adversaries can obtain from its execution can also be feasibly obtained in an execution that takes place in the ideal world. That is, no adversary is able to obtain more secret information in the real world than in the ideal one. The $\text{REAL}_{\Pi,\mathcal{A}}(\vec{pc})$ for Protocol 5.2 comprises the outputs and the views of clients T during an execution of Π. A client's output is equal to $\tilde{m}$, i.e., the decryption of $\tilde{c}$ after combining (locally) the partial decryptions received. The jth client's view is equal to $(1^\lambda, sk_j, \vec{pc})$, i.e., the security parameter, its private key and the partial

[5] We denote $\mathcal{S}$ the ideal-world adversary because it behaves as a simulator, simulating a real protocol execution for $\mathcal{A}$ while it really interacts in the ideal world.

decryptions it collects from T. Note that as $\mathcal{A}$ controls f_t corrupted parties, it knows f_t private keys. Conversely, in the ideal world, a client interacts only with the trusted party, thus its output is the $\tilde{m}$ it receives from the trusted party, and its view only consists of $(1^\lambda, sk_j, pc_j)$, i.e., its secret information and its own partial decryption. In order to make the two worlds similar and comparable, the view of a client in the ideal can be extended with simulations of other clients partial decryptions, created via a PPT algorithm given access to the client's input and output, i.e., its pc_j and $\tilde{m}$. This results in producing for each client in T a set of simulated partial decryptions $\vec{pc}'$, such that $\vec{pc}' \approx \vec{pc}$. In LWE-based MKHE schemes, as in the work of Mukherjee and Wichs (2016), the adversary $\mathcal{S}$ creates this simulations by '*xoring*' the partial decryptions of the f_t corrupted clients with $\tilde{m}$.[6] After this process, a client's output and view in $\text{IDEAL}_{\mathcal{D},\mathcal{S}}(\vec{pc})$ will be $\tilde{m}$ and $(1^\lambda, sk_j, \vec{pc}')$ respectively. As shown by Mukherjee and Wichs (2016), if $\tilde{m}$ is the correct decryption of $\tilde{c}$ then the $\vec{pc}'$ are valid simulations, i.e., each pc'_j is a valid output of $\text{DEC}(sk_j, \tilde{c})$. A simulated pc'_j only differs from a real pc_j in the random noise they contain, which makes them (statistical) indistinguishable. Hence $\vec{pc}' \approx \vec{pc}$, then the clients' views in the ideal will be indistinguishable from those of the real, and therefore $\text{IDEAL}_{\mathcal{D},\mathcal{S}}(\vec{pc}) \approx \text{REAL}_{\Pi,\mathcal{A}}(\vec{pc})$.

The *correctness* property (B) of Definition 5.3 follows from the observations that the required threshold *thd* corresponds to the minimum size of a byzantine quorum, and that there exists at least a byzantine quorum of honest clients. When a SD1(*id*, ...) or a SD2(*id*, ...) message arrives, honest clients in T discard duplicates with same *id*, and also different versions of a partial decryption sent by the same client. These checks are carried out in lines 7, 17 and 20 of Protocol 5.2. In addition, honest clients check whether the sender of SD1 and SD2 messages belongs to T. All these checks ensure that the set *partial* of any honest client collects at most t partial decryptions for $\tilde{c}$, and that at most f_t of them can be not valid. Whereas the correctness of the set *parties* follows from the integrity of parameter T contained in both SD1 and SD2 messages, which is validated against the ledger. Therefore, each honest client fulfils the condition $|partial| > \frac{|parties|+f_t}{2}$ correctly. Now considering all the $\binom{t}{thd}$ possible combinations of received partial decryptions in *partial* satisfying *thd*, there exists at least one of them, say $\tilde{Q}$, whose partial decryptions are sent by honest clients. This because, as defined in Section 5.3, for $f_t < t/3$ there is at least a quorum of honest clients. Since in Protocol 5.2 each honest client in T iterates the reconstruction, i.e., the combination of partial decryptions, until $\tilde{Q}$ is met, then eventually each of them correctly decrypt $\tilde{c}$, which by the correctness of MKHE (Definition 2.5) will be $\tilde{m} = \mathcal{F}(m_1, \ldots, m_t)$.

The *fairness* property (C) of Definition 5.3 follows from the observations that in order to successfully decrypt, byzantine clients need the partial decryptions from honest ones, i.e., byzantine clients cannot satisfy *thd* by themselves. Suppose a byzantine

[6] Particularly, Mukherjee and Wichs (2016) give a stronger notion of security by taking an adversary that controls all-but-one parties, so $f_t = t - 1$.

client BC_x wants to decrypt $\tilde{c}$ and to prevent honest ones from doing so. Then, BC_x sends an SD1 message to all other byzantine and to a subset $\tilde{H}$ of honest clients large enough to satisfy *thd*. BC_x can attach an invalid partial decryption in the message and attempt to prevent some communications among honest clients. In addition, suppose the other byzantine clients send invalid partial decryptions with an SD2 message to $\tilde{H}$. When BC_x receives enough SD2 messages it decrypts them and sends the resulting $\tilde{m}$ to other byzantine clients. Clients in $\tilde{H}$ are not able to decrypt it since they collect invalid partial decryptions from byzantine clients. However, each client in $\tilde{H}$ sends an SD2 message to all T clients, reaching the other honest clients that have been excluded by BC_x. Observe from the protocol that if a client delivers a SD2(*id*,...) message but never receives a SD1(*id*,...) one, it executes the same steps in Protocol 5.2, and thus broadcasts its partial decryption to T. This means that eventually clients in $\tilde{H}$ will deliver SD2 messages from all other honest clients in T, and will find a quorum $\tilde{Q}$ that decrypts $\tilde{c}$. Therefore, byzantine clients cannot prevent the honest to obtain $\tilde{m}$.

The *validity* property (D) Definition 5.3 follows from the integrity checks carried out via ONLEDGER function. When BC_x starts the MPC protocol with *id*, it broadcasts through a SD1 message its partial decryption for $\tilde{c}$ to the involved T clients. If BC_x is byzantine it can send the partial decryption to a set of clients different from T, or it can attach to an SD1 message wrong $\tilde{c}$ and T parameters. However, when a SD1 message arrives to an honest client, this verifies via ONLEDGER that the attached parameters correspond to the tamper-proof $\tilde{c}$, that T is stored on the ledger, and that the sender actually belongs to T. If this check fails, the SD1 message is discarded, and thus the byzantine BC_x is forced to attach the correct $\tilde{c}$ and T in SD1. Conversely, if it succeeds, honest clients populate their local set *parties* for *id* properly, and broadcast their partial decryptions to the appropriate T with an SD2 message, eventually reaching all those who are honest. At the delivery of an SD2 message, an honest client verifies again via ONLEDGER the integrity of $\tilde{c}$ and T, ruling out byzantine clients sending incorrect SD2. Since any honest client instantiates properly the local set *parties* in the first phase of the protocol, then the constraint in line 23 is validated against the correct T clients. Hence, it follows that the reconstruction operations of the last phase are carried out against valid subsets of *partial*, i.e., a valid byzantine quorum. Therefore, in any phase of the protocol, any honest client deals with the appropriate $\tilde{c}$ and T.

We prove the *agreement* property (E) of Definition 5.3 by contradiction. Assume that for a MPC decryption protocol an honest client BC_x decrypts, whereas another honest BC_y does not. To successfully decrypt, BC_x finds a quorum $\tilde{Q}_x$ composed by correct partial decryptions, of which at least one was sent by another honest client. This because being $|\tilde{Q}_x| = \left\lceil \frac{t+f_t+1}{2} \right\rceil$ and $t > 3f_t$, then

$$\frac{t+f_t+1}{2} > \frac{3f_t+f_t+1}{2} = \frac{4f_t+1}{2} = 2f_t + \frac{1}{2} > f_t$$

namely $\tilde{Q}_x$ contains more than one partial decryption belonging to honest clients. For instance, consider a MPC protocol among $T=\{BC_1, BC_2, BC_3, BC_4\}$, where BC_3 is byzantine, BC_1 can decrypt and BC_4 cannot. Consider that BC_1 decrypts with a quorum $\tilde{Q}_1=\{pc_1, pc_2, pc_3\}$. This means that BC_1 previously broadcasted pc_1 to all T, including itself. Since BC_1 obtained pc_2, and BC_2 is honest, then pc_2 was previously broadcasted by BC_2 to all T. These deductions follow from the *validity* property. As a result, BC_4 will eventually deliver both pc_1 and pc_2, and hence it creates and broadcasts pc_4, which enables it to find $\tilde{Q}_4$ and thus to decrypt. This shows the contradiction. Note that no deduction can be made on pc_3, because it is created by BC_3, which being byzantine can send different versions of pc_3 or can even avoid to send it to some clients. When BC_2 delivers pc_4 it is able to find a quorum $\tilde{Q}_2$ and it finally decrypts. Therefore, this proves that if an honest client decrypts then all other honest decrypt.

The *termination* property (F) of Definition 5.3 follows from the observation that, when a client BC_x, regardless of whether honest or byzantine, starts a MPC decryption with *id*, it sends an SD1 message to at least an honest client BC_y. Regardless of whether the partial decryption delivered by BC_x is valid or not, BC_y sends an SD2 message containing its own partial decryption to T, which by the *validity* property will be the correct set of clients and will eventually reach those who are honest. At the delivery of this SD2 from BC_y, all honest clients in turn broadcast their partial decryptions to T. Eventually, by the *agreement* property, all honest clients find a quorum $\tilde{Q}$ that by the *correctness* property correctly decrypts $\tilde{c}$, and therefore all of them terminate the protocol for *id*. □

5.8.3 Peer security

Theorem 5.10 (Data privacy for peers in PANTHER)**.** *A peer cannot learn data about client in PANTHER nor deduce secret information from the executions of* Eval*, so meeting the privacy property (A) of Definition 5.4.*

Proof. The *privacy* property (A) of Definition 5.4 follows from the *semantic security* property (Definition 2.4) of the MKHE schemes implemented in PANTHER. We consider that a PPT adversary $\mathcal{A}$ attacking PANTHER knows the CA public keys of all system peers, i.e., each Cpk_i for $i \in M$. The same reasoning as in the proof of Theorem 5.5 applies. The difference here is that the adversary $\mathcal{A}$, corrupting a peer CP_x, can directly access the data of the transactions it receives or those stored in its ledger copy. Hence, CP_x can collect fresh ciphertexts and compare them with encryptions of guessed plaintexts, however, if the MKHE schemes are semantically secure, CP_x cannot gain any insight from comparing them.

Unlike clients, peers in PANTHER perform the EVAL algorithm. A peer CP_x corrupted by $\mathcal{A}$ can inspect the ciphertexts "extended" during EVAL processing. Let $\mathcal{F}$ be a

function to be computed on input ciphertexts $\vec{c} = \{c_1, \ldots, c_t\}$ encrypted with t different public keys $\vec{HEpk}$. Then, CP_x extends each ciphertext of $\vec{c}$ with a public key $\tilde{pk}$, which will be a combination of all keys $\vec{HEpk}$. To execute this extension, CP_x applies the key-switching technique Brakerski et al. (2012), that uses the (public) evaluation keys $\vec{HEek}$ associated to $\vec{c}$. Typically, an evaluation key is generated as the encryption of the private key, or certain parts thereof. By semantic security it holds that $\vec{HEek}$ are indistinguishable from encryptions of random values, keeping private their contained private keys $\vec{HEsk}$. Hence, the key-switching does not reveal any information about the private keys of clients and CP_x cannot gain any insight from its generated extended ciphertexts. Note that LWE-based MKFHE schemes, such as Clear and McGoldrick (2015); Mukherjee and Wichs (2016); Peikert and Shiehian (2016), do not employ an evaluation key per user, and to perform key-switching they rather attach helper information to a fresh ciphertext that needs to be extended. Such information is the encryption of the randomness related to the ciphertext, which is semantic secure per definition. □

Theorem 5.11 (Data integrity for peers in PANTHER)**.** *The transactions received from clients by a peer in PANTHER are tamper-proof, and a corrupted peer cannot store tampered data into the PANTHER ledger, so meeting property (B) of Definition 5.4.*

Proof. The *integrity* property (B) of Definition 5.4 follows from the permissioned nature of the blockchain infrastructure implemented in PANTHER, and from the authentication checks carried out in Protocol 5.3. Assume the adversary $\mathcal{A}$ intercepts a transaction m_{yx} sent from a client BC_y to a peer CP_x and tampers with its contents. Upon the delivery of the message, CP_x firstly verifies that m_{yx} is correctly signed by the sender (line 7), i.e., it applies the CA key Cpk_y of BC_y to the signature in m_{yx} and checks that the result matches with m_{yx} contents. This check will fail if $\mathcal{A}$ tampers with some parameters of m_{yx} or if it replaces the original signature. Consequently, CP_x will discard the transaction.

Besides attacking the communication channels, $\mathcal{A}$ can corrupt a peer CP_x and thus tamper with the data it handles or alter the executions of its smart contracts. Suppose that CP_x tampers with the content of a transaction m_{yx} it receives from BC_y. Since BC_y broadcasts to all M peers and since the broadcast primitive is reliable, eventually all honest peers in M deliver m_{yx}. By assumption the number of honest peers is $M > 3f_p$. This implies that a majority of them, i.e., an honest quorum of $\frac{M+f_p}{2}$ peers, will propose the un-tampered m_{yx} to be appended to the chain during consensus protocol. According to PBFT Castro and Liskov (1999), after receiving a majority of proposals for m_{yx}, the leader accepts it and includes m_{yx} in a block that will finally be committed by all (honest) peers. Hence a corrupted peers, or even a colluding corrupted group up to f, cannot force the blockchain network to store a tampered transaction. The same applies if CP_x alters the execution of a smart contract for some inputs $\{d_1, \ldots,$

$d_t\}$, producing an incorrect result. Eventually, all honest peers in M correctly run the smart contract for $\{d_1,\ldots,d_t\}$, and they propose its correct result during consensus protocol. □

Theorem 5.12 (Data availability for peers in PANTHER). *If an honest client BC_x sends a message m_{xy} to an honest pseer CP_y in PANTHER, then eventually CP_y delivers m_{xy}, so meeting the availability property (C) of Definition 5.4.*

Proof. The *availability* property (C) of Definition 5.4 follows from the assumptions that system parties are fully interconnected and the communication channels are reliable. Assume the adversary $\mathcal{A}$ discards a transaction m_{xy} sent by an honest BC_x to an honest CP_y. Relying on reliable links, BC_x will keep relay m_{xy} until CP_y delivers it Cachin et al. (2011). In addition, BC_x actually broadcasts m_{xy} to all M peers. Since the broadcast primitive is reliable, every (honest) peer that gets m_{xy} relays it once again to M, ensuring that eventually all honest peers deliver it Cachin et al. (2011). Hence, CP_y eventually delivers m_{xy}. Note the case where BC_x is byzantine. It can then broadcast to M different versions of the same transaction, e.g. m'_{xy} to some peers and m''_{xy} to others. However, during the consensus protocol, each (honest) peer proposes its received version, and the leader accepts one to be appended on ledger only if gets m'_{xy} or m''_{xy} from a quorum of $\frac{M+f_p}{2}$ peers, ensuring transaction consistency. □

5.9 Implementation

In this section, we assess the overheads that MKHE, and its MPC protocol, bring to the blockchain. In particular, we measure the computational time of MKHE algorithms and the total time taken for a party to complete a MPC decryption protocol. In order to estimate the upper-bound for a MPC protocol, we consider a threshold decryption of T-out-of-T, where partial decryptions of all the T parties are required to decrypt. Clearly, any implementation of MPC decryption with a threshold $t < |T|$ (as in Protocol 5.2) will have a lower overhead, since the parties need fewer partial decryptions and thus begin reconstruction sooner. We provide a proof-of-concept level implementation of PANTHER integrating a fully MKHE scheme (i.e., MKFHE) with a permissioned blockchain platform. As a MKFHE scheme, we select the CKKS scheme described in Section 2.3.2 and we adjust it to be multi-key. As a blockchain platform, we select the Hyperledger Fabric described in Section 3.2.4 and we adjust it to integrate the multi-key CKKS scheme. We use *Go* (Google (2009)) as programming language and we develop the multi-key CKKS as a Go module, which is then imported as a library into client applications and Hyperledger Fabric smart contracts. Section 5.9.1 presents our construction of the multi-key CKKS. In Section 5.9.2, we then report our experimental results, showing the computational and message overheads introduced by the multi-key CKKS in PANTHER.

5.9.1 Multi-Key CKKS

In PANTHER, we construct a *multi-key CKKS* scheme that performs homomorphic computations, both additions and multiplications, with ciphertexts encrypted under different keys. The key characteristic of CKKS is that it works with real and complex numbers and the computations return an approximate result. For example, let ciphertexts c' be the encryption of the real value $v' = 5.20$, c'' be the encryption of $v'' = 2.00$ and c be the output of their addition $c' + c''$. Then, the decryption of c can be a value like 7.198, which is a good approximation of the sum between v' and v''.

Building upon of the CKKS, we propose a *Multi-Key CKKS* scheme (*MK-CKKS*) where a multi-key ciphertext c is of the form $(c_0, c_1, \ldots, c_t)$, where t is the number of associated parties and c_i's are elements of the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. We use the same notation as defined in Section 2.3.2.1, thus n is a power of two, q is an integer called the coefficient modulus, and $X^n + 1$ is an irreducible polynomial called the polynomial modulus. Also, we use the same RLWE-based distributions as defined in Section 2.3.2.4, thus $\mathcal{R}_3$ is the private key distribution with integer coefficients in $\{-1, 0, 1\}$, $\mathcal{X}_\sigma$ is the error distribution of variance σ^2 and $\mathcal{U}(\mathcal{R}_q)$ is a uniform random distribution over $\mathcal{R}_q$.

In MK-CKKS, each jth party holds its own set of HE keys, defined as in CKKS. The private key sk_j is a polynomial sampled from $\mathcal{R}_3$ and the public key consists of a pair of polynomials $pk_j = (pk_{j,1}, pk_{j,2}) \in \mathcal{R}_q^2$ where:

$$pk_{j,1} = [-a \cdot sk_j + e']_q$$
$$pk_{j,2} = a \leftarrow \mathcal{U}(\mathcal{R}_q)$$

The polynomial a is sampled uniformly from $\mathcal{R}_q$, and polynomial e' is a random error sampled from $\mathcal{X}_\sigma$. The evaluation key consists of a pair of polynomials $ek_j = (ek_{j,1}, ek_{j,2}) \in \mathcal{R}_{P \cdot q}^2$ where P is a big integer and:

$$ek_{j,1} = [-b \cdot sk_j + e'' + P \cdot sk_j^2]_{P \cdot q}$$
$$ek_{j,2} = b \leftarrow \mathcal{U}(\mathcal{R}_{P \cdot q})$$

The polynomial b is sampled uniformly from $\mathcal{R}_{P \cdot q}$, and polynomial e'' is a random error sampled from $\mathcal{X}_\sigma$.

A fresh ciphertext in MK-CKKS, generated by a jth party to encrypt plaintext m, is of the form $c = (c_0, c_j) \in \mathcal{R}_q^2$, where $c_0 = [pk_{j,1} \cdot u + m + e_1]_q$, $c_j = [pk_{j,2} \cdot u + e_2]_q$, $u \leftarrow \mathcal{R}_3$ and $e_1, e_2 \leftarrow \mathcal{X}_\sigma$. The fresh ciphertext can only be decrypted by sk_j. In contrast, the multi-key ciphertext $c \in \mathcal{R}_q^{t+1}$, resulting from an evaluation, can be decrypted by the private keys $sk_1, \ldots, sk_t$ of the t participants so that

$c_0 + c_1 \cdot sk_1 + \cdots + c_t \cdot sk_t$ is a randomised encoding of the plaintext. Hence:

$$\begin{aligned}
\text{DEC}(c, sk_1, \ldots, sk_t) &= \left[c_0 + \sum_{i=1}^{t} c_i \cdot sk_i \right]_q \\
&= \left[m + \sum_{i=1}^{t} (pk_{i,1} \cdot u_i + e_{i,1}) + \sum_{i=1}^{t} (pk_{i,2} \cdot u_i + e_{i,2}) \cdot sk_i \right]_q \\
&= \left[m + \sum_{i=1}^{t} ((-a_i \cdot sk_i + e'_i) \cdot u_i + e_{i,1}) + \sum_{i=1}^{t} (a_i \cdot u_i + e_{i,2}) \cdot sk_i \right]_q \\
&= \left[m + \sum_{i=1}^{t} (e'_i \cdot u_i + e_{i,1}) + \sum_{i=1}^{t} e_{i,2} \cdot sk_i \right]_q \approx m
\end{aligned}$$

Although a multi-key ciphertext can be decrypted as in the above formula, in practice it is not reasonable to assume that there is a party holding multiple private keys. Therefore, in MK-CKKS the t parties involved in a multi-key ciphertext $c \in \mathcal{R}_q^{t+1}$ run a MPC protocol to decrypt c. Particularly, once obtained c, each ith party partially decrypts c with sk_i generating the partial ciphertext $pc_i = [c_i \cdot sk_i + e]_q \in \mathcal{R}_q$, where c_i is the polynomial in c associated to the ith party and the error $e \leftarrow \mathcal{X}_\sigma$. Then, each party broadcasts its pc_i to other $t-1$ parties. Upon the delivery of other parties partial ciphertexts, each party merges them locally by computing $\left[c_0 + \sum_{i=1}^{t} pc_i\right]_q$ and retrieves the plaintext m.

Similarly to CKKS, the MK-CKKS scheme provides the encoding, decoding, relinearization and rescaling procedures. The encoding is used to convert a complex value $\mathbf{z} \in \mathbb{C}^{\frac{n}{2}}$ into a polynomial $m \in \mathcal{R}$, while the decoding is used in reverse to convert m back into $\mathbf{z}$. During encoding $\mathbf{z}$ is multiplied by a scale factor Δ to preserve approximation accuracy, and during decoding m is multiplied by $\frac{1}{\Delta}$. Both the relinearization and the rescaling are applied during a multiplication between ciphertexts c' and c'' producing a ciphertext $\tilde{c}$. The relinearization is used to reduce the magnitude of $\tilde{c}$, i.e. the number of polynomials it contains, to the standard form with linear decryption structure. The rescaling instead is used to reduce the scale factor of $\tilde{c}$ from Δ^2 to Δ. In addition to this, the rescaling also reduces the coefficient modulus q of $\tilde{c}$ from a level l to a level $l-1$, i.e. from q_l to q_{l-1}.

Note that, as in CKKS, the most expensive operation of MK-CKKS is homomorphic multiplication which consists of three steps: product, relinearization and rescaling. Consider the ciphertexts $(c'_i)_{0 \le i \le t}$ and $(c''_j)_{0 \le j \le t}$ both in $\mathcal{R}_{q_l}^{t+1}$, representing the encryptions of m' and m'' respectively. It first computes their product $(c_{i,j} = c'_i \cdot c''_j)_{0 \le i,j \le t}$ representing an encryption of $m' \cdot m''$. Then, it applies the relinearization by converting $(c_{i,j})$ back to $\mathcal{R}_{q_l}^{t+1}$. The total complexity of relinearization grows quadratically with t since the process should be repeated on $c_{i,j}$ for all $1 \le i, j \le t$. Finally, it scales down such relinearized ciphertext multiplying by

Δ^{-1} and produces an output ciphertext in $\mathcal{R}_{q_{l-1}}^{t+1}$, which still represents an encryption of $m' \cdot m''$.

Formally, a MK-CKKS scheme is defined as follow:

Definition 5.13 (*MK-CKKS scheme*)**.** Let $d > 0$ be a fixed base for scaling in approximate computations and q_0 be a modulus, and let $q_l = d^l \cdot q_0$ for $0 < l \leq L$. Let $\mathcal{X}_\sigma$ be the discrete Gaussian distribution of variance σ^2. Let $N = \{P_1, \ldots, P_{|N|}\}$ be a group of parties where each P_j holds a MK-CKKS scheme $\mathcal{E}_j$. The MK-CKKS scheme of P_j is a tuple of algorithms $\mathcal{E}_j = (\text{KEYGEN}, \text{ENC}, \text{DEC}, \text{PARTIAL-DEC}, \text{MERGE-DEC}, \text{ADD}, \text{MUL}, \text{RSC})$ with the following syntax:

- $\text{KEYGEN}(1^\lambda) \rightarrow (pk_j, sk_j, ek_j)$: For a security parameter λ, outputs a public key pk_j, a private key sk_j, and a (public) evaluation key ek_j.
- $\text{ENC}(pk_j, m) \rightarrow (c)$: For a given polynomial $m \in \mathcal{R}$, outputs a ciphertext $c \in \mathcal{R}_{q_L}^2$.
- $\text{DEC}(sk_j, c) \rightarrow (m)$: For a ciphertext $c \in \mathcal{R}_{q_l}^2$ encrypted by the public key pk_j at level l and a private key sk_j, outputs a polynomial $m \in \mathcal{R}$.
- $\text{PARTIAL-DEC}(sk_j, c) \rightarrow (pc_j)$: For a multi-key ciphertext $c \in \mathcal{R}_{q_l}^{t+1}$ at level l of t parties $\{P_i\} \in N$ and a private key sk_j where $1 \leq j \leq t \leq |N|$, samples an error $e \leftarrow \mathcal{X}_\sigma$ and outputs a partial ciphertext $pc_j = [c_j \cdot sk_j + e]_{q_l}$ where $pc_j \in \mathcal{R}_{q_l}$.
- $\text{MERGE-DEC}(c, pc_1, \ldots, pc_t) \rightarrow (m)$: For a multi-key ciphertext $c \in \mathcal{R}_{q_l}^{t+1}$ at level l of t parties $\{P_i\} \in N$ and t partial ciphertexts $pc_1, \ldots, pc_t$ where pc_i belongs to P_i, outputs a polynomial $m = \left[c_0 + \sum_{i=1}^{t} pc_i\right]_{q_l}$ where $m \in \mathcal{R}$.
- $\text{ADD}(c', c'') \rightarrow (c)$: For two ciphertexts c' and c'' in $\mathcal{R}_{q_l}^{t+1}$ at level l, outputs a ciphertext $c = [c' + c'']_{q_l}$ in $\mathcal{R}_{q_l}^{t+1}$.
- $\text{MUL}(ek_1, \ldots, ek_t, c', c'') \rightarrow (c)$: For t evaluation keys $ek_1, \ldots, ek_t$ of t parties $\{P_i\} \in N$ associated to two ciphertexts c' and c'' in $\mathcal{R}_{q_l}^{t+1}$, computes $\tilde{c} = [c' \cdot c'']_{q_l}$, relinearizes $\tilde{c}$ with $ek_1, \ldots, ek_t$ and outputs a ciphertext $c \in \mathcal{R}_{q_l}^{t+1}$.
- $\text{RSC}(c) \rightarrow (c')$: For a ciphertext $c \in \mathcal{R}_{q_l}^2$ at level l resulting from a multiplication, outputs a ciphertext $c' \leftarrow \lfloor \frac{q_{l'}}{q_l} c \rceil$ in $\mathcal{R}_{q_{l'}}^2$ where $l' < l$, i.e. c' is obtained by scaling $\frac{q_{l'}}{q_l}$ to the entries of c and rounding the coefficients to the closest integers.

We implement MK-CKKS in Go by leveraging the *Lattigo library v2.3.0* Lat (2021) implementing a CKKS scheme, and the work of Kim et al. (2022) proposing a multi-key variant of CKKS named *KKLSS*. In particular, we use the CKKS scheme of Lattigo as a basis, [7] and on top of it we construct the MK-CKKS scheme similar to KKLSS. Indeed, we use the same optimisations proposed by Kim et al. (2022), which,

[7] We import the packages `ckks`, `rlwe` and `ring` from Lattigo.

through the homomorphic gadget decomposition technique, improve the relinearization of HE multiplications and thus their performance. However, in KKLSS the decryption of a multi-key ciphertext c is not carried out via an MPC protocol, but rather via a function that locally takes as input the private keys of the parties involved in c. That is, to decrypt $c \in \mathcal{R}_{q_l}^{t+1}$ they implement the algorithm $\text{DEC}(sk_1, \ldots, sk_t, c)$. This algorithm requires the party executing it to hold the t private keys, which is not reasonable in an untrusted multiparty setting such as a permissioned blockchain.

Hence, differently from KKLSS, our MK-CKKS implementation introduces the functions PARTIAL-DEC and MERGE-DEC as per Definition 5.13 to carry out a MPC decryption protocol. They allow a party to generate a partial decryption to be broadcast and to merge received partial decryptions, respectively. This contribution enables parties to carry out a MPC decryption whenever they need to decrypt a multi-key ciphertext, which is fundamental for PANTHER. Also, we view this contribution as of independent interest since our MK-CKKS can be applied in any multiparty scenarios where users do not want to share their private key.

5.9.2 Experimental results

All experiments are performed on a machine with 1 CPU Intel(R) Core(TM) i9-9980HK @ 2.40GHz and 12GB RAM running Ubuntu 22.04 LTS. We set up a channel in Hyperledger Fabric composed by four computing peer [8] participating in the blockchain network. We implement in Go four client applications for four different users, and a Hyperledger Fabric smart contract for computing HE additions and multiplications on ciphertexts. The clients use the Hyperledger Fabric Go Gateway [9] to connect and interface with peers, and the smart contract is installed in every peer of the network.

The MK-CKKS Go module is included in the clients and the smart contract. Specifically, we import the MK-CKKS algorithms for keys generation, encryption, decryption, partial-decryption and merge-decryption in clients. Whereas, the MK-CKSS algorithms for addition and multiplication are imported in the smart contract. According to the homomorphic encryption standard Albrecht et al. (2018), for the MK-CKKS scheme in the clients and the smart contract, we use the following parameters to achieve the security level 128 bits:

- $\log_2 n = 15$;

[8] In Hyperledger Fabric, a blockchain node can play two roles: either a peer or an orderer. The peer is responsible for transaction execution (via smart contract) and endorsement. The orderer is responsible for transaction validation and ordering (via consensus protocol). We refer to computing peer as a Hyperledger Fabric node that plays both roles.

[9] https://pkg.go.dev/github.com/hyperledger/fabric-gateway/pkg/client

		P_1	P_2	P_3	P_4	Smart contract
Keys generation		629.16 ms	607.16 ms	698.97 ms	563.04 ms	
Encryption		75.39 ms	86.86 ms	88.36 ms	69.02 ms	
P_1 invokes addition						29.67 ms
P_1 starts a MPC protocol	**partial decryption**	37.96 ms	39.79 ms	39.09 ms	30.50 ms	
	broadcast	153.88 μs	138.15 μs	168.08 μs	77.92 μs	
	merge	95.87 ms	105.39 ms	95.89 ms	97.88 ms	
	total	2.88 s	1.57 s	1.53 s	1.49 s	
P_3 invokes multiplication						2.73 s
P_3 starts a MPC protocol	**partial decryption**	33.97 ms	32.28 ms	35.89 ms	37.23 ms	
	broadcast	112.65 μs	96.96 μs	82.32 μs	157.13 μs	
	merge	89.18 ms	89.74 ms	101.63 ms	96.64 ms	
	total	1.75 s	1.77 s	3.07 s	1.68 s	
P_4 invokes subtraction						30.67 ms
P_4 starts a MPC protocol	**partial decryption**	30.83 ms	27.06 ms	28.27 ms	35.89 ms	
	broadcast	84.96 μs	96.66 μs	87.04 μs	97.95 μs	
	merge	99.72 ms	92.20 ms	100.38 ms	97.39 ms	
	total	1.63 s	1.58 s	1.61 s	3.14 s	

TABLE 5.2: PANTHER performance under four users, invoking smart contracts to add, multiply and subtract ciphertexts. ms $= 10^{-3}$s and μs $= 10^{-6}$s

	pk	**ek**	**c**	$\mathbf{c_{mul}}$	$\mathbf{c_{add}}$ **and** $\mathbf{c_{sub}}$	$\mathbf{pd_{mul}}$	$\mathbf{pd_{add}}$ **and** $\mathbf{pd_{sub}}$
Size	17.66 MB	370.96 MB	15.31 MB	30.14 MB	38.28 MB	6.03 MB	7.66 MB

TABLE 5.3: Size of the MK-CKKS data types in megabyte MB

- $\log_2 q = 880$;
- $\Delta = 2^{54}$;
- $\sigma = 3.2$.

The key distribution $\mathcal{R}_3$ samples each coefficients from $\{0, \pm 1\}$ with probability 0.25 for each of -1 and 1 and with probability 0.5 for 0. The error distribution $\mathcal{X}_\sigma$ is a discrete Gaussian distribution of variance σ^2.

In our experiments, each peer is executed in a separate thread inside a docker container (Merkel (2014)). We simulate a scenario where at the outset, four users $\{P_1, P_2, P_3, P_4\}$, through their respective clients, send to the blockchain real number values encrypted with their personal keys. Later on, P_1 invokes the smart contract to sum the collected ciphertexts. The blockchain peers perform the HE computation and send the evaluated ciphertext back to P_1, which starts a MPC decryption protocol with P_2, P_3, P_4 to obtain the result. Later on, P_3 and P_4 invoke the smart contract to multiply and subtract [10] the collected ciphertexts, respectively. Upon the delivery of the corresponding evaluated ciphertexts, P_3 and P_4 start two other distinct MPC decryption protocols. The execution time of this evaluation is given in Table 5.2, which is derived from dataset Zanfino (2024). Table 5.2 reports:

[10]Note that, in CKKS, and thus in MK-CKKS, the subtraction is performed via addition, e.g. $2.3 + (-1.1) = 1.2$.

- for each user the computational time of MK-CKKS keys generation, encryption and MPC decryption protocol;
- for the smart contract the computational time of MK-CKKS addition, multiplication and subtraction.

Table 5.3 reports the size of MK-CKKS data being exchanged between a client and the blockchain, i.e. keys pk and ek, fresh ciphertext c, evaluated ciphertexts c_{mul}, c_{add}, c_{sub} and partial decryptions pd_{mul}, pd_{add}, pd_{sub}. All data sizes are in the order of megabytes, among which ek is the largest, with a size of 370MB. The experimental results show that the MK-CKKS algorithms have a computational cost in the order of milliseconds. The keys generation, encryption, partial decryption and merge have an average for the four users of 624.58ms, 79.90ms, 34.06ms and 96.83ms respectively. For the MPC decryption protocol, the participants take on average 1.975s total time to complete it. Particularly, the users initiating the MPC decryption take an average of 3.03s to complete the protocol, while the other involved users take an average of 1.62s. The difference between these two average times of about 1.41s is due to the fact that the initiator does the partial decryption and broadcast alone, while the other users do them in parallel (as shown in Figure 5.2).

To sum up, considering a single computation request, and hence a single execution of MPC decryption, the overhead introduced by MK-CKKS in a PANTHER client is on average 1.975s. On the other hand, the overhead introduced by MK-CKKS in a PANTHER smart contract is on average 30.17ms for addition and subtraction, and around 2.73s for multiplication.

5.10 Discussion

In this chapter we presented PANTHER, which integrates the MKHE model within a permissioned blockchain to provide stronger privacy properties on stored data, while still allowing smart contracts to execute functions on ciphertexts. Specifically, we adopted the MKHE model to enable homomorphic computations on data encrypted under different keys, thus overcoming the limitation exhibited by traditional single-key HE schemes. We designed an architecture where the MKHE algorithms (i.e., key generation, encryption, decryption and evaluation) are deployed on clients and blockchain peers. The decryption is realised in PANTHER as a MPC distributed protocol that runs among clients, and the evaluation is performed by peers' smart contracts. In particular, we proposed a novel MPC decryption protocol that leverages the blockchain integrity property to deliver decryption correctness against a byzantine adversary. Furthermore, we proved that PANTHER achieves security in the face of a byzantine adversary, preserving data privacy, integrity and availability.

We implemented PANTHER, and we assessed the overheads that MKHE, and its MPC decryption protocol, introduced to the blockchain. Specifically, we implemented MK-CKKS, a multi-key variant of the CKKS scheme, and we included it in the clients and smart contracts of the blockchain. The results of our experimental evaluation show that the overhead introduced by MK-CKKS in the smart contracts of blockchain peers is small for addition, about 29.67ms, and higher but feasible for multiplication, about 2.73s. This demonstrates that the impact of MKHE on the blockchain is almost zero for performing addition and reasonably low for multiplication. On the other side, the overhead introduced by MK-CKKS in clients is small for keys generation and encryption, averaging 624.58ms and 79.90ms respectively, and higher but feasible for performing a MPC decryption. Indeed, the total time to complete a MPC decryption is on average 1.975s. In particular, the client initiator completes the protocol in an average of 3.03s, while the other clients involved in average of 1.62s. This is mainly due to the time taken to exchange messages, rather than the MKHE algorithms executed during the protocol, i.e. partial decryption and merge, which on average take 34.06ms and 96.83ms respectively.

Unfortunately, the MPC protocol cannot be avoided in a system where a user encrypts her data with her personal keys, but having such a system yields the following advantages:

- the user's data cannot be decrypted by others, thereby reaching the highest level of privacy;
- the user is autonomous and does not need to trust an external party, who shares a common public key for the system and keeps the matching private key for itself.

Furthermore, if PANTHER implemented THE instead of MKHE, the performance of the clients would be worse, since the THE model comprises two MPC protocols, one for key generation and one for decryption. As we have observed, the MPC protocol, and in particular the exchange of messages, is the factor that brings the most overhead, and running it twice significantly affects client performance.

As a future work, we plan to investigate the feasibility of enhancing the security of PANTHER by tolerating stronger adversaries that can control more than $f_t < |T|/3$ clients. For example, we will explore mechanisms to dynamically involve more clients besides those in T.

Chapter 6

SHERLOCK: Sharding permissioned Blockchain

As emerged from our study in Chapter 3, the consensus protocol is the factor that most impacts blockchain performance. This direct proportion between the two is due to the fact that the consensus protocol is responsible for processing all incoming transactions, and ensuring that all nodes in the network agree on their ordering. The more transactions are processed over time, the better the blockchain system performs. Permissionless blockchains typically accommodate a large number of network nodes, without entry restrictions. Hence, they employ lottery-based consensus algorithms, in which a leader is randomly elected among the network to propose the order of transactions. The downside of this approach is that both the leader election process and the transactions validation across the entire network are time-consuming. This leads to performance degradation, which is more pronounced in blockchains employing PoW, due to the computational-intensive mining process. Conversely, this aspect is mitigated in permissioned blockchains. They are operated by authenticated parties and usually the network size is much smaller than permissionless where anyone can participate. Besides, only a restricted number of nodes are involved in the consensus protocol. These specifications allow permissioned blockchains to employ voting-based consensus algorithms, in which the leader is elected by the consensus nodes through voting. As a result, permissioned blockchains take lesser time to reach consensus: as soon as the leader receives a majority of acknowledgements from consensus nodes the proposed block is appended to the ledger.

Notwithstanding permissioned blockchains exhibit better performance, they present scalability issues. These occur when the volume of transactions to be processed grows significantly and the consensus nodes become saturated. In traditional distributed systems, this problem is solved by scaling out, i.e. adding nodes to the network in order to distribute and balance the workload. However, in voting-based protocols, the

consensus nodes work on the same transactions. As in PBFT (Castro and Liskov (1999)), when a new round of consensus begins, the leader sends the replicas a set of transactions to be ordered. Each replica orders this set and sends it back to the leader, which then waits to receive a majority of them to create and commit the final block. Consequently, scaling out in a permissioned blockchain means more messages exchanged during consensus and more time to agree on the ordering of transactions, worsening performance instead of improving it.

To solve this scalability problem, in this chapter we propose to apply the *sharding* technique in permissioned blockchains. Originally, sharding has been used by modern data storage systems, e.g. Amazon Dynamo (DeCandia et al. (2007)), to separate large databases in smaller parts, known as shards, in such a way to boost system scalability and performance. We leverage this concept to divide the consensus nodes of a permissioned blockchain network into committees. The incoming transactions to order are then spread among these committees, which concurrently run instances of the consensus protocol. Hence, we present *SHERLOCK (SHarding pERmissioned bLOCKchain)*, a scalable permissioned blockchain based on sharding. SHERLOCK provides a novel two-layer ring-based architecture for nodes engaged in the consensus, and uses the sharding technique to form parallel committees over it. When the workload grows, SHERLOCK is able to scale-out by adding nodes and creating new committees, while maintaining adequate performance.

Contributions

The novel contribution of this work consists in enhancing the scalability of a permissioned blockchain by using the sharding technique. The other contribution provided are:

- a novel two-layer ring-based network topology for consensus nodes of a permissioned blockchain;
- a qualitative analysis of SHERLOCK, where we compare the two offered versions and its communication complexity with the traditional PBFT protocol;
- the implementation of SHERLOCK in Go;
- a comparison of SHERLOCK and PBFT performance, in terms of throughput and latency, under the same network configuration and workload, where we demonstrate that SHERLOCK outperforms PBFT;
- a security analysis of SHERLOCK, where we prove that it preserves the system's security guarantees after the division into committees, and it is able to withstand against the single shard takeover attack.

Chapter structure

Section 6.1 defines the system model we consider. In Section 6.2, the scalability problem of the permissioned blockchain is highlighted, describing how scaling-out worsens performance instead of improving it. Section 6.3 describes the sharding technique. In Section 6.4 we present the two-layer ring-based SHERLOCK architecture, which comes in two versions: without and with overlapping committees. A qualitative analysis of SHERLOCK is discussed in Section 6.5, where we compare the efficiency of the two versions and the communication complexity of SHERLOCK with that of PBFT. In Section 6.6 we implement SHERLOCK and PBFT, and we evaluate them in terms of throughput and latency, proving that SHERLOCK outperforms PBFT. A security analysis of SHERLOCK is provided in Section 6.7, discussing how it resists to the single shard takeover attack. Section 6.8 presents a literature review on the application of sharding into blockchain. Finally, Section 6.9 concludes and discusses the work.

6.1 System model

We consider a system in which a consortium, composed by multiple *organizations*, sets up a private permissioned blockchain network. We consider that the organisations do not trust each other, but we assume that a majority of them, and precisely 2/3, are honest. Organizations divide the network fairly and democratically, so that each of them supplies the same number of blockchain nodes. We consider the roles separation operated in Hyperledger Fabric for blockchain nodes (see Section 3.2.4). Hence, we refer to the nodes that hold the blockchain ledger and run smart contracts as *peers*. Whereas, we refer to the nodes that participate in the consesus protocol as *orderers*. Moreover, we refer to members of organizations as *users*. Depending on the position a user covers within her organization, she can have different rights and permissions for requesting operations to the blockchain. Each user can interact with the blockchain network through an application software (e.g., web or mobile app). We refer to such application software as *client*. In particular, clients enable users to submit transactions towards the blockchain system and handle its response. Transaction types comprise:

- `read`: retrieving data from the ledger;
- `write`: storing data into the ledger;
- `sc-call`: invoking smart contract computations.

Peers are responsible for handling client requests. If the request is a `read`, the peer returns the desired data to the client. If the request is either `write` or `sc-call`, the peer

executes the transaction locally without updating the ledger state, and sends its output to other peers for endorsement. Once the peer receives transaction endorsements from the majority of peers[1], it sends the transaction to orderers to be included in a block. Orderers are responsible for ordering the incoming transactions. We consider that the PBFT consensus algorithm is in place, and accordingly that among the orderers there is a *leader* node administering the consensus instance. We refer to the remaining orderers as *replicas*, in charge of ordering the set of transactions they receive from the leader. We consider, as in PBFT, that the number of orderers is $n = 3f + 1$ (Castro and Liskov (1999)), where f is the number of byzantine orderers, and that each organization equally participates in n, i.e. each organization supplies the same number of orderers in the consensus protocol. This means that n is a multiple of the number of organizations. Orderers proceed in rounds: in each round the leader starts a consensus instance with replicas and creates a block containing ordered transactions. The block is then broadcasted to the peers to be appended to the ledger.

Being the setting permissioned, both clients, peers and orderers are authenticated: each of them has assigned a digital certificate issued by a trusted CA, that contains a set of cryptographic credentials, i.e. a pair of public-private keys. The certificate public key, along with other attributes relating to its holder (e.g., its hostname and the organization it belongs to), are publicly verifiable. A client uses its private key to cryptographically sign the transactions it submits to the blockchain. Peers identify the client by checking its signature, and authorise its requests by checking its permissions. A peer uses its private key to sign the transactions it handles during the endorsement phase. An orderer instead uses its private key to sign the messages it sends during the consensus phase.

We assume that every pair of system parties is connected by a bidirectional link, and that these point-to-point links are reliable in the face of crashing parties, i.e. a correct party eventually delivers a message sent to it by another correct party (Cachin et al. (2011)). In addition, we assume parties use reliable broadcast primitives for one-to-many communications, i.e. if a correct party delivers a message then eventually all correct parties deliver it (Cachin et al. (2011)). We consider computationally bounded adversaries, i.e. PPT adversaries, that can delay communication by dropping some message retransmissions, but eventually the intended recipient delivers the message. We consider that the adversary behaviour is byzantine, i.e. the corrupted parties can deviate from their prescribed protocol. The adversary can corrupt either peers or orderers. A corrupted peer can tamper with handled transactions or its ledger state. However, these information are checked by the other peers in the endorsement phase. A corrupted orderer can try to subvert the consensus protocol. We refer to the PBFT security (Castro and Liskov (1999)),

[1]Note that in Hyperledger Fabric it is possible to specify the number of peers e engaged in endorsement, and among them the number of approvals a required to endorse a transaction. Considering p the number of system peers, the relationship is $p \geq e \geq a \geq \frac{p}{2} + 1$.

tolerating up to $f < N/3$ corrupted orderers, under the assumption that the number of orderers N is greater or equal to $3f + 1$.

6.2 Problem statement

Considering the system model of Section 6.1, the described permissioned blockchain, implementing the PBFT consensus algorithm, presents a scalability problem. When clients significantly increase the number of submitted transactions, the orderers become overloaded and the system saturates, degrading performance. Typically, in the blockchain, performance are measured under two metrics:

- *latency*: the time that an interactive system takes to respond to a user request. In the blockchain it is measured from the moment when a user sends a transaction to when she is notified that the transaction has been stored on the ledger.
- *throughput*: the average number of transactions processed per unit of measured time. In the blockchain it is calculated by increasing the number of transactions until the system saturates, i.e. it is unable to process an incoming transaction.

Latency includes the time required by peers to execute a transaction and by orderers to run the consensus protocol. With reference to these two metrics, what actually happens as client demands increase is that latency grows, and throughput gradually decreases until it reaches zero at system saturation. When this performance degradation occurs, there are two methods a system can apply to cope with the growing workload: *scale up* or *scale out*. The former consists of expanding the resources of a node, whereas the latter consists of adding nodes to the network. Scaling up can help peers to run smart contracts faster, improving latency up to a certain extent, constrained by existing hardware capabilities. On the other side, it is useless to scale up orderers since PBFT is communication-bound, i.e. latency and consequently throughput are bound to the number of messages exchanged during a consensus round. By contrast, scaling out orderers is counterproductive: increasing the number of replicas leads to a higher message overhead and a longer time to reach agreement on transactions. As a result, the scale out method lengthens latency and worsens throughput. Therefore, neither scale up nor scale out are successful in improving the scalability of the permissioned blockchain to raise its performance.

6.3 Sharding technique

Sharding is a technique used by distributed systems to improve scalability and raise performance levels. It has been largely applied in data storage systems to handle huge

databases, including Amazon Dynamo (DeCandia et al. (2007)), Apache Cassandra Lakshman and Malik (2010) and Google BigTable Chang et al. (2008), and also in file sharing system to handle data connection, as in BitTorrent Qiu and Srikant (2004). Sharding consists of dividing a very large system into much smaller and easily manageable parts, called *shards*. Indeed, the term shard represents a small part of the whole set. When applied to data storage system, sharding enables to split a database into multiple smaller tables, assigned to different network nodes. All tables share the same schema, i.e. they have the same columns, but each table contains unique rows. In face of growing data to store, this technique helps the system to scale-out, adding new nodes to host new tables, thus increasing the parallelisation and avoiding memory saturation. As side-effect, since each node has only a shard of the data, this also helps to speed up the query processing and support more traffic, improving performance. There are different types of sharding that vary according to how new data are assigned to the network nodes (i.e., to which table a new row is assigned). They are:

- *hash-based sharding*: hash functions are used to represent each table with unique identifier (a.k.a, hash digest). An incoming data item is then plugged into the hash function, and its output is mapped with the matching table's hash digest.
- *range-based sharding*: a column is chosen, and tables are split according to ranges of the column's values, either in numeric or alphabetic order. An incoming data item is assigned to a table based on its value of the chosen column.
- *directory-based sharding*: tables are split according to the distinct values of a chosen column, so that each table stores rows with the same column value. An incoming data item is assigned to a table based on its value of the chosen column.

Generally, hash-based is preferable for dividing data evenly over nodes, range-based is preferable when a specific column is queried the most, and directory-based is preferable when the values in a column have many duplicates.

6.4 SHERLOCK architecture

In this section we present the SHERLOCK architecture. Particularly, we first present its two-layer ring-based topology for nodes engaged in the consensus protocol (Section 6.4.1), and then we describe a dual approach with overlapping committees (Section 6.4.2). In Section 6.4.3, we outline how the two versions behave when a node crashes.

6.4.1 Two-layer ring-based architecture

In SHERLOCK, we apply the sharding technique to split the orderers into *committees*, whereby each committee runs the PBFT algorithm with its own leader and its set of replicas. Inspired by the ring architecture of the *Chord* protocol Stoica et al. (2001, 2003), we arrange these committees over a two-layer ring-based topology. Specifically, all committees but one are positioned in an *outer ring*. The remaining committee instead stands alone on a *inner ring*. The outer and inner rings constitute the two-layer architecture for orderers. Committees in the outer ring are responsible for concurrently handling the workload (i.e., the incoming transactions): each of them performs consensus on a shard of the workload, and sends the resulting generated block to the inner ring. The committee of the inner ring is instead responsible for providing a total order to the provisional blocks it receives from the outer ring. Adding new orderers to this topology means creating new committees on the outer ring, and thus raising the parallelisation of consensus instances. As per PBFT configuration, we set the size of each committee to $c = 3f + 1$, to tolerate f byzantine orderers. We denote by n the total number of orderers in the system, and we set n to be a multiple of c. The outer ring contains $n - c$ orderers, whereas the inner ring contains c. We consider that the number of organizations in SHERLOCK is equal to c, and that each organization supplies a node in any committee. When the need to scale-out arises due to high workload, n increases while keeping the proportions just defined. Hence, c orderers will be added in turn to form a new committee, the orderers on the outer ring continuing to be $n - c$, and the inner ring continues to be a single committee of size c. We denote by s the scaling factor, i.e. the number of committees in the system[2], which can be augmented to scale out. The number n in the system can be then calculated by $n = c \cdot s$.

We therefore use the hash-based sharding both to form the committees and to evenly distribute the workload over them. Each orderer o_i is represented with a k-bit digest identifier, resulting from computing the hash function $h(o_i)$. Orderers are then positioned in circle over the rings according to the formula: $h(o_i) \bmod 2^k$. The first $n - c$ are positioned over the outer ring, and the last c in the inner one. Each orderer is assigned to exactly one committee, and the committees are formed by grouping c orderers sequentially. That is, starting from the first orderer in the ring, which we refer for simplicity to as o_1, the first committee is formed as $C_1 = \{o_1, \ldots, o_c\}$, and so on until reaching n. Thereafter, incoming transactions are mapped to orderers of the outer ring. Any transaction t_j is as well represented with a k-bit digest identifier $h(t_j)$. By leveraging on hashing properties, t_j is mapped to the first orderer whose identifier is equal to or follows $h(t_j)$ in the hash function space. Being represented as a ring, $h(o_i)$ will be the first orderer clockwise from $h(t_j)$. Once t_j is mapped to an orderer, it enters

[2]Note that the minimum number of system committees is 3, two for enabling the parallelisation on the outer ring, and one for the inner ring. Hence, $s \geq 3$.

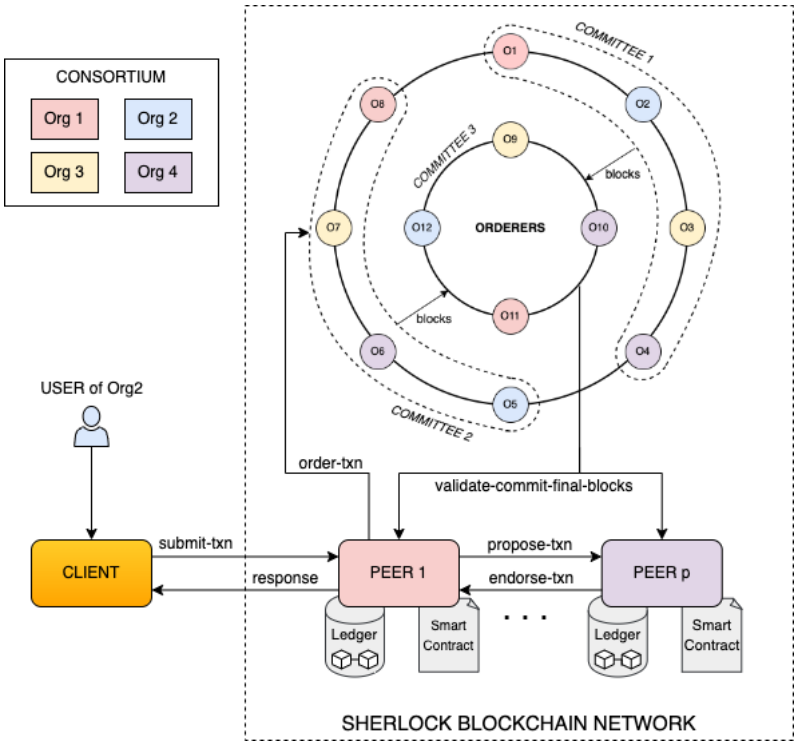


FIGURE 6.1: SHERLOCK architecture

into the pool of transactions to be ordered by the committee of which the orderer is a member. Each committee of the outer ring orders its allocated transactions in isolation, concurrently with the others. When a consensus round ends in an outer committee, a provisional block is created and sent to the committee of the inner ring. To avoid saturating the bandwidth of inner ring, the provisional block contains only the digest of processed transactions, while their content is sent in background to the endorsement peers. Upon the delivery of a provisional block, the inner ring verifies that it is signed by more than $2f + 1$ orderers of the sender committee. The inner ring committee then gives a total order to provisional blocks, and broadcasts a final block to the peers for validation and commit.

Figure 6.1 shows the architecture of SHERLOCK, in which the orderers are divided into committees and arranged over a two-layer ring-based topology. In the example there is a consortium of four organizations, each of which supplying three orderers, e.g. *Org1* supplies $\{o_1, o_8, o_{11}\}$. There are three committees (i.e., $s = 3$), two on the outer ring and one on the inner ring. Each committee is composed by $c = 3f + 1 = 4$ orderers tolerating one byzantine orderer. The number of organizations is equal to c, and they fairly participate in the consensus phase: each organization has an orderer in every committee. Indeed, in Figure 6.1 we find committee C_1 composed by $\{o_1, o_2, o_3, o_4\}$ belonging to *Org1*, *Org2*, *Org3* and *Org4* respectively. The same distribution of membership applies to committees C_2 and C_3. The leader of a committee, as per PBFT default configuration, is elected at each new consensus round,

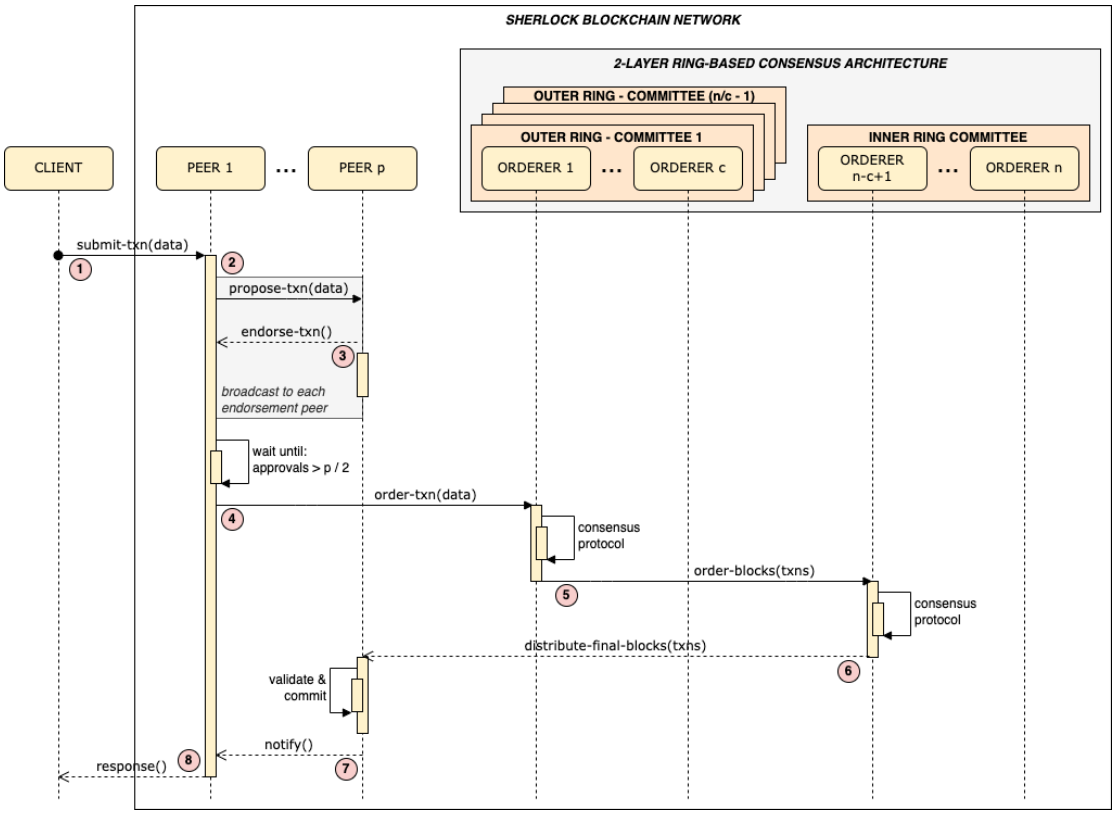


FIGURE 6.2: SHERLOCK sequence diagram

and the election mechanism is based on the sequential rotation of nodes. That is, the leader o_i of committee C_z for round r will be $i = (r \bmod c) + 1 + c(z-1)$, where i and z start from 1 and r starts from 0. For example, o_1 is leader of committee C_1 for rounds 0, 4, 8 and so on, while o_2 for rounds 1, 5, and so on. The same for applies to other committees with their respective orderers.

Figure 6.2 shows the SHERLOCK sequence diagram, where the steps are marked by a numbered pink circle. The flow starts when a client submits a transaction *txn* to peers. For simplicity of illustration, we depict *txn* arriving at a single peer, e.g. peer 1, but actually it is distributed to a set of peers, which in turn distribute to endorsement peers. The peer executes the requested *txn* and broadcasts its result to other p peers for the endorsement (step 2). The peer then waits for approvals (step 3), and once it receives a majority moves on to the next step according to the type of transaction handled. If it is a `read`, the peer jumps to step 8 and delivers the response to the client. Otherwise, the peer moves on to step 4: it calculates the transaction digest *h(txn)* and sends the *txn* to the first orderer clockwise from *h(txn)*, e.g. orderer 1 in the example. When a new consensus round starts, each committee leader of the outer ring proposes to its replicas to order the incoming transactions. In Figure 6.2 the transaction *txn* is ordered by committee 1. When the round ends, the committee sends a provisional block, containing *txn*, to the inner ring committee (step 5). Note that, during step 5, the other committees in the outer ring concurrently perform consensus on different

transactions. The inner ring committee provides a total order to received provisional blocks, and broadcasts to all peers p (step 6) a final block. Upon the delivery of final blocks, each peer validates them locally and, if valid, commits them on its ledger copy. Then each peer notifies the others (step 7) and finally sends the response to client (step 8), which can be either a confirmation that *txn* has been committed, or a notification that has been rejected as invalid.

6.4.2 Overlapping committees

As described in previous section, the orderers in SHERLOCK are split into committees, so that each orderer is assigned to exactly one committee. In this section, we relax such constraint and we propose a dual approach. We keep the same two-layer ring-based topology, with the outer ring composed by multiple committees concurrently ordering transactions, and a inner ring committee that provides total order to blocks of the outer ring. We also keep the size of a committee equal to $c = 3f + 1$ to tolerate f byzantine orderers, and the number of orderers in the outer and inner rings to be $n - c$ and c respectively. We consider as previously that the number of organizations is equal to c, and each organization supplies a node in any committee. As opposed to previous solution, we change the committees formation. Particularly, we allow an orderer of the outer ring to participate in more than one committees, and precisely in c committees. Firstly, we use the hash-based sharding to position the orderers over the two rings according to their hash digest. Differently from previous solution, we force the orderers to be sequentially positioned according to their organization membership. For example, with four organizations *Org1*, *Org2*, *Org3* and *Org4* and $c = 4$, the orderers $\{o_1, o_5, o_9, \dots\}$ belong to *Org1*, $\{o_2, o_6, o_{10}, \dots\}$ belong to *Org2* and so on. Thereafter, we form *overlapping* committees across the $n - c$ orderers of the outer ring. The first committee is created by grouping c nodes starting from the first orderer, the second committee by grouping c nodes starting from the second orderer, and so forth until the last orderer of the outer ring is reached. This is expressed by the following formula:

$$C_i = \{o_i, o_{(i \bmod n-c)+1}, \dots, o_{(i+c-2 \bmod n-c)+1}\}$$

where C_i is the ith committee on the outer ring. Figure 6.3 shows an example of the proposed approach tuned with parameters $c = 4$ and $n = 12$. On the outer ring, the first committee is $C_1 = \{o_1, o_2, o_3, o_4\}$, while the second is $C_2 = \{o_2, o_3, o_4, o_5\}$, and so forth until the last $C_8 = \{o_8, o_1, o_2, o_3\}$. Note that all organisations are involved in each committee, e.g. in C_8 the orderer o_8 belongs to *Org4*. Committees overlap each other in such a way that c consecutive committees have one orderer in common. Indeed, each orderer participates in c committees, e.g. o_1 participates to C_1, C_6, C_7, C_8. We configure the first orderer of each committee as the leader, e.g. o_1 for C_1. Consequently, a single

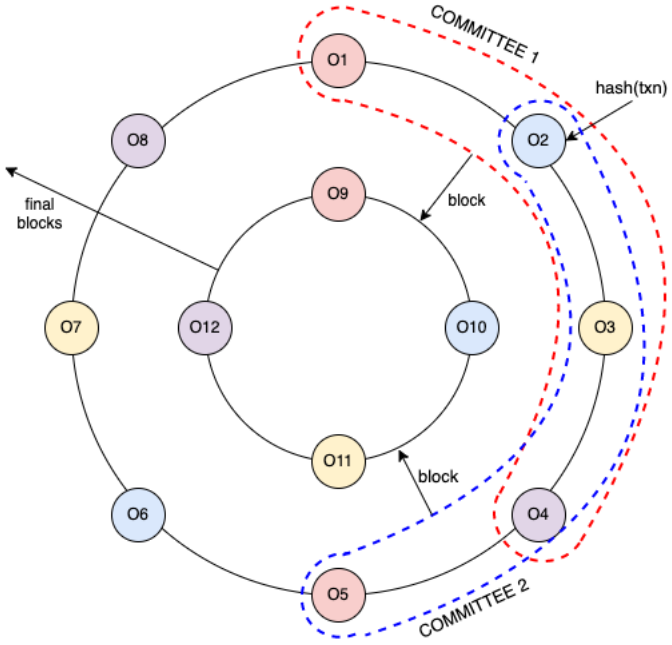


FIGURE 6.3: SHERLOCK overlapping committees

orderer acts as a leader for one committee and as a replica for the others $c-1$ it is involved in. The advantage of this overlapping approach is that we can scale out simply by adding one orderer, which causes the creation of another overlapping committee. Note that with this approach the minimum number of system orderers to enable parallel committees on the outer ring is $n \geq 2c+1$, where c nodes are dedicated to the inner ring. For example, with $c = 4$ we just need 9 system orderers, which results in 5 overlapping committees on the outer ring. Conversely, in the previous solution to have 5 committees on the outer ring we need 24 system orderers. This is because with the overlapping approach n and s are unrelated.

As regards workload distribution, it works exactly the same as the previous solution: using the sharding technique, we generate the hash digest of an incoming transaction as $h(txn)$, and we then map the transaction txn to the first orderer of the outer ring clockwise from $h(txn)$. In the previous solution the transaction would be processed by the committee to which the orderer belongs. However, in this overlapping approach, an orderer belongs to c different committees, and it is the leader in one of these. Therefore, differently from previous solution, the orderer process txn in the committee in which is leader. Then, the consensus proceeds as in the previous solution, with the outer ring committees sending provisional blocks to the inner ring committee.

Comparing the two approaches there are some aspects to note. Under the same conditions, with the same values of n and c, the overlapping approach offers more committees, and thus more parallelisation. The orders arranged on the outer ring are $n-c$ in both cases, but the committees without overlapping are $\frac{n-c}{c}$, and $n-c$ with overlapping instead. However, on the other hand, with the overlapping approach

each orderer is involved in c committees, and thus exchanges more messages, c times as many. In the next section, we evaluate how these differences between the two impact on the communication complexity of the consensus.

6.4.3 Dealing with crashing nodes

To ensure that the consortium organizations participate equally in the consensus protocol, when a nodes crashes, is to be replaced by another from the same organisation. During the replacement time, the two versions of SHERLOCK act differently. In the version without overlapping, the committee of the crashing node continues to run and order transactions. However, to guarantee that the committee is still able to tolerate f byzantine node, the replacement is required (within a time limit), as per PBFT configuration. If the crashing node is leader, then a new leader is elected from among the committee's replicas. Otherwise, if it is a replica, the committee operates as usual. In the version with overlapping, instead, each node is involved in c committees, one of which is leader and $c-1$ is a replica. When a node crashes, the number of committees in the outer ring scale in, becoming $n-c-1$. Moreover, the committees are re-formed according to the new layout of the nodes on the ring. An incoming transaction that should be mapped to the crashing node, is then mapped to its successor on the ring. Hence, it can be seen as if the committees where the crashing node is a replica continue to function, while the one where it is a leader is temporarily suspended until it is replaced. However, to provide the same level of workload distribution and consensus parallelisation, the node needs to be replaced promptly.

6.5 Qualitative analysis of SHERLOCK

In this section, we analyse the efficiency of SHERLOCK in terms of communication complexity, and we compare it qualitatively with the traditional solution of Hyperledger Fabric, implementing a single PBFT protocol for consensus. Furthermore, we compare the two versions we proposed for dividing consensus nodes into committees, i.e. without and with overlapping.

Hyperledger Fabric employs PBFT as consensus algorithm. As Figure 6.4 shows, PBFT is composed by a leader and a set of replicas, with a total number of nodes n greater or equal to $3f+1$, where f are the byzantine nodes. PBFT proceeds in rounds, and each round comprises three phase, i.e. pre-prepare, prepare and commit. When the round starts, the leader broadcasts to replicas the set of incoming transactions to be ordered (pre-prepare). Then, each replica orders the transactions and broadcasts the results to other replicas and the leader (prepare). When both the leader and the replicas receive the ordered transactions from a quorum, they broadcasts to each other

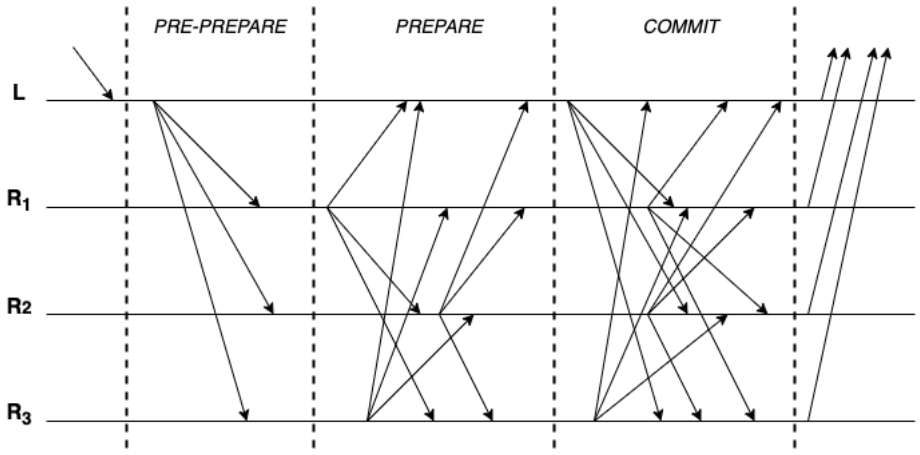


FIGURE 6.4: PBFT consensus protocol

a commit message (commit). The leader and the replicas finally send the generated block to peers. The communication complexity of PBFT is the number of messages exchanged among the leader and the replicas within a round. The number of exchanged messages during pre-prepare is $n-1$, during prepare $(n-1)^2$ and during commit $n(n-1)$, for a total of $2n^2-2n$. Hence, we have:

$$2n^2 - 2n \leq 2n^2 = O(n^2).$$

The PBFT consensus algorithm has an asymptotic cost of $O(n^2)$. On the other side, SHERLOCK splits nodes n into multiple committees, each of which runs a PBFT instance. Then, to calculate SHERLOCK communication complexity, we need to multiply the PBFT complexity by the number of committees. In SHERLOCK we denote by c the number of nodes per committee, which is generally much smaller than n, to facilitate workload distribution and create parallelisation. Hence, each committee in SHERLOCK has an asymptotic cost of $O(c^2)$. Depending on the version used, SHERLOCK provides $(\frac{n-c}{c})+1$ committees without overlapping, and $n-c+1$ committees with overlapping. Under the same system conditions, i.e. at equal nodes n, for SHERLOCK we have:

$$\left(\left(\frac{n-c}{c}\right)+1\right)c^2 = nc - c^2 + c^2 = nc = O(n); \tag{6.1}$$

$$(n-c+1)c^2 = nc^2 - c^3 + c^2 \leq nc^2 + c^2 = O(n). \tag{6.2}$$

Both versions without and with overlapping (Eq. 6.1 and Eq. 6.2, respectively) have an asymptotic cost of $O(n)$. Therefore, in a system composed by n consensus nodes, SHERLOCK exhibits a lower and then better communication complexity compared to single PBFT implementation, as in Hyperledger Fabric. This because c is much smaller than n, and in single-PBFT the communication cost grows quadratically with n, whereas in SHERLOCK c is a constant and the communication cost grows linearly with n.

As regards the two versions offered by SHERLOCK, from the Eq. 6.1 and Eq. 6.2 we can note their diversity. Considering c be the same in both settings, the overlapping version affects the communication complexity with a cost c^2, whereas the one without affects with a cost c. This because in the overlapping version each node on the outer ring is involved in c committees, and hence its communication complexity is multiplied by c. Without overlap, by contrast, a node only participates in a single committee. However, on the other hand, the overlapping version has the advantage that only one node is required to create a new committee, differently from the other which requires c nodes. To make a more thorough analysis of the two, we therefore need to introduce an additional element of comparison: the effort exerted by a node to handle transactions. Let WL be the workload, i.e. the number of incoming transactions to order. Let r be the resources, i.e. the amount of CPU, memory and bandwidth, used by a node to manage a transaction. Let ORC be the number of committees of the outer ring, and PC the number of committees in which a node participates. Under the same system conditions, i.e. at equal values of n and c, in SHERLOCK a consensus node of the outer ring has the following effort:

$$\frac{WL}{ORC} \cdot PC \cdot \frac{O(c^2)}{c} \cdot r = \begin{cases} \frac{WL}{\frac{n-c}{c}} \cdot 1 \cdot c \cdot r = \frac{WL}{n-c} \cdot c^2 \cdot r & \text{no-overlapping} \\ \frac{WL}{n-c} \cdot c \cdot c \cdot r = \frac{WL}{n-c} \cdot c^2 \cdot r & \text{overlapping} \end{cases} \tag{6.3}$$

The Eq. 6.3 shows that both versions have the same impact on a consensus node. The number of committees ORC formed in the no-overlapping version is smaller than with overlapping. Consequently, in the no-overlapping version there is less workload distribution and parallelisation, and thus a consensus node has more transactions to order. This is however balanced by the number PC of participating committees, which is c with overlapping and 1 without. As a result, in the overlapping version the communication complexity of a node is c times higher.

If we relax the constraint that n is the same in both versions (i.e., we allow them to deploy a different number of consensus nodes), and we fix instead ORC to be the same, we have from Eq. 6.3 that a node has less effort without overlapping, precisely c times less. However, the no-overlapping version needs more nodes to achieve the same amount of ORC as the overlapping version. This is evident from the following comparison:

$$\begin{aligned} &\frac{n_1 - c}{c} = ORC \Rightarrow n_1 = (ORC)c + c; && \text{no-overlapping} \\ &n_2 - c = ORC \Rightarrow n_2 = ORC + c; && \text{overlapping} \\ &(ORC)c + c > ORC + c \Rightarrow (ORC)c > ORC \Rightarrow n_1 > n_2 \end{aligned} \tag{6.4}$$

The Eq. 6.4 shows that the no-overlapping version requires a number of nodes c times bigger. This again balances the comparison between the two versions. Summarising, the two versions are equally efficient, where one offers better communication

complexity, the other offers greater workload distribution and consensus parallelisation.

6.6 Implementation and experimental results

The qualitative analysis presented in Section 6.5 shows that SHERLOCK has a better communication complexity than the PBFT consensus protocol. With the same number of nodes engaged in consensus, the communication cost for SHERLOCK is linear while for PBFT it is quadratic. The implications of such finding are that a permissioned blockchain employing SHERLOCK can sustain a higher workload and process more transactions. In this section we report experimental results that demonstrate and corroborate the complexity study conducted in Section 6.5. Particularly, we implement SHERLOCK and PBFT and we compare their performance, in terms of throughput and latency, under the same workload. For both implementations we use *Go* (Google (2009)) as programming language. Since the consensus algorithm of SHERLOCK is based on PBFT, we first develop the PBFT as a Go module and then import it as library into SHERLOCK. The Go module of SHERLOCK additionally includes a package responsible for sharding, where we implement the Chord protocol described in Stoica et al. (2001, 2003). As emerged from the study in Section 6.5, the two SHERLOCK versions, without and with overlapping committees in the outer ring, are equally performant. Thus, in this SHERLOCK implementation, we select the no-overlapping version as the approach to form committees.

All experiments are performed on a machine with 8 CPUs Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 128GB RAM running Ubuntu 22.04 LTS. For the evaluation, we set up a network of $n = 12$ nodes to both SHERLOCK and PBFT. This number is chosen considering that the no-overlapping approach is in place, the minimum number of committees for SHERLOCK is 3, and the minimum size of a SHERLOCK committee c is 4 tolerating $f = 1$ byzantine node. Hence, SHERLOCK consists of 8 nodes in the outer ring split in 2 committees and 4 nodes in the inner ring committee. Whereas, PBFT consists of a single committee with all n nodes. We implement a blockchain client in Go that generates and submits a workload of 1000 transactions per second to both SHERLOCK and PBFT.

Figure 6.5 and Figure 6.6 graphically display the performance of SHERLOCK (blue trend line) and PBFT (orange trend line) under this workload for 5 minutes. Both figures are derived from dataset Zanfino (2024). As Figure 6.5 shows, SHERLOCK processes all transactions submitted with an average throughput of 990.1 txn/s, while PBFT becomes saturated and stops processing transactions at around 75 seconds. At the moment of saturation, PBFT only manages to process 54183 transactions with an

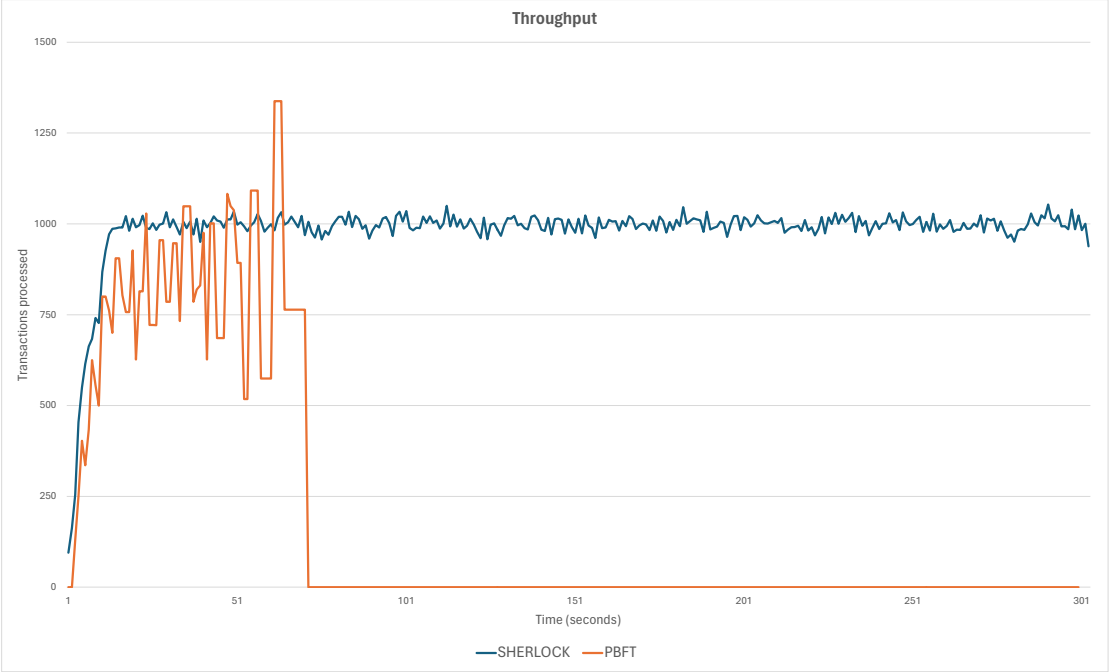


FIGURE 6.5: Throughput comparison between SHERLOCK and PBFT

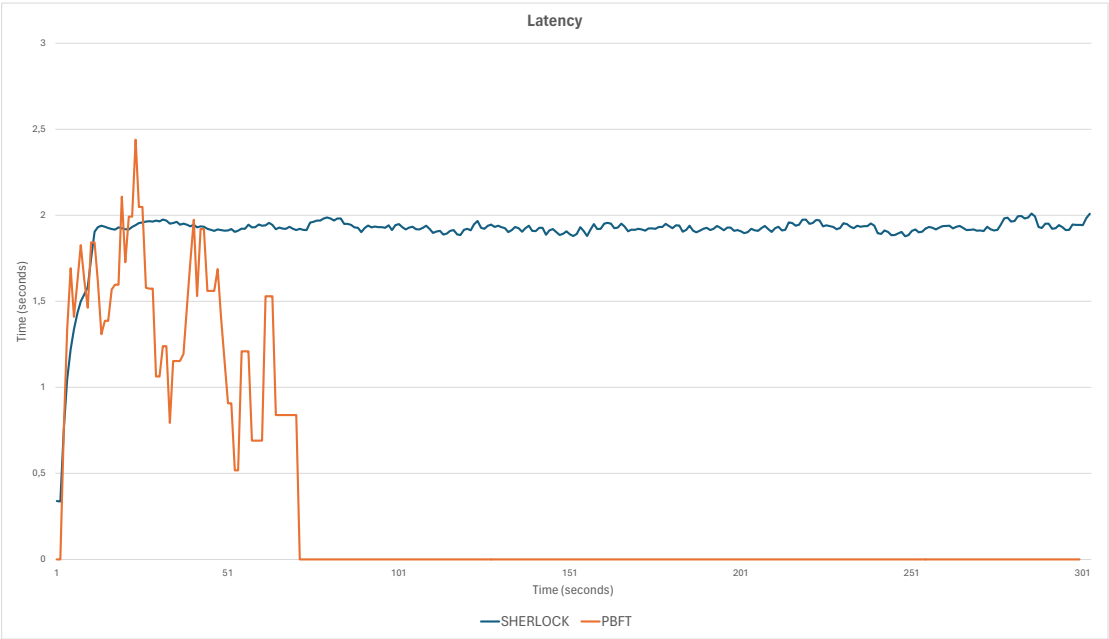


FIGURE 6.6: Latency comparison between SHERLOCK and PBFT

average throughput of 180.61 txn/s over the 5 minutes of the evaluation. As Figure 6.6 shows, SHERLOCK has a constant latency with an average per transaction of 1.937 seconds, while PBFT presents several peaks until saturation point with an average per transaction of 4.840 seconds.

6.7 Security analysis of SHERLOCK

By leveraging on particular network partition mechanisms, hash-based sharding helps a system to distribute users' requests and parallelise their processing. We apply this technique in SHERLOCK to improve blockchain scalability: in the face of growing demand and potential system saturation, we use sharding to add network nodes and create parallel consensus committees. Notwithstanding, this can come at a cost, the reduction of system security. Indeed, adding nodes and creating multiple consensus committees broadens the attack surface. As there are more nodes, an attacker has more access points at disposal. On the other hand, being the network partitioned, the honest majority is dispersed into individual committees. This significantly reduces the amount of honest consensus nodes in each committee, making it easier for an attacker to subvert one of them. Depending on how the committees are interconnected, this can lead, with a cascade effect, to break the security of the entire blockchain system. This is know as *single shard takeover attack*, where an attacker aims at controlling the majority of consensus nodes within a single committee to control the whole blockchain. Usually, this type of attack gains more ground in permissionless blockchains, where nodes have no restrictions to participate in the network. An adversary can then target one specific committee and enlist compromised nodes in its consensus protocol. Permissionless blockchain tackle this attack by introducing randomness into the committee formation and by reshuffling the nodes in the committees at the end of each consensus round. A pseudo-random function is used to randomly assign nodes to committees, whereby nodes with higher mining power or stakes are more likely to be selected.

In SHERLOCK we tackle the single shard takeover attack by combining three factors:

1. the *authentication* of consensus nodes;
2. the *trust assumptions* of systems based on permissioned blockchains;
3. the introduction of *fairness* into the two-layer ring-based architecture.

As first factor, SHERLOCK is a permissioned blockchain where all network nodes are authenticated. Before a node is assigned to a committee, its certificate is checked against a public repository, and once it participates in a committee, it is checked that the signature of its messages matches the public key of its certificate. These authentication checks enables SHERLOCK to form committees with identifiable nodes and to verify their actions within each single committee. As second factor, SHERLOCK relies on trust assumptions commonly used in permissioned settings, which regard both the participating organizations and the nodes engaged in the consensus protocol. As described in Section 6.1, these assumptions are:

- the 2/3 of participating organizations are honest;
- the consensus nodes are fairly distributed among organizations.

The former is indeed a necessary condition for the establishment of the system: a majority of organizations is required as a guarantee that the system achieves its intended goals, shared by its participants. The latter is related to the security of the consensus protocol: it prevents a single malicious organisation from having a number of consensus nodes above the security threshold, which typically is 1/3.

These two factors are sufficient to maintain the security of a traditional permissioned blockchain, as long as the number of malicious nodes remains less than 1/3. However, once the network is partitioned via sharding, the malicious nodes can end up in a single committee. Taking PBFT as example, in order to tolerate $f = 5$ byzantine nodes the system network needs to be composed by at least $n = 3f + 1 = 16$ nodes. Supposing that the n consensus nodes is then divided via sharding in 4 committees of the same size, each of them can then tolerate up to 1 byzantine node. However, it may happens that 2 out of 5 byzantine nodes are placed in a single committee, possibly causing its failure and a security breach. The transactions ordered by such committee are not guaranteed to be valid, and if appended into the ledger can bring inconsistency of the information. This situation may get worse if further 2 out of 5 byzantine nodes are placed in another committee, so that half of the committees possibly being compromised. In this scenario, the 50% transactions can be incorrect even though the system originally tolerates $f < n/3$.

Therefore, as third factor, SHERLOCK introduces fairness for the organizations participation in committees. Particularly, we first set the committee size to be equal to the number of organizations. Then, during the committees formation, we configure SHERLOCK so that each organization possesses one consensus node in every committee. This configuration applies to both proposed topologies, i.e without and with overlapping committees. The difference between the two lies only in the nodes positioning over the ring, which in the case of overlapping is also dictated by the organisations sorting. The addition of this third factor ensures that the initial security guarantees of the system are retained even after the division into committees. This because it prevents that a minority of malicious nodes (below the security threshold) in the entire system from becoming a majority in a single committee. With reference to the PBFT example mentioned before, SHERLOCK forms a committee with one node per organization. Hence, in a system composed by 4 organizations and $n = 16$ consensus nodes, we would have 4 committees of size $c = 4$. If one organization is dishonest, it can supply 4 byzantine nodes. Differently from previous example, in SHERLOCK at most one of them (i.e., 1-out-of-4) is placed in a single committee. This ensures that in each committee the majority of nodes belong to honest organisations,

and the number of byzantine nodes (i.e., 1 in this case) is below to the PBFT security threshold $c/3$, as long as the byzantine nodes in the system are $f < n/3$.

In conclusion, by combining authentication, trust assumptions and fairness, SHERLOCK preserves the security guarantees offered by a permissioned blockchain after partitioning the system into committees. Under the traditional trust assumptions of permissioned blockchains, SHERLOCK withstands against the single shard takeover attack as long as the $2/3$ of organizations are honest and the byzantine nodes in the system are $f < n/3$.

6.8 Related work

Sharding technology is currently identified, by both blockchain operators and researchers, has one of the most efficient methodologies to overcome the scalability issues of blockchain. Indeed, the co-founder of Ethereum, Vitalik Buterin, states that sharding is the future of the Ethereum platform, being the best approach to simultaneously fulfil the blockchain requirements of decentralisation, security and scalability (Buterin (2021)). As described by recent surveys (Yu et al. (2020); Wang et al. (2019); Hafid et al. (2020)), however, at the state-of-the-art sharding has been mainly applied in the context of permissionless blockchains.

In order to make a proper comparison with SHERLOCK, the following are some papers in which sharding is applied to permissioned blockchains. Dang et al. (2019) propose a approach based on trusted hardware to apply sharding to permissioned blockchain. Particularly, they exploit the Intel SGX to both (i) randomly form committees with random seeds created into the trusted environment, and (ii) to run inside it more secure PBFT-based committees. Marson et al. (2021) present Mitosis, a permissioned blockchain in which shard-chains are dynamically created when new nodes join the network. Differently from traditional sharding solutions where one chain is divided into multiple parallel committees, they recursively split an existing chain in two child. For security, they rely on trust assumptions, inherited from parent chain to child chains. Amiri et al. (2021) present SharPer, a permissioned blockchain that uses sharding to split network nodes into geographically clusters. They also apply sharding to split the ledger, such that each cluster only maintains its intra-committee transactions and the cross-committee transactions in which it is involved. To handle cross-committee transactions they represent the blockchain ledger as a directed acyclic graph. These mentioned papers (Dang et al. (2019); Marson et al. (2021); Amiri et al. (2021)), differently from SHERLOCK, employ a single-layer architecture for committees, which operate independently on the same level. Additionally, in paper Amiri et al. (2021) are used different version of the ledger, one per each committee. As a result, they have to handle transactions involving

Papers	committees formation	cross-committee	security
Dang et al. (2019)	probabilistic	yes	trust hardware
Marson et al. (2021)	deterministic	yes	trust assumptions
Amiri et al. (2021)	deterministic	yes	trust assumptions
Zhou et al. (2020)	probabilistic	no	committees change randomly in epochs
Du et al. (2020b)	probabilistic	no	committees change randomly in epochs
SHERLOCK	deterministic	no	trust assumptions and fairness

TABLE 6.1: Work applying sharding to permissioned blockchain

distinct committees, i.e. cross-committee transactions. Conversely, similar to SHERLOCK, papers (Zhou et al. (2020); Du et al. (2020b)) employ a two-layer architecture, in which there is a leading committee responsible for providing a total order to blocks generated by committees in the first layer. Indeed, Zhou et al. (2020) propose a sharding permissioned blockchain where consensus nodes are divided into committees, which order transactions in parallel, and there is a "boss committee" responsible for ordering their provisional blocks. However, as opposed to SHERLOCK, they assign nodes to committees via random numbers. Du et al. (2020b) propose a two-layer sharding permissioned blockchain, with committees on the second layer sending mini-blocks to a "high-level" committee that creates the final block. To form committees, similarly to Zhou et al. (2020), they rely on random numbers that they create by combining VRF and threshold secret sharing. For security, committees members are changed at the beginning of a new epoch.

Table 6.1 shows a comparison of the aforementioned papers with SHERLOCK. The first difference is that the papers of Dang et al. (2019), Zhou et al. (2020) and Du et al. (2020b) use randomness for committees formation. This because they do not rely on trust assumptions about the number of honest organizations, and how the network nodes are distributed among them. Hence, Dang et al. (2019) uses random number generated by a trusted hardware, Zhou et al. (2020) uses hashing functions with random inputs to map a node to a committee in the next epoch, Du et al. (2020b) uses VRF and threshold secret sharing to randomly assign nodes to next epoch committees. The randomness introduced by these papers is also reflected in the security of their systems, as shown in the last column of Table 6.1. The second difference is that papers of Dang et al. (2019), Marson et al. (2021), Amiri et al. (2021) employ cross-committee mechanisms to deal with transactions involved in different committees. This is due to the lack of a second layer, with a leading committee to assemble this type of transactions. The last difference regards the methods used by papers of Marson et al. (2021) and Amiri et al. (2021) for system security. They both rely only on trust assumptions that each committee has a majority of honest nodes, and this is

propagated when they create new ones. However, they are vulnerable to the single shard takeover attack.

To complete our literature review we reference some sharding solutions proposed for permissionless setting. They differ from those permissioned in the way they add nodes to various committees. Since permissioned blockchains operate in a relative trust environment, the nodes can be assigned to committees according to their organization membership and to the overall consortium policies. In permissionless blockchains, conversely, a node can participate to any committee without restrictions. This opens the doors to Sybil attack (Douceur (2002)), where an adversary creates multiple fake identities to take control of committees. Luu et al. (2016) propose Elastico, the first sharding protocol for dividing the network of a permissionless blockchain into multiple committees. Elastico uses PoW for committee formation and PBFT for intra-committee consensus. Specifically, at the beginning of a new epoch, Elastico uniformly re-allocates nodes into committees according to the PoW results of previous epoch. In Elastico the number of committees grows almost linearly with the size of the network, and it reaches a throughput of about 40 tx/s with 1600 nodes. Similarly to Elastico, Kokoris-Kogias et al. (2018) propose OmniLedger, a sharding blockchain that uses PoW for committee formation and ByzCoinX, a BFT-based protocol, for intra-committee consensus. OmniLedger relies on VRF to produce an unpredictable and unbiasable randomness for re-allocation and leader-election of each committee. In addition, OmniLedger uses Atomix, a byzantine shard atomic commit, to deal with cross-committee transactions. Compared to Elastico, OmniLedger shows a higher throughput, up to 500 tx/s with 1800 nodes. Following the release of Elastico and OmniLedger, Zamani et al. (2018) present RapidChain, a sharding blockchain likewise characterised by PoW and BFT. Specifically, Rapidchain comprises a "reference committee" that collects the PoW results of all nodes during an epoch, and according to them, assembles the committees for the next epoch. Rapidchain outperforms both Elastico and OmniLedger with a throughput up to 4220 tx/s when the network grows up to 1800 nodes. Inspired by OmniLedger and Rapidchain, in the whitepaper Team Harmony (2018) the authors propose a sharding blockchain platform named Harmony. Differently from OmniLedger and Rapidchain it uses PoS for committee formation, and similar to them a Fast-BFT for intra-committee consensus. Harmony contains a "beacon committee" that generates the randomness by combining VRF and Verifiable Delay Function (VDF). In order to become a committee member, a node in Harmony stakes a certain amount of tokens into the beacon committee: the more tokens are staked, the more voting shares one obtains in the BFT consensus.

6.9 Discussion

In this chapter we presented SHERLOCK, a permissioned blockchain that uses the sharding technique to enhance system scalability. Specifically, we first designed a two-layer ring-based architecture for nodes engaged in the consensus protocol, composed by two concentric rings. Then, we used the hash-based sharding to split the consensus nodes into committees and arrange them over the architecture rings. We formed multiple committees on the outer ring, and a single leading committee on the inner ring. The committee arrangement on the outer ring is offered in SHERLOCK in two versions: without and with overlapping committees. Regardless the version, the incoming set of transactions to order, i.e. the workload, are distributed via sharding across the outer ring committees, which concurrently and in isolation process them. The inner ring committee lastly order the blocks of the outer ring and provides a final block to be appended to the ledger. In the face of growing workload, SHERLOCK scales out by adding nodes and thus committees to its network, improving scalability.

We analysed and compared the two versions qualitatively, by calculating their communication complexity and the effort a node exerts to handle a transaction in both approaches. Given the same number n of consensus nodes in the system, the result was that the two versions have the same communication cost, linear to n, and that a node exerts the same effort in both versions. That is, they are equally efficient. The overlapping version forms more committees in the outer ring than the no-overlapping, leading to a greater workload distribution, but conversely in the no-overlapping a node participates in lesser outer ring committees than in the overlapping, exerting less effort. A key difference between the two lies in the number of nodes required to create a new committee in the outer ring. The no-overlapping version needs a number of nodes equal to the system committee size c, while the overlapping version just needs one node. Therefore, a system administrator should choose the version of SHERLOCK according to the number of available nodes and the resources available to a node. If there are few nodes available, the version with overlapping is preferable. If the nodes assigned to the outer ring have few resources available, the version without overlapping is preferable.

In order to assess the advantages of our proposed solution, we compared SHERLOCK with PBFT in two steps. Firstly, we compared their communication complexity. Secondly, we implemented them and evaluated their performance under the same network configuration and workload. Given the same number n of consensus nodes, the result of the first comparison was that SHERLOCK has a communication cost linear to n, while PBFT has a communication cost quadratic to n. This first finding demonstrates that SHERLOCK has a better communication complexity than PBFT, requiring fewer message exchanges, which means less time to agree on transactions order and reach consensus. More importantly, SHERLOCK splits n in multiple

committees of size c, where c is generally much smaller than n, and then distributes the workload over the outer ring committees. Hence, scaling out to deal with a growing workload means in SHERLOCK adding a new committee, with a cost of $O(c^2)$, while in PBFT it means adding nodes to the same committee, with a cost of $O(n^2)$. Furthermore, by adding a new committee, SHERLOCK reduces the workload on the other committees, enabling them to process more transactions and move away from the saturation point. Conversely, the new nodes added in PBFT handle the same workload as the others, and the cost of communication will deteriorate the overall performance, bringing the saturation point closer.

This becomes evident in our second comparison where we measured the throughput and latency of SHERLOCK and PBFT with a workload of 1000 txn/s for 5 minutes. Both SHERLOCK and PBFT were configured with $n = 12$, and the committee size of SHERLOCK was set to $c = 4$. The evaluation shows that PBFT reaches the saturation point at around 75 seconds and stops processing transactions. Scaling out the PBFT in this context worsens its performance, causing the saturation point to occur before 75 seconds. Paradoxically, scaling in the PBFT would help improve performance, but only up to a certain extent since the saturation point occurs shortly after the evaluation begins. On the other side, the evaluation shows that SHERLOCK processes all transactions submitted with very good performance, i.e. an average throughput of 990.1 txn/s and an average latency per transaction of 1.937 seconds. Therefore, these comparisons demonstrate that SHERLOCK outperforms PBFT, and a SHERLOCK-based permissioned blockchain can scale out by spreading the workload with a small computational cost of $O(c^2)$.

Finally, as regards security, we proved that SHERLOCK withstands the main attack carried-out on sharding-based blockchain networks, i.e. the single shard takeover attack. In SHERLOCK, we tackled this attack by combining nodes authentication and trust assumptions of permissioned blockchains with organisations' fairness in committee participation. Each organisation possesses one consensus node in every committee, and SHERLOCK is secure as long as the $2/3$ of organizations are honest and the byzantine nodes in the system are $f < n/3$.

As a future work, we will investigate the possibility of adding a third ring to the SHERLOCK architecture in the event of saturation of the inner ring.

Chapter 7

Conclusions

In this thesis, we studied the properties of blockchain technology and we proposed methodologies to enhance the privacy and scalability of permissioned blockchains. In the first part of the thesis we analysed the blockchain security, by defining a taxonomy of the security properties relevant for a blockchain-based system and for the consensus protocol in place. We then examined five prominent blockchain platforms, i.e. Bitcoin, Ethereum 2.0, Algorand, Ethereum-private and Hyperledger Fabric, and their respective consensus protocols, PoW, Casper PoS, Pure PoS, PoA and PBFT. We evaluated their security against the defined taxonomy. Furthermore, we listed smart contract issues and we evaluated whether blockchains are vulnerable and their native resistance/mitigations. This study was pivotal in understanding what the differences are between the two types of blockchain, permissionless and permissioned, and identifying why they respectively lack data privacy and have performance issues. As a result, we decided to focus on the permissioned blockchains, because unlike permissionless they offer node authentication and exhibit better performance. These two factors are crucial for any privacy-preserving solution. Moreover, this study showed that permissioned blockchains have scalability problems as the number of nodes in the network increases, causing performance to deteriorate. Hence, we also decided to investigate solutions that solve scalability in permissioned blockchains.

The second part of the thesis focuses on permissioned blockchains, examining the two aforementioned open challenges: the lack of data privacy and the lower scalability of network nodes. As regards privacy, the only feature offered by permissioned blockchains is the enforcement of data access control rules. However, high privilege system users are authorised to read confidential data, and since data are stored in the ledger as plaintexts, all blockchain nodes can read its content. Although encrypting data solves the privacy problem, on the other hand it opens up another challenge: once data are in ciphertexts, the smart contracts can no longer execute functions on them. As regards scalability, permissioned blockchains rely on voting-based consensus protocols, which require a high number of messages exchanged between

nodes to agree on transactions ordering. Adding nodes in this protocol, in an attempt to cope with an increasing workload, results in more message exchanges, a longer time to reach consensus, and thus a degradation of performance.

We addressed the privacy issue in two related stages. In the first stage, we proposed a permissioned blockchain combined with the HE model, capable of simultaneously encrypting data and executing functions on their ciphertexts through smart contracts. Particularly, we designed an architecture in which clients are endowed with HE keys to autonomously encrypt and decrypt their data, while blockchain nodes can perform HE-based computations with smart contracts. We applied this architecture into a Smart Grid system, to provide both integrity and privacy to users' energy data, and to carry out decentralised privacy-preserving energy billing and trading. We implemented this HE-equipped blockchain by integrating the fully homomorphic CKKS scheme with Hyperledger Fabric. The results of the experimental evaluation show that the overhead introduced by this FHE scheme in the blockchain is particularly small, demonstrating the effectiveness and feasibility of this solution. Besides Smart Grid systems, this HE-equipped blockchain can be applied in any distributed setting composed of multiple parties who wish to keep their data private and benefit from decentralised application. Indeed, we provided a generalisation of this architecture for IoT-based systems.

In the second stage, we extended the previous combination between blockchain and HE by presenting PANTHER, a permissioned blockchain integrated with MKHE. Besides offering the same functionalities as the previous solution, PANTHER also enables smart contracts to perform computations on data of different users, encrypted with distinct keys. The results of these MKHE-based computations are then decrypted in PANTHER through MPC decryption protocols among involved clients. We detailed the PANTHER's architecture and the protocols run by clients and blockchain nodes, and we implemented it by integrating MK-CKKS, a multi-key variant of CKKS, with Hyperledger Fabric. Since CKKS works with real and complex numbers, this prototype implementation of PANTHER is particularly well-suited for machine learning and data analytics use cases, where numerical precision is crucial. The experimental results showed that the overhead introduced by MK-CKKS in the blockchain is small for HE-based addition (29.67ms), and higher but feasible for HE-based multiplication (2.73s), demonstrating that MKHE does not significantly affect the blockchain performance. On the other hand, due to the execution of MPC protocols for decryption, MK-CKKS introduces an average overhead of 1.975 in clients. Indeed, we estimated that the client initiator of a MPC decryption takes on average 3.03s to complete the protocol, while the other clients involved take on average 1.62s. Unfortunately, the MPC protocol cannot be avoided in a system where users have personal encryption keys, regardless of whether they are generated via MKHE or even THE. Notwithstanding, its associated overhead is fair price to pay to

obtain the benefits offered by PANTHER, namely the high level of data privacy and user independence in performing computations on data encrypted by others. Furthermore, we proved that PANTHER provides the security properties of data integrity and availability, which along privacy, are preserved against a byzantine adversary.

To overcome the scalability issue, we presented SHERLOCK, a permissioned blockchain based on sharding technology. We designed in SHERLOCK a two-layer ring-based architecture for consensus nodes. We used hash-based sharding to split nodes into committees and arrange them over the two concentric rings of the architecture. As the volume of transactions to process grows, SHERLOCK scales out by adding committees, thereby increasing the consensus parallelisation, reducing the workload per node and boosting overall performance. We provided two versions of SHERLOCK: without and with overlapping committees in the outer ring. We analysed qualitatively the two versions, without and with overlapping, and we demonstrated that are equally efficient. Furthermore, we compared SHERLOCK with PBFT to assess its advantages. In particular, we first compared their communication complexity, then we implemented them and measured their performance under the same network configuration and workload. The first comparison demonstrated that SHERLOCK has a communication cost linear to the number of consensus nodes, while PBFT has a quadratic communication cost. In the second comparison, we evaluated SHERLOCK and PBFT with a workload of 1000 txn/s for 5 minutes. The experimental results demonstrated that SHERLOCK outperforms PBFT, because PBFT becomes saturated and stops processing transactions after 75 seconds, while SHERLOCK processes all submitted transactions with a very good average throughput and latency. The flipside of the coin is that SHERLOCK needs a minimum number of nodes greater than PBFT to be realised. Indeed, the minimum configuration for SHERLOCK is two outer and one inner committees, and each committe must have a minimum size of 4 to tolerate a byzantine node. Hence, systems intending to implement it must have sufficient resources. Regarding SHERLOCK security instead, we proved that it withstands against the single shard takeover attack.

PANTHER and SHERLOCK achieve the ultimate goal of this thesis, namely providing two effective and viable solutions to be potentially applied on the same permissioned blockchain, simultaneously solving privacy and scalability issues.

As future work, we will explore two ways to optimise PANTHER and SHERLOCK solutions. To enhance PANTHER security, we will investigate mechanisms to dynamically involve more honest clients in a MPC decryption protocol, so as to raise byzantine tolerance above 1/3. To enhance SHERLOCK resilience, we will investigate the addition of a third middle ring in the architecture, where other committees can be arranged, so as to lighten the load on the inner ring.

Appendix A

Internet of Things

The *Internet-of-Things (IoT)* is one of the most prominent paradigm of computer science field, and as claimed by Whitmore et al. (2015) it plays a remarkable role in all aspects of our daily lives. The IoT concept encompasses Internet-connected objects able to collect informations from surrounding environment and exchange data with each other without human intervention. The ”things” in this context refer to physical devices (e.g. smart meters, smart bulbs, smart locks, IP cameras and more), equipped with sensors and processing power that enable them to monitor and create informations about machines or human behaviours, analyse such informations and undertake automatic action. Additionally, these IoT devices, communicating over Internet, allow to control objects remotely like turn on and off heating or air conditioning. This eases some daily activities and helps save resources in terms of money and time, increasing the way people interact with surrounding environment.

The IoT paradigm leads to a new era of Internet where the physical world and the digital world are fully integrated. Indeed, in 2020 IoT is expected to connect more than 30 billions of devices (Lund et al. (2014)). It finds application in many fields, starting from private sector (e.g. domotics, e-health, assisted driving) to business sector (e.g. industries in manufacturing, transportation and logistics, production processes), with the aim of improving efficiency, accuracy, and as a consequence, gaining economic benefits. Ideally, the IoT architecture, as shown in Figure A.1, can be classified into four layers, namely application, middleware, network, and perception, in which the data flow is bidirectional:

- *Application Layer*: It is on top of the architecture and exports the functionalities offered by middleware layer to final users. It can be structured in several ways based on the applications it has to provide. Applications including, but not limited to, smart grid, smart home, smart city, healthcare system, and intelligent transportation protocols, constitute this layer. An application protocol (e.g. web

or mobile service protocol) is distributed over multiple end systems, with which application instances belonging to different end systems can exchange information packets.

- *Middleware Layer*: It is an intermediate layer between the physical layer and the final end-user application layer, capable of abstracting devices functionalities and communications protocols. It hides the details of different technologies adopted by the lower layers, simplifying their integration for the development of IoT applications. The middleware layer provides a common set of services, characterised by the *Service Oriented Architecture* (SOA) approach. Generally the SOA-based middleware encompasses three others layers, namely service composition, service management and object abstraction. The *Service Composition* is on top of the middleware layer and provides composition functionalities to services offered by the service management layer, allowing to build various applications depending on the context. The *Service Management* provides the catalogue of services associated to each IoT device including dynamic discovery, monitoring, computation, information storage, configuration, and management of Quality of Service (QoS) and policies. This layer might enable the remote deployment of new services during run-time, in order to satisfy application needs. The *Object Abstraction* exposes interfaces that wrap and standardise the different functions offered by the heterogeneous set of IoT devices, thus hiding their inherent specifications and business logic.

- *Network Layer*: It provides the required communication features to the system. Specifically, it guarantees a reliable information transition protocol for the operations (i) between devices, (ii) within the network, (iii) between the network and the upper layers. In the IoT, devices are connected prevalently using wireless technology, populating the so-called Wireless Sensor Network (WSN). Most of the WSNs communicate using the communication standard proof 802.15.4 (Gutierrez et al. (2004)) which defines the operation of low-rate wireless personal area networks (LR-WPANs) and is lighter than other standards, e.g. Bluetooth (802.15.1 (2006) - proof 802.15.1) and Wireless (802.11 (2016) - proof 802.11). Starting from this, other solutions appeared for WSN. ZigBee (ZigBee Alliance (2012)) is the most famous one because it provides low power consumption, low data rate, low cost, and high throughput. However, bridging between ZigBee and non-ZigBee networks is challenging and it requires a complex application layer gateway. For this reason and because modern IoT devices should be connected with the Internet, objects in the IoT will need to use a more flexible solution. 6LoWPAN (Mulligan (2007)) is a new communication protocol who delivers the IPv6 version of the Internet Protocol (IP) over WPANs. It offers interoperability between wireless 802.15.4 devices as well as with devices on any other IP network link (e.g. Wi-Fi) with a simple bridge device.

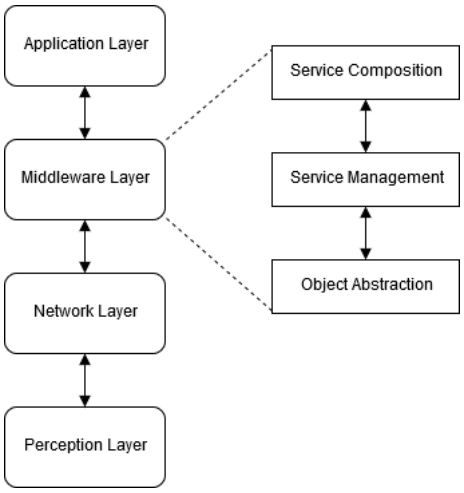


FIGURE A.1: IoT architecture layers.

- *Perception Layer*: It is composed by end point devices, sensors, controllers and actuators which characterise the IoT system, i.e. things. IoT devices are interconnected among a large-scale distributed network, they collect information from the environment and communicate to the upper layers for analytics operations. Perception layer include various technologies, such as the Radio Frequency Identification (RFID) to identification and location tags, and the WSNs, which are considered the IoT major building blocks. Originally networks of devices were able to communicate with other services through gateways. Nowadays devices can also be connected to the Internet with IP address, using the connectivity protocols offered by the network layer.

Appendix B

Smart Grid

Because of its inherent skills and features, IoT principle has been deployed in several key technological areas, as reported in Atzori et al. (2010), including energy sector. Indeed, as described in Collier (2016), current electric grid is no longer viable due to the customers need to reduce energy consumption and to avail of renewable energy sources (RES), e.g. wind and solar power. This causes the shift towards the introduction of IoT devices within electric grid, establishing the so-called *Smart Grid*. It leverages on smart meter devices, which are able to record the consumptions and report the power-related informations (e.g. voltage) to electricity supplier for monitoring and billing. Smart meters are the key driver of this digital revolution, in which energy supply is aligned with actual customers demand, accurately balancing energy load. In addition, smart meters empowers consumers to adjust their energy usage and reduce their costs. Hence, on one hand they make the grid more efficient, reliable and sustainable. From the other, they open up new opportunities in energy market, introducing decentralised and distributed systems, known as *micro-grids*, where energy can be produced and exchanged autonomously (Farhangi (2010)). An example of a micro-grid environment is depicted in Figure B.1, where communities (e.g. homes, buildings or campuses) within a predefined perimeter can operate isolated as an electric island or connected with the traditional main grid [1].

As defined in Ton and Smith (2012) by U.S. Department of Energy Microgrid Exchange Group, the micro-grid is a distributed system composed by distributed energy resources (DER) and RES, able to be self-governed and energy self-supplied. The distinguishing properties of a micro-grid network, as described in Rahimi and Ipakchi (2012), are:

- It dispatches energy derived by DER and RES inside the micro-grid, potentially bringing its demand from the grid to zero;

[1] The image was taken from Microgrid Institute.

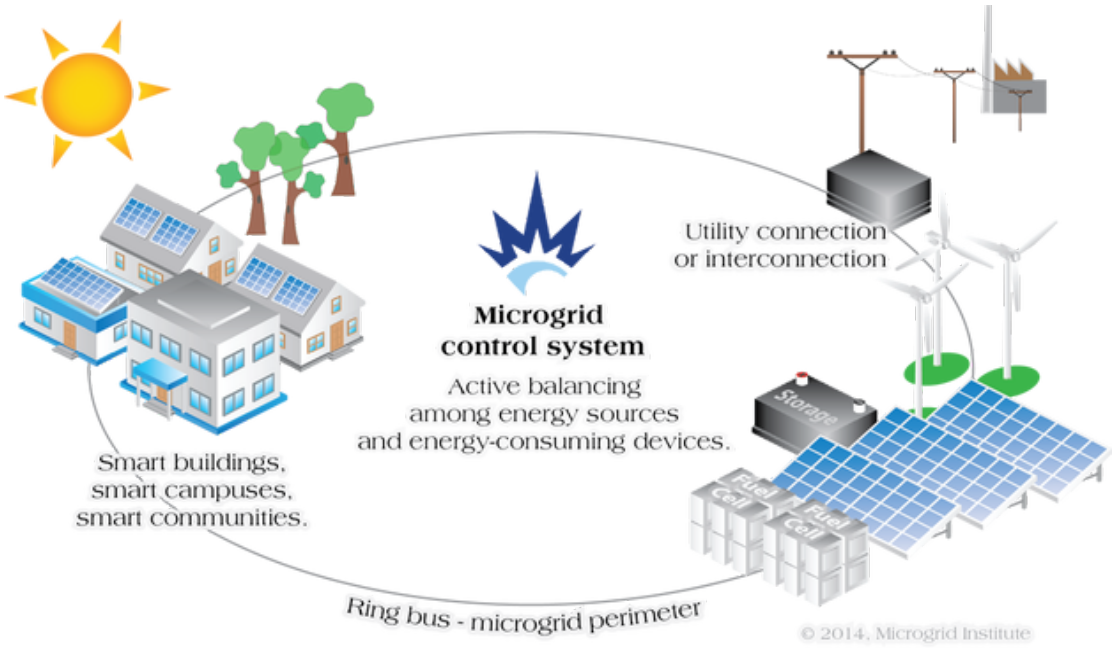


FIGURE B.1: Micro-grid environment.

- It exports exceeding generated energy outside to others micro-grids or the main grid, creating a bidirectional flow of power (and related informations) between power providers and end-use consumers.

Therefore, the micro-grids advent leads to a new free market economy where consumers become themselves producers, i.e. *prosumers*, diversifying wholesale domain with retail sales. Prosumers can trade energy bilaterally both inside and outside the boundaries of a micro-grid, adapting to the price of power providers or through an auction-based approach (Rahimi et al. (2016)). This emerging market needs regulations and new distributed control methods, which fall under the concept of *transactive energy* (TE) (Kok and Widergren (2016); Liu et al. (2017); Sahin and Shereck (2014)). As defined in Melton (2013), TE is a set of market-based techniques exploited to dynamically balance the energy load and control energy exchange across a distributed set of prosumers in the electrical infrastructure, clearly stating:

- Who are the transacting parties, i.e. prosumers involved in the transaction;
- What informations are exchanged between prosumers to create and execute transaction(s);
- What are the rules governing transactions;
- What is the mechanism for reaching an agreement on the transaction price.

The TE techniques use these operational informations, along with economic signals, to (i) enable an optimal integration of DER and RES with the main grid and (ii) ensure

stability of the entire electrical system, avoiding erroneous or malicious transactions which can create a gap between demand and supply.

References

Lattigo v2.3.0. Online: https://github.com/ldsec/lattigo, October 2021. EPFL-LDS.

Std 802.11. *IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*. IEEE, 2016.

Std 802.15.1. *IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*. IEEE, 2006.

Nurzhan Zhumabekuly Aitzhan and Davor Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Transactions on Dependable and Secure Computing*, 15(5): 840–852, 2016.

Fadele Ayotunde Alaba, Mazliza Othman, Ibrahim Abaker Targio Hashem, and Faiz Alotaibi. Internet of things security: A survey. *Journal of Network and Computer Applications*, 88:10–28, 2017.

Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

Algorand Foundation. Algorand, 2017.

Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *SIGMOD '21: International Conference on Management of Data*, pages 76–88. ACM, 2021.

Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating

system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

Apache Software Foundation. Apache kafka, 2011a.

Apache Software Foundation. Apache storm, 2011b.

G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7237 LNCS:483–501, 2012.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST - 6th International Conference, Proceedings*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

Algirdas Avizienis, Vytautas U, Jean-claude Laprie, and Brian Randell. Fundamental concepts of dependability. *Department of Computing Science Technical Report Series*, 2001.

M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 1–10, 1988.

Fabrice Benhamouda, Shai Halevi, and Tzipora Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. *IBM Journal of Research and Development*, 2019.

J. Bernal Bernabe, J.L. Canovas, J.L. Hernandez-Ramos, R. Torres Moreno, and A. Skarmeta. Privacy-preserving solutions for blockchain: Review and challenges. *IEEE Access*, 7:164908–164940, 2019.

Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349. ACM, 2012.

BitFury Group. Incentive mechanisms for securing the bitcoin blockchain, 2015.

BitFury Group and Jeff Garzik. Public versus private blockchains part 1: Permissioned blockchains. *White Paper*, 2015.

G.R. Blakley. Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge, MARK 1979*, pages 313–317, 1979.

P. Bogetoft, D.L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5628 LNCS:325–343, 2009.

Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.

Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012 - Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 97–106, 2011.

Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 2016.

Vitalik Buterin. Why sharding is great: demystifying the technical properties, 2021.

Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, 2000.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186. USENIX Association, 1999.

CertiK. The state of defi security, 2021.

F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 2020.

Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

M. Clear and C. McGoldrick. Multi-identity and multi-key leveled fhe from learning with errors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9216:630–656, 2015.

Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

Steven E Collier. The emerging enernet: Convergence of the smart grid with the internet of things. *IEEE Industry Applications Magazine*, 23(2):12–16, 2016.

ConsenSys. Mythril, 2020.

R. Cramer, I. Damgård, and J.B. Nielsen. Multiparty computation from threshold homomorphic encryption. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2045: 280–300, 2001.

Ronald Cramer, Ivan Bjerre Damgrd, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4586 LNCS:416–430, 2007.

Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, pages 123–140. ACM, 2019.

Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. In *Proceedings of the Second Italian Conference on Cyber Security*, 2018.

Stefano De Angelis, Gilberto Zanfino, Leonardo Aniello, Federico Lombardi, and Vladimiro Sassone. Blockchain and cybersecurity: a taxonomic approach. *EU Blockchain Observatory and Forum*, 2019.

Stefano De Angelis, Gilberto Zanfino, Leonardo Aniello, Federico Lombardi, and Vladimiro Sassone. Evaluating blockchain systems: A comprehensive study of security and dependability attributes. In *Proceedings of the 4th Workshop on Distributed Ledger Technology co-located with the Italian Conference on Cybersecurity (ITASEC)*, volume 3166 of *CEUR Workshop Proceedings*, pages 18–32, 2022.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

John R. Douceur. The sybil attack. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

M. Du, Q. Chen, J. Xiao, H. Yang, and X. Ma. Supply chain finance innovation using blockchain. *IEEE Transactions on Engineering Management*, 67(4):1045–1058, 2020a.

Mingxiao Du, Qijun Chen, and Xiaofeng Ma. MBFT: A new consensus algorithm for consortium blockchain. *IEEE Access*, 8:87665–87675, 2020b.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

Ethereum. Geth: Ethereum implementation in go, 2013.

Ethereum. Parity ethereum: A blockchain infrastructure for decentralised web, 2018a.

Ethereum. Solidity, 2018b.

Ethereum. Proof-of-stake, 2022a.

Ethereum. Vision of ethereum2, 2022b.

Hassan Farhangi. The path of the smart grid. *IEEE power and energy magazine*, 8(1): 18–28, 2010.

Q. Feng, D. He, S. Zeadally, M.K. Khan, and N. Kumar. A survey on privacy protection in blockchain system. *Journal of Network and Computer Applications*, 126:45–58, 2019.

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

Edoardo Gaetani, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Blockchain-based database to ensure data integrity in cloud computing environments. In *ITA-SEC*, volume 1816. CEUR-WS.org, 2017.

Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009a.

C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8042 LNCS(PART 1):75–92, 2013.

Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009b.

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

Go-Ethereum Docs. Ethereum private networks, 2022.

Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM, 1987.

S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 365–377. Association for Computing Machinery, 1982.

Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

Shafi Goldwasser, Silvio Micali, and Charles Rackoff. Knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

Google. Golang programming language, 2009.

Jose A Gutierrez, Edgar H Callaway, and Raymond L Barrett. *Low-rate wireless personal area networks: enabling wireless sensors with IEEE 802.15. 4*. IEEE Standards Association, 2004.

Abdelatif Hafid, Abdelhakim Senhaji Hafid, and Mustapha Samih. Scaling blockchains: A comprehensive survey. *IEEE Access*, 8:125244–125262, 2020.

Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–5. IEEE, 2017.

Shai Halevi. Homomorphic encryption. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 219–276. Springer International Publishing, 2017.

Ryan Henry, Amir Herzberg, and Aniket Kate. Blockchain access privacy: Challenges and directions. *IEEE Security Privacy*, 16(4):38–45, 2018.

Maurice Herlihy and Mark Moir. Enhancing accountability and trust in distributed ledgers. *CoRR*, 2016.

Ghassan O. Karame, Elli Androulaki, Marc Roeschlin, Arthur Gervais, and Srdjan Čapkun. Misbehavior in bitcoin: A study of double-spending and accountability. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):1–32, 2015.

C. Killer, B. Rodrigues, E.J. Scheid, M. Franco, M. Eck, N. Zaugg, A. Scheitlin, and B. Stiller. Provotum: A blockchain-based and end-to-end verifiable remote electronic voting system. In *Proceedings - IEEE Conference on Local Computer Networks, LCN*, volume November, pages 172–183. IEEE Computer Society, 2020.

Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition, 2022.

Koen Kok and Steve Widergren. A society of devices: Integrating intelligent distributed resources with transactive energy. *IEEE Power and Energy Magazine*, 14 (3):34–45, 2016.

Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018*, pages 583–598. IEEE Computer Society, 2018.

Ioannis Kounelis, Gary Steri, Raimondo Giuliani, Dimitrios Geneiatakis, Ricardo Neisse, and Igor Nai-Fovino. Fostering consumers' energy market through smart contracts. In *2017 International Conference in Energy and Sustainability in Small Developing Economies (ES2DE)*, pages 1–6. IEEE, 2017.

Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

Leslie Lamport. Paxos made simple, fast, and byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems*, 2002.

Aron Laszka, Abhishek Dubey, Michael Walker, and Doug Schmidt. Providing privacy, safety, and security in iot-based transactive energy systems using distributed ledgers. In *Proceedings of the Seventh International Conference on the Internet of Things*, page 13. ACM, 2017.

Zhihu Li, Haiqing Xu, Feng Zhai, Bing Zhao, Meng Xu, and Zhenwei Guo. A privacy-preserving, two-party, secure computation mechanism for consensus-based peer-to-peer energy trading in the smart grid. *Sensors*, 22(22):9020, 2022.

Zhaoxi Liu, Qiuwei Wu, Shaojun Huang, and Haoran Zhao. Transactive energy: A review of state of the art and implementation. In *2017 IEEE Manchester PowerTech*, pages 1–6. IEEE, 2017.

A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 1219–1234, 2012.

Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *IACR Cryptology ePrint Archive, Report 2013/094*, 2013.

Denise Lund, Carrie MacGillivray, Vernon Turner, and Mario Morales. Worldwide and regional internet of things (iot) 2014–2020 forecast: A virtuous circle of proven value and demand. *International Data Corporation (IDC), Tech. Rep*, 1, 2014.

Fengji Luo, Zhao Yang Dong, Gaoqi Liang, Junichi Murata, and Zhao Xu. A distributed electricity trading system in active distribution networks based on multi-agent coalition and blockchain. *IEEE Transactions on Power Systems*, 2018.

Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

S. Ma, Y. Deng, D. He, J. Zhang, and X. Xie. An efficient nizk scheme for privacy-preserving transactions over account-model blockchain. *IEEE Transactions on Dependable and Secure Computing*, 18(2):641–651, 2021.

Katiuscia Mannaro, Andrea Pinna, and Michele Marchesi. Crypto-trading: Blockchain-oriented energy market. In *2017 AEIT International Annual Conference*, pages 1–5. IEEE, 2017.

Giorgia Azzurra Marson, Sébastien Andreina, Lorenzo Alluminio, Konstantin Munichev, and Ghassan Karame. Mitosis: Practically scaling permissioned blockchains. In *ACSAC '21: Annual Computer Security Applications Conference*, pages 773–783. ACM, 2021.

Ronald B Melton. Gridwise transactive energy framework (draft version). Technical report, Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2013.

Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th IIWBAS*, 2018.

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

Silvio Micali. Algorand co-chains, 2020.

Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

T. Mitani and A. Otsuka. Traceability in permissioned blockchain. *IEEE Access*, 8: 21573–21588, 2020.

Malte Möser. Anonymity of bitcoin transactions an analysis of mixing services, 2013.

Mozilla. Rabbitmq, 2010.

P. Mukherjee and D. Wichs. Two round multiparty computation via multi-key fhe. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9666:735–763, 2016.

Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 2008.

Sana Noor, Wentao Yang, Miao Guo, Koen H van Dam, and Xiaonan Wang. Energy demand side management within micro-grid networks enhanced by blockchain. *Applied energy*, 228:1385–1398, 2018.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

OpenZeppelin. Solidity safemath library, 2017.

P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1592:223–238, 1999.

Parity. Aura - authority round consensus protocol, 2017.

C. Peikert and S. Shiehian. Multi-key fhe from lwe, revisited. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9986 LNCS:217–238, 2016.

Manisa Pipattanasomporn, Murat Kuzlu, and Saifur Rahman. A blockchain-based platform for exchange of solar energy: Laboratory-scale implementation. In *2018 International Conference and Utility Exhibition on Green Energy for Sustainable Development (ICUE)*, pages 1–9. IEEE, 2018.

D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 367–377. ACM, 2004.

Farraokh Rahimi, Ali Ipakchi, and Fred Fletcher. The changing electrical landscape: end-to-end power system operation under the transactive energy paradigm. *IEEE Power and Energy Magazine*, 14(3):52–62, 2016.

Farrokh A Rahimi and Ali Ipakchi. Transactive energy techniques: closing the gap between wholesale and retail markets. *The Electricity Journal*, 25(8):29–35, 2012.

Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM, 2005.

Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *CoRR*, abs/1812.05934, 2018.

Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.

Moein Sabounchi and Jin Wei. Towards resilient networked microgrids: Blockchain-enabled peer-to-peer electricity trading mechanism. In *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–5. IEEE, 2017.

Tugcan Sahin and Daniel Shereck. Renewable energy sources in a transactive energy market. In *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pages 202–208. IEEE, 2014.

Noama Fatima Samreen and Manar H. Alalfi. A survey of security vulnerabilities in ethereum smart contracts. *CoRR*, abs/2105.06974, 2021.

A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

W. She, Z.-H. Gu, X.-K. Lyu, Q. Liu, Z. Tian, and W. Liu. Homomorphic consortium blockchain for smart home system sensitive data privacy preserving. *IEEE Access*, 7: 62058–62070, 2019.

Bano Shehar, Sonnino Alberto, Al-Bassam Mustafa, Azouvi Sarah, McCorry Patrick, Meiklejohn Sarah, and Danezis George. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019.

M. Shen, X. Tang, L. Zhu, X. Du, and M. Guizani. Privacy-preserving support vector machine training over blockchain-based encrypted iot data in smart cities. *IEEE Internet of Things Journal*, 6(5):7702–7712, 2019.

Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.

Parminder Singh, Mehedi Masud, M. Shamim Hossain, and Avinash Kaur. Blockchain and homomorphic encryption-based privacy-preserving data aggregation model in smart grid. *Comput. Electr. Eng.*, 93:107209, 2021.

Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160. ACM, 2001.

Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

Tim Swanson. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. *Report, available online, Apr*, 2015.

Péter Szilágyi. Clique - ethereum proof-of-authority consensus protocol, 2017.

Team Harmony. Harmony: Technical whitepaper, 2018.

Dan T Ton and Merrill A Smith. The us department of energy's microgrid initiative. *The Electricity Journal*, 25(8):84–94, 2012.

Carmela Troncoso, Marios Isaakidis, George Danezis, and Harry Halpin. Systematizing decentralization and privacy: Lessons from 15 years of research and deployments. *Proceedings on Privacy Enhancing Technologies*, 2017:404–426, 2017.

M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6110 LNCS: 24–43, 2010.

Shivangi Vashi, Jyotsnamayee Ram, Janit Modi, Saurav Verma, and Chetana Prakash. Internet of things (iot): A vision, architectural elements, and security issues. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 492–496. IEEE, 2017.

Heribert Vollmer. *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2015.

Marko Vukolic. Eventually returning to strong consistency. *IEEE Data Eng. Bull.*, 39(1): 39–44, 2016.

Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 41–61. ACM, 2019.

Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–73. IEEE, 2017.

Andrew Whitmore, Anurag Agarwal, and Li Da Xu. The internet of things—a survey of topics and trends. *Information Systems Frontiers*, 17(2):261–274, 2015.

Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. In *IEEE International Conference on Software Architecture, ICSA*, pages 243–252. IEEE Computer Society, 2017.

X. Yang, X. Yi, S. Nepal, A. Kelarev, and F. Han. Blockchain voting: Publicly verifiable online voting protocol without trusted tallying authorities. *Future Generation Computer Systems*, 112:859–874, 2020.

Yuchen Yang, Longfei Wu, Guisheng Yin, Lijie Li, and Hongbin Zhao. A survey on security and privacy issues in internet-of-things. *IEEE Internet of Things Journal*, 4(5): 1250–1258, 2017.

Andrew C. Yao. Protocols for secure computations. In *Annual Symposium on Foundations of Computer Science - Proceedings*, pages 160–164, 1982.

B. Yu, J.K. Liu, A. Sakzad, S. Nepal, R. Steinfeld, P. Rimba, and M.H. Au. Platform-independent secure blockchain-based voting system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11060 LNCS:369–386, 2018.

Guangsheng Yu, Xu Wang, Kan Yu, Wei Ni, J. Andrew Zhang, and Ren Ping Liu. Survey: Sharding in blockchains. *IEEE Access*, 8:14155–14181, 2020.

Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 931–948. ACM, 2018.

Gilberto Zanfino. Data in support of the thesis: Enhancing privacy and scalability of permissioned blockchain. University of Southampton, `https://doi.org/10.5258/SOTON/D3096`, 2024. Dataset.

R. Zhang, R. Xue, and L. Liu. Security and privacy on blockchain. *ACM Computing Surveys*, 52(3), 2019.

Zhixuan Zhou, Zhijie Qiu, Qiang Yu, and Hong Chen. A dynamic sharding protocol design for consortium blockchains. In *2020 IEEE International Conference on Big Data (IEEE BigData 2020)*, pages 2590–2595. IEEE, 2020.

ZigBee Alliance. Zigbee specification, 2012.