Analysis of the Ethereum state

Carlos Pérez Jiménez

Màster Universitari en Seguretat de les Tecnologies de la Informació i de les Comunicacions

TFM Seguretat en xarxes i aplicacions distribuïdes ·2017-18 Sem.1
Supervisor: Guillermo Navarro Arribas

Abstract

Blockchains have gathered phenomenal interest due to their potential to disrupt established business models. Ethereum expanded the properties of blockchains to become a platform for the development and implementation of decentralised Applications. We survey the platform and its associated ecosystem and study the data structures that support the protocol, in particular the global state of the network. We also introduce a tool to extract the state from the native storage of a node of the network with the aim to facilitate the statistical study of the dataset.

Resum

Les Cadenes de Blocs han generat un gran interès gràcies a la seva capacitat per alterar els models de negoci establerts. Ethereum va ampliar les seves propietats per convertir-se en una plataforma per al desenvolupament i implementació d'aplicacions descentralitzades. Analitzem la plataforma i el seu ecosistema associat i estudiem les estructures de dades que donen suport al protocol, en particular l'estat global de la xarxa. També introduïm una eina per extreure l'estat de l'emmagatzematge natiu d'un node de la xarxa amb l'objectiu de facilitar l'estudi estadístic del conjunt de dades.

Contents

1. Introduction

The advent of blockchains and cryptocurrencies sparked a wave of interest that goes beyond the technical achievements of the existing platforms and set great expectations in terms of a paradigm shift and disruption in many areas such as finance and e-commerce. This interest has generated substantial coverage on mainstream media, as well as on newly created specialised digital media such as Coindesk[1]. Undoubtedly, a substantial proportion of this excitement is linked to the financial gains that can be obtained through mining, Initial Currency Offerings (ICOs) and the performance of investment on cryptocurrencies. Not to be overlooked is the appropriation from certain political quarters, notably libertarians of the ideas of disintermediation and decentralisation, especially towards governments, central banks and large financial institutions or big corporations in general. Finally, we should consider the hype generated in the start-up sector, mostly, but not restricted to, Fintech, with regard to the change of paradigm represented by the emergence of distributed applications, the so-called Đapps and the lowering of entry barriers on those established industries that they represent. And yet, big corporations have turned their attention to the technology and have announced initiatives such as Quorum[2], centralised cloud services host Blockchain as a Service (BaaS)[3] and institutions such as the IMF and central banks are toying with some of the ideas behind cryptocurrencies [28]. Moreover, a certain backlash against speculation on cryptocurrencies [37] and the security of the investments [45, 14] is starting to emerge and to this day, there is no mass adoption of cryptocurrencies as means of payment.

Leaving socio-economic considerations aside, the blockchain and Bitcoin were conceived, as per the Bitcoin white paper [38], as a solution to the double-spending problem of digital currencies. Furthermore, they simultaneously introduced the idea of distributed trust through consensus protocols, dispensing with the need of a trusted third party (TTP) and established the idea of a distributed ledger of transactions that are nearly impossible to tamper with and are auditable by any willing observer. Subsequently, the Ethereum white paper [6] extended the idea beyond cryptocurrencies with the introduction of a Turing-complete programming

1 https://www.coindesk.com
2 https://www.jpmorgan.com/global/Quorum
3 https://azure.microsoft.com/en-gb/solutions/blockchain and https://cloud.oracle.com/en_US/blockchain

language and storage to represent any state which is respectively executed and stored by every node in the network, adding to the blockchain the ability to implement any distributed application in addition to the existing means of payment.

Ethereum is conceived as an open-source project with the backing of a non-profit organisation, The Ethereum Foundation[1]. Despite the success of the associated cryptocurrency, it could be argued that as a project of such ambition and complexity, it is still in its infancy. After all, the white paper was only published in 2013 and the first incarnation of the network, Frontier, dates from 2015. A roadmap setting the evolution of the platform was laid out in the early days of the project and it sets a phased approach with several major releases or versions of the network: after the Frontier and Homestead releases, Metropolis-Byzantium, the current version, went live on the 27th October 2017 and the plan continues with the Metropolis-Constatinoble and Serenity releases. Consequently, there is intense research in many areas associated with the protocol, e.g., the consensus mechanism migration from Proof of Work to Proof of Stake, privacy provision in the form of zero-knowledge proofs (zkSnark) or ring signatures or scalability improvement through the use of sharding.

Other than the protocol itself, there is a vast associated ecosystem in permanent state of flux: client nodes are implemented in an ever expanding range of languages; essential components for the wider adoption of the platform such as Mist, the wallet/browser of Đapps, are in beta status; languages for the development of smart contracts proliferate and are subsequently abandoned; a myriad of parallel infrastructure related projects such as Swarm for decentralised storage, Whisper for decentralised messaging, Plasma for child blockchains, etc. are being developed and in parallel; there is a frantic activity in the development and funding of distributed applications that would make use of this infrastructure. And yet, there is a conspicuous lack of "killer apps" that makes the platform mainstream and the biggest impact so far [29] is an application to generate digital pets…

Possibly as a consequence of this frenetic activity, the literature and documentation are often obsolete, poorly formalised as well as distributed across many platforms and formats: wikis, blogs, discussion forums, etc. hampering the understanding of the platform by the neophyte.

Finally, the security of the platform, or rather the applications in the ecosystem [42], is in need of reinforcement, let's not forget that a notorious attack, the heist of The DAO, was so severe that it forced an unscheduled release of a

[1] https://www.ethereum.org/foundation

new version of the platform to reverse the effects of the attack [31] and triggered a split on the network between Ethereum and Ethereum Classic when a fraction of miners refused to follow suit.

1.1. Statement of work

This final master's degree project has a dual goal. Firstly, we will study the Ethereum platform and produce a survey of the state of the art, how it works and how it stores data internally, as well as the ecosystem of applications, languages, libraries, etc., associated with it. Secondly, we will put knowledge into practice and develop a tool to query the state of the network, current and historical, and produce statistics.

In terms of the survey, given on the one hand the infancy of the project, the breadth of the ecosystem and its constant evolution and on the other, the lack of coherent documentation, we will aim to convey a cohesive introduction of the field, centred in the aspects less likely to become rapidly outdated but nevertheless cover all the relevant ones. Simultaneously, we will keep focus on the ultimate goal of producing a new tool to query the state.

Likewise, the software components of the Ethereum ecosystem proliferate and become obsolete, not unlike projects in neighbouring open-source area, web development, where frameworks also come and go. Notably, pyethapp, once a popular client for the network favoured by some members of the core development teams as a means of quick experimentation, is no longer able to connect to the mainnet, that is, the production network. We would therefore need to survey and carefully select the client node, the libraries and other tools before proceeding to the development of our tool.

There already exist ways of querying the network stats, for instance, the client nodes expose a standard API which offers a limited number of queries about the state of the network through its interactive Javascript console [21] or a JSON RPC channel, such as the number of accounts, etc. Web portals such as Etherscan[1] offer APIs to obtain statistics, however they are restricted to the current state and are subject to a fair usage policy. We would like overcome those limitations and introduce the possibility of historical queries e.g. the number of accounts at a given date. On the other hand, it is worth noting that it is not our intention to provide a real-time statistics platform for Ethereum nor transfer the information contained in the blockchain to an external database, which would be beyond the scope of a work of this nature in terms of complexity and for which some implementations have already been proposed [34], but to read the data directly from the native storage and process it. We would therefore use existing libraries, such as pyethereum, to

[1] https://etherscan.io

query directly the internal database of the client of our choice, generally LevelDB, so that we have freedom to design the set of queries to implement in our tool.

1.2. Methodology

As per the previous section, this project is a hybrid between a survey of the domain and the application of the acquired knowledge in the form of the design and implementation of a tool to query the state of the Ethereum network, where each part would weigh approximately half of the effort. Not being either an experimental type of work, for which we would define how to obtain, process and interpret the results, nor a complex software development project, for which we would apply specific methodologies such as agile development, etc., the discussion or implementation of a formal methodology is of little relevance. Nevertheless, it is worth mentioning the reasoning behind the objectives and how they are conceived in order to solve the problem exposed in the previous section of this document and how they translate into a near sequential list of tasks.

Our first objective, the study of the Ethereum protocol and ecosystem would serve as an introduction to the field and will inform us of the necessary concepts to analyse the current implementations and the existing tooling. In turn, this analysis will help us choose the development environment suited for the development to follow. We will pursue our survey with a detailed study of the Ethereum protocol data structures. This will provide us with the necessary information to evaluate what sort of queries are feasible within the scope of this project: current state, historical queries, etc., and to design the library. The list of tasks and the planning for the project are listed in the Appendix A.

2. Blockchains: from Bitcoin to Ethereum

We will introduce in this section the concept of digital currency and how it led to the advent of Bitcoin and the blockchain as a solution to the problems that prevented the success of previous attempts at creating such schemes. We will continue describing the evolutions of blockchain technology sparked by Bitcoin and follow with a brief definition of the properties and concepts often used when discussing blockchains.

2.1. Digital currency and Bitcoin

We can loosely define a digital currency as a means of payment between two parties that aims at replicating some properties of physical cash such as transferability, non counterfeitability, anonymity and divisibility. Research on digital currencies date back to 1983 with the study by David Chaum of an untraceable mean of payment [13]. One of the main problems of digital currency schemes is double-spending, i.e., the need to implement a mechanism to prevent the spending of the same digital coin more than once given the triviality of duplicating a digital token. Chaum's scheme achieved those properties with the introduction of a new cryptographic primitive, the blind signature, and solved the problem of double-spending, albeit by resorting to a TTP, the institution issuing the currency, that could verify if double spending was attempted by keeping a database, or in financial terms, a ledger, of spent coins and a ledger of user account balances.

Satoshi Nakamoto proposed in 2008 [38] the Bitcoin protocol as a solution to the double-spending problem that disposes of the need of a TTP by defining a set of rules implemented by the nodes of a peer-to-peer network (P2P) which collectively maintain a ledger of the transactions carried with the protocol unit or token of value: bitcoin. Any node in the network can generate and broadcast to the rest of the nodes a transaction, a message signifying the transfer of an amount of bitcoins between N input and M output addresses, the latter known as unspent transaction outputs (UXTO). An address is a string of bytes generated from the public key of an asymmetric encryption algorithm and it is the private/public key pair that allows the authentication of the transactions, i.e., that the sender has indeed the control over the funds. Each input refers to a previous output, thus generating a verifiable chain of transfers of value tokens. Note that there are no explicitly defined coins such a string of bytes stored in a central database as the Chaum protocol defines: the amount of bitcoin in possession of a user of the network is the sum of amounts held in the UXTOs under his control. A ledger of transactions, i.e., a time-ordered list of transactions, is generated as they are processed from the

unprocessed transaction pool and grouped together in a block, which in turn are chained to each other as they are created, forming a linked list, known as the blockchain, a copy of which is stored locally by each node, thus constituting a distributed ledger. The chronological ordering of the transactions and blocks is the basis of double-spending avoidance as the transaction verification process checks that each input in a transaction is linked to a previous output and that the sender of the transaction effectively controls the input accounts by verifying a cryptographic signature.

A distributed ledger maintained by a P2P network needs a mechanism to achieve consensus of what the status of ledger is, as transactions are not necessarily broadcast to all the nodes of the network synchronously and therefore not processed in the same chronological order, leading to inconsistencies between the copies of the ledger maintained by each node. On the one hand, the overall linked list of blocks mutates to a tree as different nodes might link to the same parent block the blocks that each created separately and on the other hand, it opens the door to double-spending as two transactions with the same input but different outputs are sent by a malicious actor to two different nodes which would be able to verify them separately and include them in their copies of the ledger. The obvious mechanism to achieve consensus is by majority voting, however in an open network this is subject to Sybil attacks by malicious actors, where the attacker subverts the majority vote by forging several identities. This an example of a well known problem in distributed computing: the Byzantine Generals Problem [33], where a network of peers that do not trust each other and some of which could act maliciously, need to achieve consensus on the state of a shared resource. Bitcoin solves this problem with a consensus mechanism known as proof of work, a cryptographic puzzle which requires a substantial amount of computational work to solve and thus renders Sybil attacks inviable and by enforcing amongst the peers the rule that the branch of the tree that represents the longest path is the one to be chosen when extending the chain with new blocks.

In addition to that, Bitcoin incentivises good behaviour through economic rewards: the node that solves the puzzle and therefore contributes to the security of the network receives a reward for the creation of the block and the fees implied in the transactions included in the block, i.e., the difference between the input and output amounts.

The principles behind the design rationale and implementation of Bitcoins and blockchains in general are abundantly described in wikis and blogs over the internet and in several scientific papers, [48, 5]. We will briefly discuss the most relevant concepts below after describing some of the evolutions sparked by Bitcoin in the following section.

2.2. Evolutions

As Bitcoin gathered attention, it sparked the creation of other cryptocurrencies or alt-coins. Soon, the principles of disintermediation and distributed trust associated with blockchains began to be generalised beyond the sphere of means of payment and the concept of coloured coins appeared to represent other tokens of value. Thereafter, a further generalisation of those concepts to be applied to the development of generic distributed or rather decentralised applications, resulted in the specification of the Ethereum protocol. Finally, major corporations paid attention to the technology and its potential effects in their industries and introduced the concept of permissioned blockchains.

Alt-coins

The advent of Bitcoin sparked the creation of many other alt-coins, often as a fork of the existing code of Bitcoin, each with a particular focus or objective, such as Litecoin[1], which aims at faster processing of transactions and block creation, Ripple[2], focused on inter-banking payments and Zcash[3], designed for privacy.

Colored Coins

The Colored Coins protocol [15] was introduced in 2013 with the aim of attaching metadata to bitcoins using the limited scripting functionality already available in the Bitcoin protocol so that bitcoins could be tied up to real-world assets or services based on the promise of redemption of the coloured coin for the asset or service by the issuer of the coin. The protocol uses the returned code by the execution of Bitcoin scripts to point to metadata stored in BitTorrent.

Ethereum

Also in 2013, Vitalik Buterin proposed a brand new blockchain, to be developed from scratch rather than forking the existing Bitcoin code, with the aim to serve as a platform for the development of generic decentralised applications unconstrained by the perceived single use case of Bitcoin, i.e., a payment network. Two key elements differentiate Ethereum from Bitcoin: the introduction of a Turing complete execution environment on the client nodes of the network and the implementation of an explicit current global state of the network model instead of the chain of UXTOs used by Bitcoin.

---

1 https://litecoin.com

2 https://ripple.com

3 https://z.cash

Enterprise blockchains

As companies started to realise the potential disruption and benefits that blockchains could bring to their core businesses, the concept of private or permissioned blockchains started to emerge, addressing particular concerns such as privacy, efficiency of consensus protocols, scalability, etc., that affect public chains such as Bitcoin and Ethereum.

A fully private blockchain would be an implementation controlled by a single organisation in order to leverage the properties of replicated state machine and cryptographic authentication. A blockchain is not necessarily the most optimal way to achieve those goals for a single organisation with full control of the software, although it could be argued that it could provide a cheap and easy way of connecting different systems, often legacy, used by different departments within a big organisation. Nevertheless, it is not a popular set up and most of the effort is focused on permissioned blockchains.

Permissioned or consortia blockchains, consist of a P2P network whose peers are generally companies with a commercial relationship and other stakeholders such as regulators. The consensus is provided by a pre-selected set of nodes and other nodes might have limited privileges, such as read-only functions. As the validators are known, the consensus algorithm does not need to be as stringent as those of an open network [12]. The most popular use cases of these blockchains are settlement platforms between organisations with the aim to allow cheaper inter-institutional transactions and easier scalability and supply chain management, again with a focus in interoperability and traceability. As the network is under the control of a few players, rules can be changed, transactions reverted, etc., without the risk of forks (see below) present on the open networks. On the other hand, the software platforms for these blockchains can still leverage the improvements achieved by the open source public chains on which they might be based. The most known initiatives around permissioned blockchains are Corda[1], Hyperledger[2] and within the context of Ethereum, the Ethereum Enterprise Alliance[3], all participated by many companies in a varied set of industries.

2.3. Blockchains

The technology behind Bitcoin, Ethereum and other blockchains or Distributed Ledger Technologies (DLT) as it is also known, has several implications and properties that go beyond the use case of a payment system.

---

1 https://www.corda.net

2 https://www.hyperledger.org

3 https://entethalliance.org

The implementation of a P2P network of equal nodes following a common protocol which allow for the processing of transactions without the recourse to a TTP has profound implications for the existing models of e-commerce. Alternative business models or use cases are possible which do not require an entity whose sole purpose is to broker between parties while charging for the service, as trust between parties is provided by computer programs whose execution is delegated to all nodes of the network. It is said then that blockchains provide a form of distributed or decentralised trust.

The basis of this distribution of trust is the consensus protocol implemented by the blockchain to maintain a common state across the network. Bitcoin uses PoW and the longest chain rule to enforce a common state, however it is not the only possible choice. Practical Byzantine Fault Tolerance (PBFT) and Proof of Stake (PoS) amongst others are the most commonly considered alternatives.

Nevertheless, PoW continues to be the most popular choice, at least amongst public chains. It does attract, however, a considerable amount of criticism, mostly aimed at the energy consumption (or waste according to the critics) required for its fulfilment. In its most common implementations, the calculations that constitute the proof of work executed by the peer, the search for a random number which added to the block of transactions results to a digest (cryptographic hash) of the whole data structure that fulfils a certain property, only contributes to the security of the network, instead of potentially more constructive uses of the computing power. Two terms are commonly used within the context of PoW: (a) difficulty, a network wide parameter which drives the chance of solving the PoW puzzle and whose value is set dynamically or at least adjusted periodically in order to maintain a steady interval between the creation of blocks and (b) the network hashrate, an estimate of the number of hashes calculated by the entire network per second given the current difficulty and interval between blocks.

One of the consequences of mining, i.e., the process of constructing a valid block within the constrains of PoW, is that there is no formal transaction finalisation, i.e., there is no point in time within the exchange protocol when the transaction can be considered irrevocably settled. It is always possible for a parallel chain to become longer than the chain where a particular transaction was processed and therefore revoke that transaction. Under normal operating conditions, this would be an improbable event after a certain number of blocks have been added to the chain after the block on which the transaction was included. However, if an attacker were able to control more than 50% of the hashrate of the network, it could easily build a longer parallel chain and consequently double spending becomes feasible.

This process of disintermediation by the empowerment of nodes in a P2P network provides, by its very own nature, resistance to single point of failure, i.e., a failure of the TTP that would cause a centralised network such as AirBnB or eBay to fail in its entirety, as well as resistance to censorship, the possibility for a TTP to deny service to a party due to a conflict of interests. As a corollary of censorship resistance, the distributed ledger becomes (near) immutable as no central authority is able to single-handedly modify the contents or the history of the blockchain. Only a network-wide consensus would allow for the rewrite of history through a change in the protocol.

Such changes in the protocol are known as forks, not to be confused with the temporary forks in the blockchain, the different paths in the blocktree described above, formed as the consensus is dynamically established. Two types of protocol changes or forks are usually considered: (a) soft forks, where the change allow for the coexistence of nodes running different versions of the protocol with only potential disadvantages for the minority of nodes in older versions but with the interoperability remaining nearly intact and (b) hard forks, where the changes are so profound that two groups of nodes using pre- and post-fork versions effectively constitute two separate P2P networks and their mining efforts build two separate blockchains with a common ancestor. Generally, a soft fork will correspond to changes in the protocol that result in more restrictive rules and therefore nodes that do not upgrade within a network where the majority of nodes have will still accept upgraded nodes´ blocks but they might have theirs rejected by the network. Conversely, hard forks will be the result of less restrictive rules and consequently, a non upgraded node will reject all the majority of the peers blocks.

Finally, another property of blockchains worth mentioning is auditability: as transactions and the blockchain itself, with all its history since the first block, known as the genesis block, is available to any node upon joining the network, the blockchain constitutes a public ledger and thus auditable.

3. The Ethereum platform

Ethereum is often described as a protocol, however it really represents a P2P network, a development platform and an associated ecosystem. In this section we will provide a high-level view of the protocol, the ecosystem built around it and how the project is governed as an open-source project.

3.1. High level overview

Ethereum is often described as the world's computer, sometimes more specifically as a transactional state machine. These two sound-bites refer to the key deviations of Ethereum from the Bitcoin blockchain model, the Turing complete execution environment, the Ethereum Virtual Machine (EVM) and the maintenance of a network-wide state instead of a chain of historical transactions.

As other blockchains, the protocol defines its own token of value, the ether (represented by the ISO4217-like code ETH and uppercase Greek letter Xi, Ξ, as symbol), with the following fractional denominations: Wei (10^{-18} Ξ), Szabo (10^{-12} Ξ) and Finney (10^{-15} Ξ). Its role is to act as a means of exchange similar to other cryptocurrencies and most importantly, as the mechanism to pay for the computational costs related to the processing of the transactions. Note that Wei is the sub-denomination used internally by the protocol.

The protocol defines two types of accounts, (a) externally owned accounts, controlled by an entity off-chain, usually a person, whose main purpose is to hold funds in ether and interact with the other type of account, (b) the contract, that holds business logic as an executable program written in EVM opcodes which is executed by each node of the network as transactions or rather message calls are sent to the contract account. In order to avoid the Halting Problem inherent to Turing machines, which would lead to denial of service (DoS) to the network, the concept of gas is introduced: every computational step is assigned a certain cost in terms of gas units, every transaction sent to the network states the upper limit of gas to be used during its execution and the price in Weis that it is willing to pay per gas unit. The execution of the validation process, i.e., the execution of the contract code or the transfer of funds, is metered and aborted if it exceeds the limit of gas stated.

The current consensus mechanism in place is PoW, implemented through a memory intensive algorithm, Ethash, that favours CPU/GPU execution over purpose-built equipment based in ASICs, in order to avoid the dominance of few individual miners or pool of miners. However, at the time of writing, a

quick look at the network statistics show that more that 50% of the hashrate is in the hand of three pools[1].

3.2. Ethereum as an open source project/community

Ethereum is an open source project which has gathered a wide community of developers. However there is a key group of developers, whose periodic meetings are recorded and available through different channels, who steer the development of the core platform. In addition to that, the Ethereum Foundation was set up in 2014 as a non-profit organisation with the mission of promoting the development of new technologies and applications in the fields of decentralised software architectures and particularly the Ethereum protocol and associated technologies and applications. The Foundation is in charge of managing the funds raised during the ether pre-sale of July-September 2014 before the general launch of the network. Other for-profit companies are big players in the field, often founded by ex-members of the Ethereum foundation, such as Consensys and Parity Technologies.

The community of developers as well as users and other interested parties rally around several collaboration or social media platforms, notably the Ethereum blog, reddit, the "Issues" sections of the relevant GitHub repositories, gitter and others[2]. There are many MeetUp groups devoted to Ethereum and developer conferences, out of which the most prominent is DEVCON, organised by the Ethereum Foundation. The multitude of platforms combined with an apparent lack of leadership in terms of writing a canonical documentation leads to one of the issues most agreed and commented about the project: the dispersion of the information, the lack of updates and the consequent difficulty for a starter to delve into the subject. Besides, despite all the hype surrounding the fields of blockchain, "cryptoeconomics" and Ethereum in particular, the academic interest on the subject is just starting and the amount of academic literature is relatively scarce.

The governance of the evolution of the platform is achieved through Ethereum Implementation Proposals (EIP), a design document providing a technical specification and rationale for a new feature [20]. Ethereum Request for Comments (ERC) are a type of EIP describing application level standards and conventions. EIPs would normally be proposed to the community in the usual forums to gauge the public interest before they are formalised and submitted to a committee for approval to discuss as a draft proposal. Subsequently the proposal can be accepted, rejected, withdrawn, deferred to

1 https://www.etherchain.org/charts/topMiners

2 https://github.com/ethereum, https://blog.ethereum.org, https://ethresear.ch, https://gitter.im/ethereum/home, https://ethereum.stackexchange.com, https://www.reddit.com/r/ethereum.

or superseded. After acceptance, as implementations of the feature are completed, the EIP is considered final.

Whereas governance amongst developers follows a fairly regulated and formalised process, the governance of the wider community including users and miners is rather controversial [18, 51, 8].

3.3. History of the development up to the present

From its inception [9], a roadmap for the development of the platform was defined and included the following releases:

- Olympic – a testing network,
- Frontier – a beta release of the mainnet,
- Homestead – the first stable release of the mainnet,
- Metropolis – to be delivered in two stages: Byzantium, the current live mainnet, and Constantinople.
- Serenity – future release which will include Proof of Stake.

In addition to these versions, there have been several other unplanned hard forks: the DAO, the EIP-150 and the Spurious Dragon forks, to address security incidents.

3.4. The ecosystem

The Ethereum protocol and its subprotocols are either implemented or used by a range of applications that can be addressed collectively as the Ethereum ecosystem which spans from the software used by the nodes that constitute the P2P network, to the applications and entities that interface the network to the rest of the internet or the real world.

Client node and wallets

The client node of the P2P network is the application that implements the protocol, validates the blockchain and processes the transactions sent over the network. There is no canonical client in Ethereum and several implementations on a variety of languages exist: go-ethereum developed by the Ethereum Foundation and Parity developed by Parity Technologies are the most popular. Nodes can operate in different modes to optimise the synchronisation time with the network and the storage. A client operating in full mode and standard synchronisation will download all the blockchain since the genesis block and execute all transactions, thus generating the full history of the state, this is known as an archive node. If operating in full mode but with fast synchronisation, it will also download the blockchain but it will not execute the transactions to generate the current and historical states, so it downloads a snapshot of the current state and from that point it carries on

operating as a full node, i.e., generating historical state [46]. A client operating in light mode, a feature under development, only downloads the block headers and queries the network ad hoc for other data such as account states [22]. Although mining is possible with generic clients, specialised nodes such as Ethminer or Claymore would be necessary for profitable mining in conjunction with specialised and dedicated hardware with a set up of several GPUs.

The most visible application in the ecosystem for the end user is the wallet, generally a web application whose purpose is to manage the accounts owned by the user and interact with the network, i.e., send transactions, create contracts, etc. Given the emphasis of Ethereum on decentralised applications rather than payment network, Mist is the tool developed by the Ethereum Foundation to interact with them, Ethereum Wallet is an implementation of Mist that interacts with a single distributed application: the Wallet. Note that wallets tend to be bundled with an implementation of the client node, the download of Mist/Ethereum Wallet or the Parity wallets installs a local node, go-ethereum for Mist and the namesake node for Parity.

Exchanges

From the perspective of the end user, the next most visible component of the ecosystem are the exchanges, companies that facilitate the conversion of cryptocurrencies amongst them, i.e., from Ethereum to Bitcoin, or to fiat currencies i.e. real world currencies such as euros, dollars, etc. These companies operate a centralised market and offer their services through "traditional" websites.

The web3 stack

Ethereum is conceived as a platform for the development of decentralised applications, i.e., the backend of such applications is implemented by a P2P network whose nodes execute the business logic concurrently rather than by the servers of a single company. In addition to the business logic, implemented through smart contracts following the Ethereum protocol, storage and inter-application messaging capabilities, also following a decentralised paradigm, are needed to build fully functioning applications. These three components are referred to as the web3 stack and it comprises the Ethereum blockchain and the Swarm and Whisper P2P subnetworks as the storage and messaging layers respectively. In addition to those, web3.js, a Javascript extension, is provided to allow client-side applications developed with the html/css/Javascript paradigm to interact with the Ethereum/Swarm/Whisper nodes. Furthermore, application logic that does not require consensus directly, such as account management but complex enough not to

be executed on the browser can run alongside the node in the backend and interface with the node [27].

Swarm

Swarm is the P2P network for the distribution of data amongst the applications. Note that the client nodes, e.g. go-ethereum, implement the protocol alongside Ethereum. Although the blockchain is capable of storing data and any data subject to consensus, notably the state of a contract, must indeed be stored on-chain, this bears an impractical cost in gas and latency of delivery for large datasets such as digital media which otherwise are not subject to consensus. Distributed storage systems in the form of P2P networks already exist, e.g., BitTorrent, IPFS, however the nodes of those do not have any incentive to provide long term storage of content, nor any compensation for their contribution to a healthy network in the form of bandwidth and disk-space which leads to seeder/leecher situations. Swarm implements a payment channel between nodes as they deliver content to each other [47], the micro payments for the delivery of the content are netted however if a certain threshold is crossed a payment is triggered. Swarm also implements a messaging service between applications akin to a postal service suitable where anonymity is not required.

Whisper

Whisper is yet another P2P subnetwork whose implementation is bundled with the nodes. It is designed as a means for distributed applications to communicate with each other when consensus is not needed i.e. the message is not to change directly and explicitly the state of the blockchain and therefore save on the gas costs that a call to a contract would otherwise carry and foremost, when privacy is a concern as otherwise, in the form of a message call, it would be seen in the clear by all nodes of the network. Even if the content within the message was to be encrypted, there would be metadata leakage as the patterns of communication between nodes could be analysed. The routing of the messages is multi- or broadcast and (may be) kept private using probabilistic routing. The messages are assigned “Topics”, a cryptographic ID which is used by the nodes to filter (Bloom filtering) what they are interested in. The protocol is asynchronous, i.e., the sender and the recipient do not need to be online simultaneously. Finally, in order to avoid the spamming of the network, the sender is requested to complete a proof of work in order to send a message.

Ethereum Name Service

A name service, Ethereum Name Service (ENS) akin to Internet's DNS is implemented through blockchain contracts [32] to facilitate the location of contracts and accounts.

Đapps and DAO

The terminology around decentralised applications, often referred to as Đapps[1], and other decentralised constructs, such as a decentralised autonomous organisation (DAO) is far from formalised [7]. For the purpose of our survey of the ecosystem around Ethereum, we would describe a decentralised application as an application built with the components provided with the aforementioned web3 stack in order to make use of the concepts of decentralisation, disintermediation, censorship resistance, etc., characteristic of blockchains to provide a complex business use case. However, given the current state of development of Swarm and Whisper, they tend to be hybrids where the backend logic other than the business logic implemented on the blockchain is hosted centrally on the web servers of the company behind the Đapp.

The next step in extending the ideas behind blockchains is the creation of an organisation, such as a company, that is in itself decentralised and autonomous. The management, its day-to-day operations, etc., are all coded as smart contracts and executed in the blockchain autonomously from human intervention, although interaction is obviously allowed or even expected. "The DAO" was such an organisation whose purpose was to act as an incubator for Ethereum related start ups: investors would vote in what projects to fund according to the size of their investment in "the DAO" and the code behind it would do the rest. As per the criticism of smart contracts as a mechanism to implement law ("code is law") with the unambiguous and implacable logic of a computer program [39], implementing an organisation so removed from the real world is not exempt from problems and still subject to social consensus (as opposed to consensus algorithms), the actual law and the unpredictability of human behaviour, as the history of "the DAO" itself shows. The exploitation of a bug in its software led to the theft of its funds which triggered a hard fork of the chain in order to stop the heist and a split of the community between those who followed the hard fork and those who did not by principle and carried on as Ethereum Classic.

[1] A common pun within the Ethereum community as Đ is the uppercase Old English Eth letter, in other blockchain contexts, they are simply referred to as Dapps.

Tokens

Digital tokens, i.e., a digital representation of an asset (e.g., an alternative currency, shares in a company, physical objects, such as cars, …) are generally created as a means to claim or transfer ownership and are implemented in Ethereum as an on-chain contract, which on a basic level, implements a mapping between token accounts and its balances. In order to standardise tokens and allow their interaction with other contracts an interface was defined by ERC20 [49] which defines two roles, the owner and the spender and provides a set of operations to transfer the ownership of tokens according to the token amount balance, set allowances to spend, etc.

Tokens have been used as way to fund projects by a vehicle known as ICO (Initial Coin Offerings). Tokens are issued to the public in exchange for ether or another cryptocurrency and they are akin to shares in a company or a promise to receive a future service. However this issuance is not subject to regulation from the financial authorities as it is the case with shares and IPOs. ICOs have raised many criticisms and concerns. They are regularly described as the "wild west" although some people might consider them a fair way to fund open-source projects. The attention attracted is so phenomenal that several websites[1] are dedicated to report on them. However there are already the first signs that the financial regulators are considering regulating the process [2].

Oracles

Oracles, or Data Feeds, are contracts that feed data requests to other contracts. Many potential use cases for contracts would require the input of real world, off-chain, events or state, relaying this data to a contract poses the problem of deterministic execution: all the nodes of the chain need to receive the same data, even if requested at different times so that all can reach a consensus state. It also poses the problem of trustworthiness of the data feed itself and re-centralisation, i.e., the contracts will depend on a trusted third party to supply the data. Companies like Oraclize[2] set themselves as an honest broker between the contract an established and trusted data source and provide proof of authenticity of the fetched data and normalises it for the consumer contract. An alternative solution is to implement a contract as an N-of-M multisig, where N parties will produce the requested piece of data and only the M parties that reach a consensus within whatever criteria defined,

[1] https://www.icoalert.com

[2] http://www.oraclize.it

e.g., within the standard deviation, are rewarded for the data. This is the Augur[1] and Gnosis[2] model for prediction data.

So far it is evident that the implementation is highly dependent on the business case and that the terminology is vague and there is a conspicuous lack of standardisation, for instance as opposed to tokens.

[1] https://augur.net
[2] https://gnosis.pm

4. The Ethereum protocol and architecture

There is no canonical conceptual model to describe the architecture of the Ethereum platform. We have chosen to model the architecture in four layers: (a) the network layer, which describes the aspects related to the P2P network that constitutes the platform, such as network formation or supported application protocols, (b) the data layer, which covers the data types and structures used by the protocol and how they are stored locally by the node, (c) the consensus layer, where we briefly discussed the process to achieve consensus on the state amongst the nodes and finally, (d) the application layer, which describes the EVM and smart contracts. We follow with a brief survey of the future evolutions of the protocol and the security issues affecting the platform.

4.1. The network layer

Ethereum nodes form a P2P network which constitutes Ethereum itself. The network protocols are implemented by the devp2p libraries which are included in any Ethereum client. The devp2p libraries are basically divided in three layers: (a) the transport protocol (RLPx), which defines the format of the data as it is transmitted over TCP and provides symmetrical encryption (AES256) for the data and authentication over elliptic curve digital signatures (ECDSA – secp256k1), (b) the application layer where the protocols of the different applications that the network supports are defined, that is, ***eth*** for Ethereum (in the blockchain sense) nodes and ***les*** for light clients, ***bzz*** for Swarm data distribution, ***pss*** for Swarm postal service, and ***shh*** for Whisper, and (c) the node discovery protocol, which uses a distributed hash table (DHT) to find other nodes in the network, tries to establish a connection using RLPx and exchanges the capabilities, i.e. which applications and versions the node supports, and the blockchain network id that the node is running.

This network layer has so far been the most stable part of the protocol with very few changes, possibly due to the difficulty of implementation as consensus from all participants is needed for any change of the core network protocols and usually any modification is reserved for the hard forks. However on the application layer of devp2p, changes are easier to implement as they are viewed as extras for which a hard fork is not needed, hence, the on-going development of application protocols the Swarm and Whisper projects.

4.2. The data layer

This section describes the serialisation format, Recursive Length Prefix (RLP), used to stored data locally or transmit it over the network, the abstract data

types, the Patricia Merkel Trees (PMT or trie) and the Bloom filter, used by the Ethereum protocol and the data structures, implemented by those abstract types, that an Ethereum node maintains: the account, the account storage, the transaction, the receipt and the block.

Recursive Length Prefix serialisation format

Recursive Length Prefix (RLP) is a serialisation format for lists of items where items are either a string of bytes or nested lists of items. It allows the encoding of an empty list and a null string. As opposed to other serialisation formats, it does not define any data type such as floats, integers, etc. and the data is treated as string bytes. It is left to the application to apply any formatting to it.

The first byte of the serialised content indicates the type of data serialised:

- If the first byte is lower that 0x80, it indicates that the serialised content is a single element, a byte, whose value is lower than 0x80 and it has been serialised as that first and only byte, e.g., value 0x44 is encoded as 0x44.
- If the first byte value is between 0x80 and 0xB7, the serialised content is a single item of a length between 0 and 55 bytes and its encoded as itself following the first byte whose value is 0x80 plus the length of the string, e.g. string 0xFF FF is encoded as 0x82 FF FF. Note that an empty string is then coded as 0x80.
- If the first byte is between 0xB8 and 0xBF, the serialised content is a single item of a length between 56 and 14EiB (2^{64} bytes) and it is encoded as itself following a first byte whose value is 0xB7 plus the length in bytes of the big-endian integer representing the length of the item and said integer, e.g. a string of 2048 0xFF bytes is encoded as 0xB9 08 00 FF … FF.
- If the first byte is between 0xC0 and 0xF7, the serialised content represents a list whose encoded length in bytes is between 0 and 55 bytes and it is encoded as the concatenation of the RLP encoding of its elements following the first byte whose value is 0xC0 plus the length of the encoded list, e.g. list (0x44, 0xFFFF) is encoded as 0xC4 44 82 FF FF. Note that an empty list is encoded as 0xC0.
- Finally, if the first byte is between 0xF8 and 0xFF, the serialised content represents a list whose encoded length in bytes is between 56 and 16EiB and it is encoded as the concatenation of the RLP encoding of its elements following the first byte whose value is 0xF7 plus the length in bytes of the big-endian integer representing the length of the encoded list and said integer, e.g. a list of 1024 integers all equal to 255 is encoded as 0xF9 04 00 81 FF … 81 FF.

Bloom filters

A Bloom filter is a probabilistic data structure, a string of 32 bytes, used to test if an element is a member of a set. For a given element, a test of the Bloom filter generates either of the following two outputs: definitely not a member or maybe a member. Bloom filters might provide a false positive, hence "maybe" but never a false negative, hence "definitely".

In Ethereum, they are used as helpers to search for the logs generated during the execution of transactions without having to resort to include the logs in the block. The elements in the set are the keys to the logs, i.e., the account that produced the log message and the topics of the message. When searching for a particular type of message produced by a contract, it suffices to traverse the block chain: for each block test the key against the Bloom filter stored in the block header, if response is 'definitely not', continue to the next block on the chain, if response is 'maybe yes', extract the transaction receipts of the block and iterate through them, for each receipt, test the Bloom filter of the receipt, again if 'definitely not', go to next receipts, if 'probably yes', re-execute the transactions to find the log.

Merkle Patricia trees

Merkle Patricia trees are used within Ethereum wherever there is a need to verify, through a Merkle proof, that a particular item is indeed included in a larger dataset, e.g., to verify that a particular account and its state at a given block was part of the world state at that point in time. They are designed to allow efficient Merkle proofs as well as updates, with execution time proportional to the logarithm of the number of nodes in the trie. Storage is also optimised: an update of a leaf in the node does not need the storage of a full new trie but the leaf and branches on the path to the root of the trie. The Ethereum implementation is a modification of the original implementation in Ripple, that tries to optimise the storage of the trie by codifying consecutive branches with a single child as a single branch, known as extension.

MPTs combine the features of two trees: a radix tree and a Merkle tree. As a radix tree, it implements a map between a key and a value, both a string of bytes. The nibbles of the bytes (the four right and left-most bits in the byte) constitute an alphabet of 16 symbols, i.e., 0..F in hexadecimal representation. The path from a leaf to the root represents the key of the map. We would refer thereafter the radix tree map as a (path, value) pairing. As an example, the resulting tree for the following pairings, (0xA05FE, 0x123), (0xA05F, 0x4567), (0xA055D, 0x890) and (0x5E, 0xABC) is represented by the diagram in Figure 1 where we can observe that each node is implemented as a list of 17 items. The first 16 items correspond to the symbols of the alphabet and contain a

pointer to the next node of the tree as a path is followed and the last element is the value stored by the node.

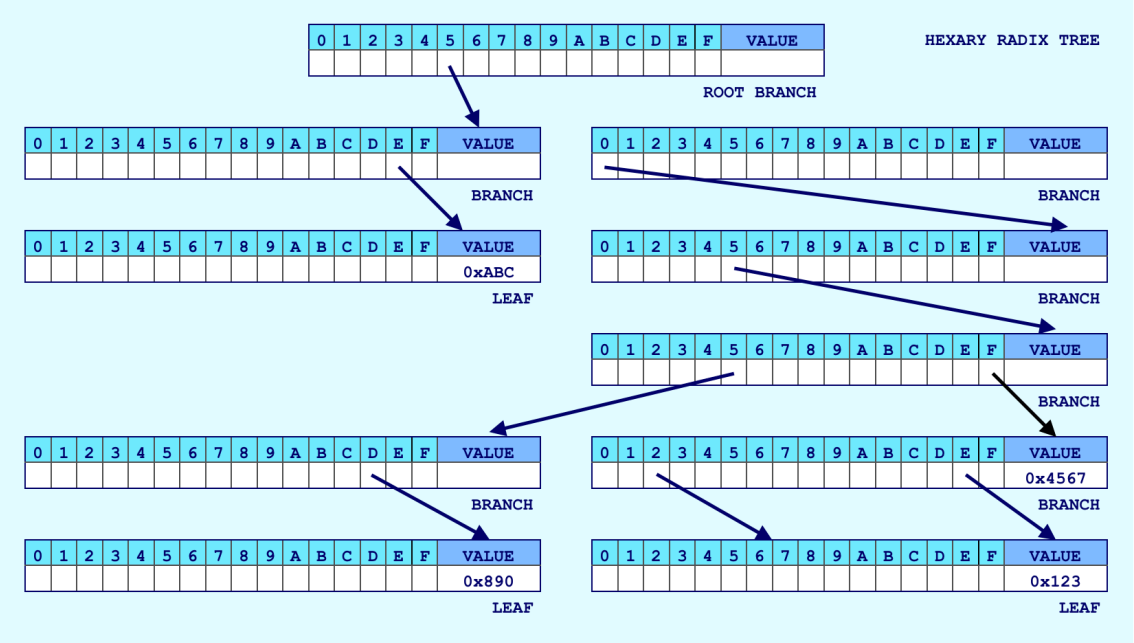

Figure 1 : Hexary radix tree example

For the sake of storage efficiency, the trie is compacted as follows:

- any branch (including root) with no value and with diverging paths is still implemented as a list of 17 elements,
- any branch with value, irrespective of having or not diverging paths, is still implemented as a list of 17 elements,
- any branch (including root) with no value and with no diverging path is implemented as a list of two elements, the encoded path and a pointer to the node at the end of the encoded path and it will be referred thereafter as an extension,
- any leaf is implemented as list of two elements, the encoded path and the value.

The encoded path is built as follows, the first nibble encodes whether the node is an extension or a leaf as well as whether the path is an even or odd sequence of symbols, thus the first nibble will always be one of the following values (in binary):

- 0000 – even path extension,
- 0001 – odd path extension,
- 0010 – even path leave,
- 0011 – odd path leave.

In the case of even paths a 0 nibble is added at the end of the encoded path to maintain overall evenness of the encodedPath. As an example, the even path extension 0xAB CD would be encoded as 0x0A BC D0.

The pointers to the nodes mentioned above are no other than the look up key in the key/value database that constitutes the local storage of the node, generally LevelDB. The keys are generated as the Keccak-256 hash of the RLP encoding of the list representing the node. As those lists might contain the keys to other nodes, a Merkle tree is effectively built.

The diagram in Figure 2 shows the compacted version of the trie featured in the diagram in Figure 1.

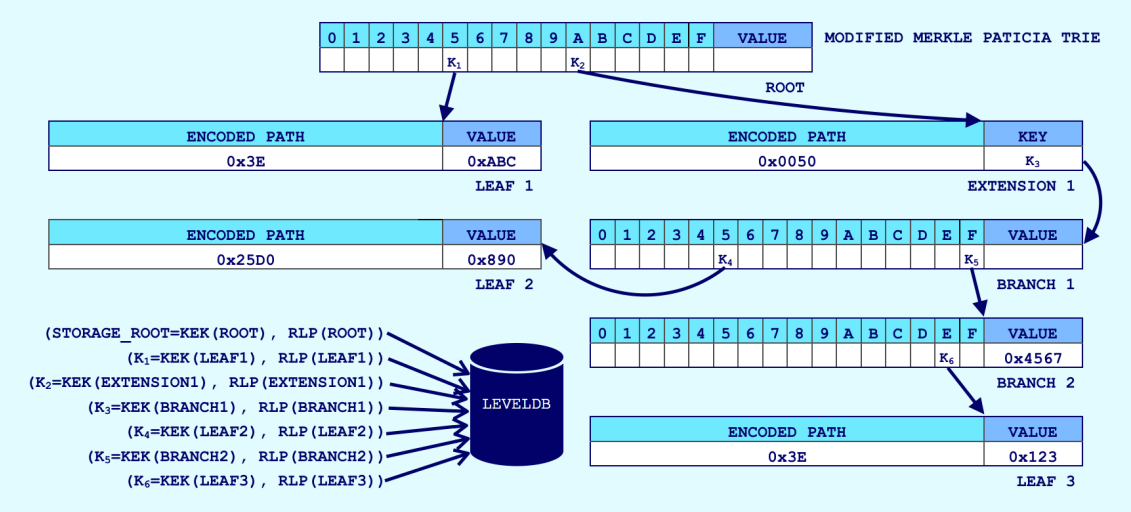

Figure 2 : Modified Merkle Patricia Trie

Amendments to the trie, either updates, inserts or deletes do not require the recalculation of the entire trie, nor the creation of a new copy in storage if we need to keep the history of the data structure. It suffices to recalculate and create the records along the path of the modified data. Following with the previous example, a modification of key/value pair (0xA055D, 0x890) to (0xA055D, 0x891) would be stored as seen in Figure 3. This allows for the efficient preservation (with the obvious exception of deletes) of the history of modifications of the trie.

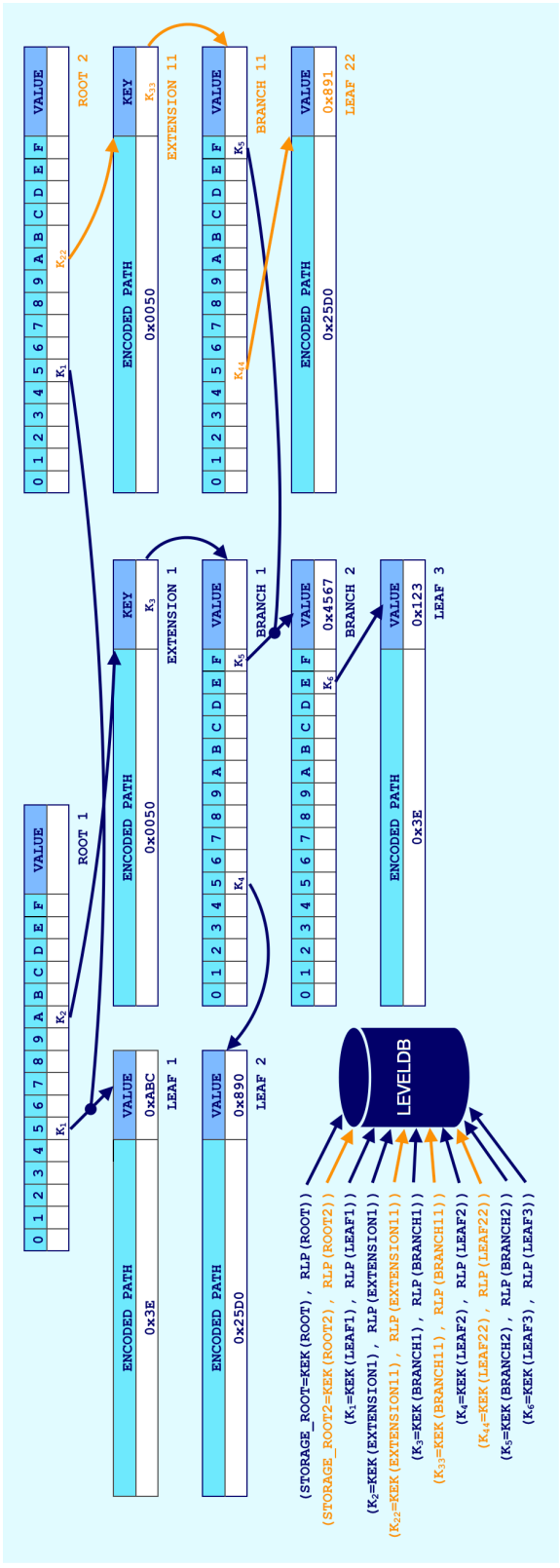

Figure 3 : Update on Modified Merkel Patricia Trie

The account and the state trie

An account is represented by its address, a 160-bit identifier. In order to generate and account and its address, a private/public key pair is generated using the ECDSA – secp256k1 algorithm. The address corresponds to the right most 160-bits of the Keccak-256 hash of the public key.

Currently, Ethereum accounts can be of two types:

- **externally owned accounts**, an account controlled by its private key and owned by a human user or a program external to the blockchain, they are used to send transactions to other accounts, either to transfer value or execute a contract, and
- **contract accounts**, controlled by its code. They can receive transactions from externally owned accounts or message calls from other contract accounts. Upon reception of the transaction/message, if valid, the code of the account is executed (at least until the exhaustion of the amount of gas stated in the transaction).

This might change in the near future where all accounts will be contract accounts and the current externally owned ones will be implemented as a simple contract.

Each account state contains:

- **nonce**, the number of transactions sent by the account,
- **balance**, the amount of Wei owned by the account,
- **storageRoot**, a pointer to the root of the trie where the data used by the account is stored and
- **codeHash**, the hash of the compiled bytecode that implements the contract of the account, the hash is the key to an entry on the local storage database, as the code is not modifiable, there is no need to implement it as a trie.

All accounts are stored as nodes in a trie, the account trie, where the path into the trie is the Keccak-256 of the account address. It is worth mentioning that externally owned accounts are only included in the account trie as they enter into their first transaction; creating an account in the node, simply generates the public/private keys and the derived account address.

The account storage trie

Contracts store their state data in a MPT where the data correspond to the words stored by the EVM instruction SSTORE and the key is built by the compiler of the chosen language in which the contract is written according to the data structure to store. For example, Solidity will pack state variables of elementary types, such integers, etc., together in a single word if possible and

distribute complex data types such as maps and dynamic arrays across the trie with a given algorithm to calculate the key in the trie according to their offset and map keys [43].

The transaction

A transaction is a signed message broadcast to the network by one node from one of the accounts under its control. There are two types of transactions: (a) contract creation, a message whose successful processing results in the creation of a contract account and (b) message call transaction, a message that results in one or many message calls contracts, see below.

When a contract calls another contract, the message sent is known as a message call. It can be likened to a function call within the EVM execution environment where the input parameters are a series of system parameters: the sender, the originator of the transaction, the recipient of the message call, the account of the contract being called, the available gas value, gas price and an array with the functional parameters, i.e., the interface of the contract. At the end of the execution of a message call, an output value is returned to the caller. Note that message calls, as opposed to the transactions that generate them are not serialised into the transaction trie nor are broadcast over the network and exist within the execution environment of the EVM.

A transaction contains the following fields:

- **nonce**, the number of transaction sent by the sender which must coincide with the nonce of the sender account in the current state to avoid replay attacks,
- **gasPrice**, the price in Wei that the sender is to pay for unit of gas spent during the execution of the transaction,
- **gasLimit**, the maximum number of gas units to be spent during the execution of the transaction,
- **to**, the address of the recipient (empty if the transaction is used to create a contract),
- **value**, the amount of Wei to be transferred to the recipient,
- **init**, a byte array with the EVM code for the account initialisation procedure if the transaction is to create a new contract account,
- **data**, arbitrary message or function call to a contract if the transaction represents a message call,
- **v**, **r** and **s**, the parameters of the signature.

and it is built as follows:

- all the fields except the signature are RLP encoded,
- the resulting RLP buffer is hashed using Keccak-256,

- the result is signed using the private key of the sender, thus obtaining the three parameters of the signature,
- all the fields in the signed transaction are RLP encoded and transmitted over the wire.

Note that the sender's account is not explicitly part of the transaction message, however it is retrieved as the signature is validated.

All transactions are permanently stored in the body of the block where they were processed as well as in a trie where the value stored in the leaves is the RLP encoding of the signed transaction and the path to the leaf is the hash of the value, known as TxHash.

The pool of pending transactions

Nodes implement a pool of pending transactions which is fed from the transactions received over the wire. However, the protocol does not mandate the implementation of such data structure and it is left to each implementation to decide how to handle the incoming transactions.

The transaction receipt

The transaction receipt is a data structure that contains the information related to the execution of the transaction as it is included in a block. It contains the following fields:

- the post-transaction state,
- the cumulative gas used in the block containing the transaction,
- the list of logs entries created as the transaction was executed, a log entry being itself a list containing the address of the logger (the contract the transaction was sent to), a list of log topics and the logged message, where topics are 32 or fewer byte long identifiers which allow for the search of message types and
- a Bloom filter built upon the information in the logs entries mentioned above.

All receipts are stored in the receipts trie where the value stored in the leaves is the RLP encoding of receipt fields and the path to the leaf is the TxHash.

The block

The block data structure contains three items forming a list:

- the header,
- the list of ommer block headers and

- the list of transactions included in the block.

The header of the block (or ommer block) contains the following fields:

- **parentHash**, the Keccak-256 hash of the parent block's header,
- **ommersHash**, the Keccak-256 hash of the list of ommer blocks headers,
- **beneficiary**, the account address to deposit all the fees, transaction gas and mining rewards, granted for the creation of the block,
- **stateRoot**, the Keccak-256 hash (the LevelDB key) of the root node of the state trie after the execution of the transactions included in the block,
- **transactionsRoot**, the Keccak-256 hash (the LevelDB key) of the root node of the transaction trie at the time of the block finalisation.
- **receiptsRoot**, the Keccak-256 hash (the LevelDB key) of the root node of the receipt trie at the time of the block finalisation,
- **logsBloom**, the Bloom filter built upon the logs generated by all the transaction receipts generated as the transactions were executed during block finalisation,
- **difficulty**, the difficulty calculated for this block as part of the finalisation process,
- **number**, the number of ancestor blocks since the genesis block, also known as block height,
- **gasLimit**, the maximum number of gas units to be spent per block, this is a parameter set at node configuration level, however as the block validation only allows for a certain deviation of its value with regards to the difficulty stated in the parent block, the nodes need to agree on the parameter value,
- **gasUsed**, the amount of gas used as the transactions of this block were executed,
- **timestamp**, Unix-like time (i.e. number of seconds elapsed since midnight on the 1st of January 1970 in UTC time) at the time of the block creation,
- **extraData**, an arbitrary string of 32 bytes or fewer, corresponding to any information that the miner wants to attach to the block,
- **mixHash**, a 256-bit hash generated as one of the two outputs of the PoW algorithm used, proves that the correct dataset was used as a parameter for the PoW function.
- **nonce**, the 64-bit number used in conjunction with the hash of the header (excluding the nonce and the mixHash) and the dataset as parameters of the PoW function.

Blocks are stored in the database broken up into header and body, both serialised with RLP and accessible through a key based on the hash of the header.

4.3. The consensus layer

At the time of writing, Ethereum is using PoW as the basis for its consensus layer. It is implemented with a few particularities, the calculation algorithm, Ethash, and the accounting of orphaned blocks, the ommers.

Ethash

The PoW mining algorithm currently used by Ethereum is Ethash. The implementation of this algorithm tries to penalise ASICs and favour CPU/GPU computation by requiring a large amount of memory for its completion. A large dataset known as DAG is created every certain number of blocks and it depends only on the block number. The header of the block (excluding the nonce and the mixHash parameters that are dependant of PoW), the nonce and the DAG set are the parameters of the PoW function. The result of the computation returns two values, the mixHash field to be stored in the block header for subsequent verification of PoW and a number that needs to be below or equal to 2256 divided by the block difficulty for the block with the current nonce to be valid. The current block difficulty is calculated from the previous block difficult and the difference in timestamps between the current and previous block, in such fashion that the difficulty of the current block decreases as the time difference between the blocks increases, as the goal is to maintain a relatively constant rate of block creation. As per other blockchains, the verification of PoW is designed to be a simple computation.

Ommers

As nodes concurrently mine the unprocessed transactions into blocks, the blockchain becomes a blocktree, either due to delays in the propagation of the newly minted blocks across the network or due to the lower difficulty of a block with respect to other blocks with the same parent, these discarded blocks are known as ommers. In Ethereum, nodes are incentivised to include ommer headers in their blocks by granting them extra mining rewards per ommer included, limited to a maximum of two ommers per block and with the extra condition of not exceeding six levels of ancestry with the block. The miners that mined an ommer that is included in a block receive a fraction of the reward that the miner of the block does. Also, the fork rule, i.e., the rule to choose a canonical chain amongst all the branches of a blocktree, includes the ommers difficultly in the calculation of the accumulated difficultly of the chain. These rules are based on the "Greedy Heaviest Observed Subtree" (GHOST) protocol [44] which was first proposed for Bitcoin and they are implemented with the aim of allowing faster confirmation times without loss of security or increase of centralisation.

4.4. The application layer

In this section we will discuss the Ethereum Virtual Machine which is responsible of changing the system (or world) state as it executes the transactions to be added to a block. Also, we will briefly discuss smart contracts.

The Ethereum Virtual Machine

The EVM is the component of the node of the network responsible for the execution of the transactions and therefore the sole component allowed to change the system state. The EVM is a Turing Complete machine with the added characteristic that its computation and resource consumption is metered (gas being the unit of measurement) and bounded by the gas limit of the transaction that is being processed. The reason behind this limitation is to avoid the halting problem (infinite loop/execution) that affects Turing complete machines which would cause DoS attacks. The fees levied, gas is paid for in ether, are both a means to provide miners with an incentive to select complex transactions and simultaneously to discourage contracts being written as profligate consumers of computation power and resources such as storage on-chain. Note that the fees are carefully selected to incentivise 'good' behaviour and a refund in some cases of resource liberation might be provided.

The EVM is implemented with a stack based architecture and the size of its word, 256 bits, is designed to accommodate the hash and elliptic-curve computations used across the protocol. The instruction set of the EVM include the arithmetic and logic operations and the flow control and memory and stack access instructions expected of a Turing machine plus a few specialised instructions for hashing, management of gas, block access and logging operations specific to Ethereum. However there are no instructions for elliptic cryptography, these, along with other complex and frequently executed operations, are coded and implemented as part of the protocol in the form of contracts, known as pre-compiled contracts, which can be viewed as system calls.

The EVM execution cycle: the block finalisation

Any node in the network will follow an infinite loop of block finalisations, the process of validating a broadcast block and updating the state of the locally stored blockchain. The same cycle applies if the node is mining a new block or assisting a specialised mining application.

The first stage of block finalisation is the verification (or selection, if mining) of the validity of the block headers of the ommers included in the block (or to be

included) and of their ancestry relation to the current block, that is, that the ommers and the current block share a common ancestor and that the depth of the ancestry, i.e. the number of generations, is below the networkwide threshold.

A block header, of the ommers as well as the broadcast block, is validated according to the following rules: (a) the block number must be equal to the parent's plus one, (b) the stated difficulty of the block must correspond to the calculated one, which is a function of the difficulty of the parent, the time elapsed between the creation of the block and its parent and the block number, (c) the gas limit of the block is within a percentage of the parent's gas limit and above a protocol-defined threshold, (d) the timestamp of the block is greater than the parent's, (e) the extra data on the block header is within the maximum size specified by the protocol and (f) the proof of work stated in the block is correct.

The second stage is the validation and execution of the transactions included in the block. Transactions are validated prior to their execution to ensure that (a) it is well-formed RLP, (b) the signature is valid, (c) the nonce is equal to the nonce of the account in the account trie, (d) the gas limit of the transaction is above the minimum gas calculated for the transaction, (e) the sender's account balance is above the upfront cost of the execution, i.e. the product of the gas limit by the gas price plus the value to be transferred if any, and (f) the gas limit of the transaction plus the cumulative gas consumed from the transaction receipt of the previously processed transaction is below the gas limit. As the execution of transaction commences, the state of the sender's account is irrevocably changed, i.e., the changes would not be rolled back if the transaction execution subsequently produces an exception, to increment the nonce of the account and reduce the balance by the upfront cost. Note that any gas not consumed during the execution of the transaction will be refunded at the end of the process.

The third stage is the modification of the beneficiary accounts of this block and the ommers to collect the block and ommers reward fees. Note that any fees related to the gas consumption of the transactions in the block have already been credited to the beneficiary of the current block in the previous stage.

The fourth and final stage is the verification of the match between the stateRoot stated in the block and that calculated so far in the process of finalising the block (or set the stateRoot of the block if mining) and that the proof of work of the block is correct (or calculate it if mining).

Smart contracts

Business logic is implemented through the use of contracts and a collection of them would constitute a fully fledged application.

Generally, contracts will not be written directly in EVM bytecode but in a purpose-built high-level language that will eventually compile into bytecodes. The initial high-level languages implemented were Mutan (a go like language), Serpent (Python like) and LLL (Lisp like). Whereas Mutan was abandoned in the early stages and Serpent has been recently deprecated for security reasons [11], LLL is seeing a revival due to its features that allow for a closer access to the low-level implementation of the EVM and its well optimised compilation into bytecode [19]. On the other hand, Solidity, a javascript like language, is meant to be the primary development language for contracts and indeed the most widely used. Finally a new Python like language, Viper is currently under development with a focus on security and simplification.

From a high level perspective, the interface to a contract, coded in the data field of the message call, is defined by the Application Binary Interface (ABI) in which the list of parameters and their types is specified.

Finally, several design patterns for contracts are emerging and they are being analysed and surveyed in the academic literature [4]. Also, tools for the formal verification of contracts, such as OYENTE [35] are being developed to increase the security of contract development.

4.5. Future developments

Blockchains and Ethereum in particular are relatively new technologies and consequently the platform is still evolving and needs to address a series of issues [17], amongst them the viability of its consensus protocol, scalability and privacy, as well as technological debt such as the design of its serialisation protocol or internal data structures [23]. In this section we will briefly describe what the plans are to address some of those concerns.

Proof of Stake

Proof of Work consensus algorithms, such as the one currently used by Ethereum, are notorious for their waste of energy as the calculations involved contribute exclusively to the security of the network and do not produce any meaningful results, furthermore they are subject to attacks such as the 51%. Proof of Stake is an algorithm where consensus is achieved through a set of participants in the network, the validators, which upon their voting power based in their economic stake on the network, a deposit of ether, decide on the state of the chain and crucially, provide finality, i.e. as opposed to PoW,

there is no possible reversal of a state due to a parallel chain overtaking the currently canonical one. The migration of PoW to PoS, codenamed Casper, has always been part of the roadmap of Ethereum, planned as the core feature of the Serenity release. There are currently two parallel proposals of implementation under research [10, 52].

Scalability

Blockchains and Ethereum in particular, in their current form, present at least two problems of scalability. Nodes of the network are generally required to store the whole blockchain and in the case of Ethereum, the whole state, in order to secure the network, as the usage of the network grows, the disk requirements to store such data will outgrow the capacity of consumer hardware such as desktop and laptops, not to mention mobile phones [1]. In addition to that, the rate at which transactions are processed is far from the rates at established, and centralised, payment networks such as Visa and Mastercard. Several proposals exist to tackle these problems with varying degrees of integration with the blockchain.

- State Channels are a connection established temporarily between two parties where the transactions between them are netted and allowed within the context of the channel as long as the balance does not exceed a collateral deposit. The interaction with the blockchain is limited to the establishment and the closure of the channel. An implementation for Ethereum can be found in the Raiden Network[1].
- Subchains such as the ones implemented by the Plasma framework [40] where the subchain holds a state and processes transactions among the participants with a mechanism to match the subchain state with a root chain, i.e., Ethereum.
- Sharding [24] where the network and the state of the blockchain are divided into several partitions, known as shards. Transactions between accounts within a shard are processed within that shard and a protocol that connects either synchro- or asynchronously the partitions is available so that a consensus between shards is shared across the entire network, i.e., the set of all shards. The implementation of such construct is not free of concerns, mostly in terms of inter-shard security and several approaches to its design with varying degrees of maximalism are possible.

State channels and subchains such as the ones implemented by Plasma are known as level 2 technologies as their run on top of an existing blockchain, as opposed to the concept of sharding which is a modification of the protocol that implements the blockchain, in this case, Ethereum.

[1] https://raiden.network/101.html

Privacy

As the state of Ethereum is public, the information stored in it is also public. In order provide some degree of privacy, Ethereum implements a zero-knowledge proof in the form of zk-SNARKS [41] that allows to shield the information generated by the execution of a transaction and yet allow verification by the peers of the network.

4.6. Security

Despite early efforts in strengthening the security of the platform such as a security bounty program, all along its short history the Ethereum project has been plagued with numerous attacks. The most notorious attacks show a surface of attack ranging from programming bugs, such as the heist of the DAO, due to a reentrancy problem, underpricing of certain operations of the EVM, which lead to the DoS attacks in June 2016, to the unauthorised removal of contract code that lead to the freezing of funds held on the Parity multi-sig wallet in November 2017. Not surprisingly, a prominent developer warned publicly about using Ethereum for Production projects [53].

Academic research is gathering pace and it is so far focused on network/PoW attacks, such as the 51% and eclipse attacks [25] and application bugs: re-entrancy, [3] including effort towards formal verification of contracts [35].

5. Set up of the development environment

In this section we will describe the existing implementations of Ethereum nodes, libraries and other development environment issues such as the test networks. We will then make a choice and justify it.

5.1. Study of existing node implementations

As opposed to other blockchain platforms, e.g., Bitcoin and bitcoind, Ethereum does not have a canonical node client software. At the time of writing, several implementations exist, developed in different languages and with different degrees of operability and maintenance.

Client	Language	Latest Version (date)
go-ethereum	Go	1.7.3 (21st Nov 2017)
Parity	Rust	1.7.11-stable (28th Dec 2017)
cpp-ethereum	C++	Not clear (possibly 10th Apr 2017)
pyethapp	Python	1.5.1a0 (23rd Oct 2017)
ethereumjs-lib	Javascript	N/A, set of libraries rather than ready made client node
Ethereum(J)	Java	1.6.3 (3rd Nov 2017)
ruby-ethereum	Ruby	N/A, set of libraries rather than ready made client node
ethereumH	Haskell	Possibly abandoned

Table 1 - Client node implementations

However go-ethereum, also known as geth, and Parity clients are by far the most popular and stable[1].

5.2. Testnets

At the time of writing there are three test networks, known as testnets:

- ROPSTEN – a proof of work based network,
- RINKEBY – a proof of authority based network for the use of geth clients.
- KOVAN – a proof of authority based network for the use of Parity clients.

[1] See https://www.ethernodes.org/network/1

Note that proof of authority is a consensus algorithm where only a restricted set of nodes are allowed to create blocks. The lack of costly computation makes it suitable for the maintenance of a test network. However, geth and Parity teams developed different algorithms and therefore generated two different incompatible networks.

Note that for the earlier phases of development and testing, a private network could be a more suitable solution.

5.3. Study of existing APIs

The nodes offer an API accessible by JSON RPC calls which allow access to blockchain data such as account states, blocks, etc., however it is reputed to be a slow method to access the state [30]. Most crucially, it does not expose the complete list of accounts present on the state at a given block, but the list of accounts owned by the node. Therefore we would not be able to extract the state in its entirety.

Several web based APIs exist to obtain statistics, see a list below. At first sight there seem to be a web frontend to a node running alongside the webserver that delivers the web page, so the functionality they use seems to be similar to the JSON RPC API of the nodes mentioned above. They also impose a fair usage policy.

- https://etherscan.io/apis
- https://developers.blockapps.net
- http://docs.infura.apiary.io

On the other hand, there is a healthy ecosystem of Python libraries related to Ethereum [36], possibly due to the fact that is a popular language amongst the Ethereum researchers, which will allow us to access the local storage directly and therefore extract the full world state.

5.4. Rationale for our choice

We will write our library in Python as we will be able to use the existing libraries mentioned above, which will provide the necessary utilities to encode and decode RLP data, scan a trie, etc. Python also has the advantage of rapid prototyping and, in terms of calculating and displaying statistics, it has a range of specialised libraries such as NumPy and matplotlib, a choice also seen in other statistical studies of blockchains [16]. Finally, Jupyter notebooks, also associated with Python, will allow us to create an interactive environment to extract and visualise the data that will allow the user to experiment, as opposed to a web based UI which will restrict the possibilities of use to the use cases foreseen during its design.

There is no strong reason to choose a geth over Parity, however geth is the most popular of the two and there are Python libraries to integrate with.

5.5. Set up of the workspace

We choose Linux as our development and working environment, in particular we use the Ubuntu distribution in its latest stable release: 16.04.3 LTS codename xenial. We set up a virtual machine with VirtualBox and we install the following components :

- geth – latest stable version,
- Python 3.5 and its packaging tool pip – if not already present,
- Jupyter – note that some of tools and libraries needed for some of the functionality, such as conversion to pdf, might require the installation of supplementary components.

For a development environment we will also need the following components.

- Integrated Development Environment (IDE) – we choose Pycharm,
- Version Control System (VCS), we use git and we host the repository in GitHub[1].

Note that we establish a private network that is used as a testing fixture and our library results are compared with their equivalent obtained via the JSON-RPC API provided by the geth client.

5.6. Library dependencies

Our library uses several existing Python libraries for the extraction of the data from the LevelDB database, its subsequent processing and testing.

The following libraries provide the utils to read and decode the data from the database and retrieve tries, accounts, blocks, etc.

- leveldb – provides the interface with the LevelDB database,
- rlp – provides functions to decode the RLP encoded data found in the LevelDB database,
- eth_utils – provides several functions for conversions between strings and arrays of bytes, address formatting, etc.,
- ethereum – provides the class to scan a trie and some other utility functions.

The following libraries provide the data structures needed for the efficient extraction and analysis of the data.

- numpy – popular package for scientific computing in Python, provides an efficient array object and it is used by pandas

[1] https://github.com/carlesperezj/ethereum-analysis-tool

- pandas – provides the data structure and the tools for analysis,
- matplotlib – provides the plotting capabilities.

Finally, the following libraries provide the functionality needed for testing.

- web3 – provides a Python wrapping to the JSON-RPC interface of the geth client,
- py-geth – provides tools to control the geth client programatically from Python code,
- pytest – provides a testing framework.

5.7. The geth client

This section describes the characteristics of the geth client that affects the design of the library and its use.

Synchronisation mode

The geth client, as described in the Client nodes and wallets section above, offers different modes of synchronisation. When running the node to obtain a LevelDB database to analyse, care needs to be taken to parameterise the command line to select the mode that suits the needs of the analysis, e.g., if we only intend to analyse the state on the last block, it suffices to start the node in fast mode; if we intend to analyse the whole history, the node will need to start in full (archive) mode as the fast mode only generates the state after the synchronisation is completed. Note that a full history requires a sizeable disk space [1].

Use of the database

The geth client stores several "shortcuts" in the LevelDB database besides the tries, blocks and pending transaction pool. These shortcuts will be used across the library to locate the data required by the library efficiently. As it can be seen in the code[1], several key formats are defined that allow easy access of blocks, block hash by number, the pre-image of the address hash or the hash of the latest block in the chain.

[1] See var declaration section in https://github.com/ethereum/go-ethereum/blob/master/core/database_util.go

6. Design of the library

We define classes corresponding to some of the Ethereum data structures described in the data layer section: the account and the block header. The attributes of these classes will generally correspond to the fields described above and we define several methods that will help with the data extraction, taking into consideration what information is more likely to be extracted, e.g., the code of a contract itself is unlikely to be susceptible of an aggregate query, however its size might. Two modules implement the core functionality of the library: the statedataset module, which extracts the world state at a given stateroot into a pandas data frame, and the blockrange module, which defines a consecutive range of blocks within the blockchain and provides the functionality to iterate over the range and extract the data into a pandas series.

6.1. The statedataset module

The statedataset modules contains two classes: the Account class and the StateDataset class. This module provides the functionality needed to extract the information about the state of the network at a given block.

The Account class

The Account class models the account data structure of the Ethereum networks. In addition to the fields described in the data model section, it also holds two flags to indicate whether the account is found within the statedataset and whether the address is stored in the database or we only have its hash. It provides several methods implemented as properties to convey extra information about the account: a boolean to denote the account as a contract, the size of the contract code and the associated storage, if applicable. Finally, it provides two alternative constructors as class methods: one to build the object from the RLP buffer stored in the database and a second one to build the object when the account is not found in the database.

The StateDataset class

The StateDataset class is the class that extracts the accounts contained in a state trie given by its root. It holds four attributes: the database and trie objects, the state root that points to the state trie in the database and a flag to convey whether the state trie is indeed found in the database or not. Two methods are defined to extract the set of accounts in the trie as a dictionary

and as a pandas dataset. Finally, a method to provide an account object for a given account address which encapsulates the fact that accounts are stored in the trie by the Keccak-256 hash of the address rather than the address itself.

6.2. The blockrange module

The blockrange module contains the classes that allow for the exploration of the blockchain, i.e., the namesake class that defines a range of consecutive blocks in the chain and on the class that holds the data related to the block header.

The BlockHeader class

The BlockHeader class models the block header data structure defined in the protocol and contains all the fields described in the block section in the data layer analysis above as attributes. The class has several methods to retrieve the data from the local storage, amongst them a method to get the closest block for a given timestamp and a method to get the latest block in the database.

The BlockRange class

The BlockRange class define a range of consecutive blocks in the blockchain. Its attributes are simply the database handler, the lower and upper block numbers in the range and the current block number within the range during an iteration. An alternative constructor has been defined to create a range from an interval of dates which calculates what is the closest block number for the given timestamp. The iteration over the range returns a Block object whose attributes can then be extracted to build a pandas series.

6.3. Usage

The ethereum-analysis-tool library is intended for a technical user who wishes to analyse the state of the Ethereum network. Basic knowledge of scientific Python and in particular the numpy, pandas and matplotlib packages as well as the use of Jupyter notebooks is assumed.

An example of usage can be found in Appendix B.

7. Conclusions

We have surveyed the Ethereum arena and shown that despite the hype around blockchains, there is some serious technology in the making with an ever increasing interest from private companies and academia. We have analysed the current implementation of the protocol and provided a view in depth of its data structures. We have followed the analysis with the development of tools to interface between the existing Ethereum libraries to extract data from the local storage of a node and well known data analysis packages in Python to facilitate the analysis and visualisation of the information contained in the blockchain.

7.1. Future work

Several lines of work are possible after this study. First and most obvious is to use the tool to analyse the state of the mainnet and extract conclusions about its current usage such as the level of activity of accounts and contracts, the amount of data held on-chain by the contracts, etc. Second line of work could be to extend the library to extract the information contained in the data structures of the protocol other than the block headers and the state. In particular, extracting the transaction data and analyse the network generated by the transaction flows with Python libraries such as networkX. Finally, the Ethereum protocol is far from being static, it is rather evolving at a fast pace. Current lines of research such as Proof of Stake or sharding will have a significant impact on the shape of the data structures that support the protocol and consequently an update of the tool would be needed.

References

[1] AFRI, Schoedon. The Ethereum-blockchain size will not exceed 1TB anytime soon. dev.to. 2017. URL: https://dev.to/5chdn/the-ethereum-blockchain-size-will-not-exceed-1tb-anytime-soon-58a

[2] Autorité des Marchés Financiers. Discussion paper on Initial Coin Offerings (ICOs). 2017. URL: http://www.amf-france.org/technique/multimedia?docId=workspace://SpacesStore/a2b267b3-2d94-4c24-acad-7fe3351dfc8a_en_1.0_rendition

[3] ATZEI, Nicola; BARTOLETTI, Massimo; CIMOLI, Tiziana. A Survey of Attacks on Ethereum Smart Contracts (SoK). In: International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2017. p. 164-186.

[4] BARTOLETTI, Massimo; POMPIANU, Livio. An empirical analysis of smart contracts: platforms, applications, and design patterns. arXiv preprint arXiv:1703.06322, 2017.

[5] BONNEAU, Joseph, et al. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, 2015. p. 104-121.

[6] BUTERIN, Vitalik, et al. Ethereum white paper. 2013.

[7] BUTERIN, Vitalik. DAOs, DACs, DAs and More: An Incomplete Terminology Guide. Ethereum Blog. 2014. URL: https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide

[8] BUTERIN, Vitalik. Notes on Blockchain Governance. Vitalik Buterin's website. 2017. URL: http://vitalik.ca/general/2017/12/17/voting.html

[9] BUTERIN, Vitalik. A Prehistory of the Ethereum Protocol. Vitalik Buterin's website. 2017. URL: http://vitalik.ca/general/2017/09/14/prehistory.html

[10] BUTERIN, Vitalik. GRIFFITH, Virgil. Casper the Friendly Finality Gadget. 2017. URL: https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf

[11] CASTOR, Amy. One of Ethereum's Earliest Smart Contract Languages Is Headed for Retirement. Coindesk. 2017. URL: https://www.coindesk.com/one-of-ethereums-earliest-smart-contract-languages-is-headed-for-retirement

[12] CACHIN, Christian; VUKOLIĆ, Marko. Blockchains Consensus Protocols in the Wild. arXiv preprint arXiv:1707.01873, 2017.

[13] CHAUM, David. Blind signatures for untraceable payments. In: Advances in cryptology. Springer US, 1983. p. 199-203.

[14] COLLINSON, Patrick. Bitcoin investors could lose all their money, FCA warns. The Guardian. 2017. URL: https://www.theguardian.com/business/2017/sep/12/cryptocurrency-investors-bitcoin-could-lose-money-fca-warns

[15] Colored Coins Developers. The Colored Coins Protocol. Colored Coins wiki. 2017. URL: https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification/wiki

[16] DELGADO-SEGURA, Sergi, et al. Analysis of the Bitcoin UTXO set. Cryptology ePrint Archive. 2017. URL: https://eprint.iacr.org/2017/1095

[17] DE SILVA, Matthew. Seven Critiques of Ethereum According To The Creator. URL: https://www.ethnews.com/seven-critiques-of-ethereum-according-to-the-creator. ethenews.com. 2017.

[18] EHRSAM, Fred. Blockchain Governance: Programming Our Future. Medium. 2017. URL: https://medium.com/@FEhrsam/blockchain-governance-programming-our-future-c3bfe30f2d74

[19] ELLISON, David. An Introduction to LLL for Ethereum Smart Contract Development. Medium. 2017. URL: https://media.consensys.net/an-introduction-to-lll-for-ethereum-smart-contract-development-e26e38ea6c23

[20] Ethereum Developers. EIP Purpose and Guidelines. GitHub. 2017. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1.md

[21] Ethereum Developers. JSON RPC. Ethereum Wiki. 2017. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC

[22] Ethereum Developers. Light client protocol. Ethereum Wiki. 2017. URL: https://github.com/ethereum/wiki/wiki/Light-client-protocol

[23] Ethereum Developers. Wishlist. Ethereum Wiki. 2017. URL: https://github.com/ethereum/wiki/wiki/Wishlist

[24] Ethereum Developers. Sharding Introduction. 2017. URL: https://github.com/ethereum/sharding/blob/develop/docs/doc.md

[25] GERVAIS, Arthur, et al. On the security and performance of proof of work blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016. p. 3-16.

[26] Etherscan Developers. Ethereum Developer APIs. 2017. URL: https://etherscan.io/apis

[27] Go-ethereum Developers. Native: Introduction. go-ethereum wiki. 2017. URL: https://github.com/ethereum/go-ethereum/wiki/Native:-Introduction

[28] HELMORE, Edward. Are cryptocurrencies about to go mainstream? The Guardian. 2017. URL: https://www.theguardian.com/technology/2017/jul/01/cryptocurrencies-mainstream-finance-bitcoin-ethereum

[29] HERTIG, Alyssa. Loveable Digital Kittens Are Clogging Ethereum's Blockchain. Coindesk. 2017. URL: https://www.coindesk.com/loveable-digital-kittens-clogging-ethereums-blockchain

[30] HORROCKS, Richard. How to access Geth's state trie. Ethereum StackExchange. 2017. URL: https://ethereum.stackexchange.com/questions/25620/how-to-access-geths-state-trie

[31] JENTZSCH, Christoph. The History of the DAO and Lessons Learned. Medium. 2016. URL: https://blog.slock.it/the-history-of-the-dao-and-lessons-learned-d06740f8cfa5

[32] JOHNSON, Nick. EIP 137 Ethereum Domain Name Service – Specification. 2016. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-137.md

[33] LAMPORT, Leslie; SHOSTAK, Robert; PEASE, Marshall. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982, 4.3: 382-401.

[34] LI, Yang, et al. EtherQL: A Query Layer for Blockchain System. In: International Conference on Database Systems for Advanced Applications. Springer, Cham, 2017. p. 556-567.

[35] LUU, Loi, et al. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016. p. 254-269.

[36] MERRIAM, Piper. The Python Ethereum ecosystem. Medium. 2017. URL: https://medium.com/@pipermerriam/the-python-ethereum-ecosystem-101bd9ba4de7

[37] MONAGHAN, Angela. Bitcoin is a fraud that will blow up, says JP Morgan boss. The Guardian. 2017. URL: https://www.theguardian.com/technology/2017/sep/13/bitcoin-fraud-jp-morgan-cryptocurrency-drug-dealers

[38] NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. 2008.

[39] O'HARA, Kieron. Smart Contracts-Dumb Idea. IEEE Internet Computing, 2017, 21.2: 97-101.

[40] POON, Joseph; BUTERIN, Vitalik. Plasma: Scalable Autonomous Smart Contracts. *White paper*, 2017. URL: https://plasma.io/plasma.pdf

[41] REITWIESSNER, Christian. zkSNARKs in a nutshell. 2016. URL: http://chriseth.github.io/notes/articles/zksnarks/zksnarks.pdf

[42] SCHNEIER, Bruce. Ethereum Hacks. Schneier on Security. 2017. URL: https://www.schneier.com/blog/archives/2017/07/ethereum_hacks.html

[43] Solidity Developers. Layout of State Variables in Storage. Read the Docs. 2016-2017. URL: http://solidity.readthedocs.io/en/develop/miscellaneous.html#layout-of-state-variables-in-storage

[44] SOMPOLINSKY, Yonatan; ZOHAR, Aviv. Secure high-rate transaction processing in bitcoin. In: International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2015. p. 507-527. 40

[45] STECKLOW, Steve et al. Chaos and hackers stalk investors on cryptocurrency exchanges. Reuters. 2017 URL: https://www.reuters.com/investigates/special-report/bitcoin-exchanges-risks

[46] SZILÁGYI, Péter. Difference between a pruned and unpruned blockchain. Ethereum StackExchange. 2016. URL: https://ethereum.stackexchange.com/questions/1229/difference-between-a-pruned-and-unpruned-blockchain

[47] TRÓN, Viktor. FISHER Aron. Generalised swap swear and swindle games. Dropbox. 2017. URL: https://www.dropbox.com/s/7r3jasjho35ojc7/sw3paper.pdf

[48] TSCHORSCH, Florian; SCHEUERMANN, Björn. Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Communications Surveys & Tutorials, 2016, 18.3: 2084-2123.

[49] VOGELSTELLER, Fabian. BUTERIN, Vitalik. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md

[50] WOOD, Gavin. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014, 151.

[51] ZAMFIR, Vlad. Against on-chain governance. Medium. 2017. URL: https://medium.com/@Vlad_Zamfir/against-on-chain-governance-a4ceacd040ca

[52] ZAMFIR, Vlad. Casper the Friendly Ghost A "Correct-by-Construction" Blockchain Consensus Protocol. 2017. URL: https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf

[53] ZAMFIR, Vlad. Ethereum isn't safe or scalable. It is immature experimental tech. Don't rely on it for mission critical apps unless absolutely necessary!. Twitter. 4 Mar 2017 4:40 URL: https://twitter.com/vladzamfir/status/838006311598030848 .

Appendix A

We provide in this annex the original task break out and planning.

As defined in the methodology section above, the tasks in which this project is to be divided are tightly coupled with the list of goals, see the list of tasks below in Table 3. The organisation of the time is mostly sequential and only a few tasks can be done in parallel. We planned a roughly equal distribution of the time between the theoretical and the practical sides, plus a final amount of time dedicated to the composition of the memoir and the presentation. In the two cases where parallel tasks was possible, the dedication to each was distributed as follows:

- Study of current implementations/tooling and internal data structures of the protocol – 80%, Set up of development environment – 20%.
- On the composition of the memoir, the write up of survey materials – 50%, and the documentation of the API and UI – 50%.

The resulting Gantt chart is represented in Figure 4

Task	Duration (days)	Dedication	Start	Finish	Predecessors
Study of the Ethereum platform.	10		9 Oct 2017	20 Oct 2017	
Research	4	1	9 Oct 2017	12 Oct 2017	
Document	6	1	13 Oct 2017	20 Oct 2017	2
Study of current implementations and tooling	5		23 Oct 2017	27 Oct 2017	1
Research	2	1	23 Oct 2017	24 Oct 2017	
Document	3	0.8	25 Oct 2017	27 Oct 2017	5
Study of the internal data structures of the Ethereum protocols	10		30 Oct 2017	10 Nov 2017	4
Research	4	0.8	30 Oct 2017	2 Nov 2017	
Document	6	0.8	3 Nov 2017	10 Nov 2017	8
Set up of a development environment.	13	0.2	25 Oct 2017	10 Nov 2017	5
Development of API	15		13 Nov 2017	1 Dec 2017	7
Design	3	1	13 Nov 2017	15 Nov 2017	8
Development	6	1	16 Nov 2017	23 Nov 2017	12
Testing	6	1	24 Nov 2017	1 Dec 2017	13
Development of UI	15		4 Dec 2017	22 Dec 2017	11

Design	3	1	4 Dec 2017	6 Dec 2017	12
Development	6	1	7 Dec 2017	14 Dec 2017	16
Testing	5	1	15 Dec 2017	21 Dec 2017	17
Deployment	1	1	22 Dec 2017	22 Dec 2017	18
Memoir	5		25 Dec 2017	29 Dec 2017	15
Selection and collation survey materials	4	0.5	25 Dec 2017	28 Dec 2017	3;6;9
Documentation of the design and implementation of the API and UX.	4	0.5	25 Dec 2017	28 Dec 2017	12;16;17;19
Review of the ensemble	1	1	29 Dec 2017	29 Dec 2017	22
Presentation	5		1 Jan 2018	5 Jan 2018	20
Design and composition of the presentation materials	3	1	1 Jan 2018	3 Jan 2018	
Recording and editing of the presentation media	2	1	4 Jan 2018	5 Jan 2018	25

Table 2 - List of tasks

We could therefore estimate the effort during weekdays as follows and use the weekends and bank holidays as contingency.

total weekdays	65
total hours (9 credits * 25 hours/credit)	225
hours/weekday	3.46

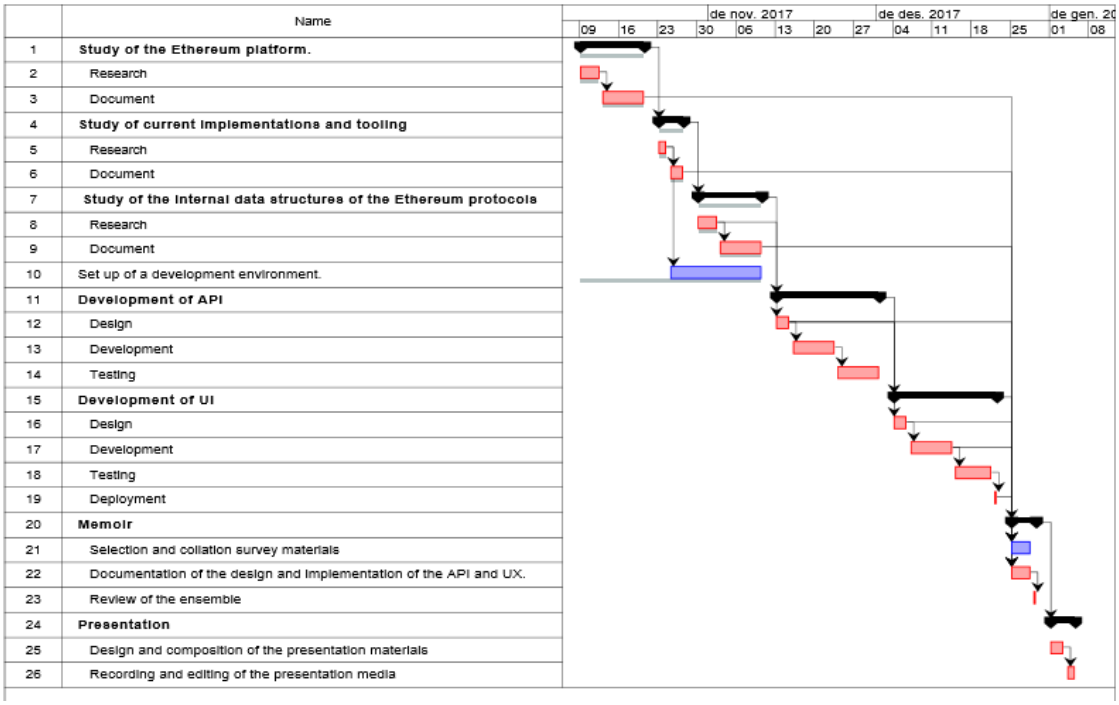

Figure 4 - Project Gantt chart.

Appendix B

A typical notebook will have a first set of cells setting the environment with all the necessary imports, the opening of the database and the logging level handler, see an example below.

```
import logging

%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from ethereum_stats import levelDB, statedataset, blockrange

LOGGING_LEVEL = logging.WARNING
logging.basicConfig(level=LOGGING_LEVEL)

DB_DIR = '/home/ethereum/eth-rinkeby/datadir/geth/chaindata'
db = levelDB.LevelDB(DB_DIR)
```

Subsequent cells will extract and manipulate the data and plot the results, again see an example below of extracting the state of the last block into a pandas dataframe, .applying filtering and plotting the results.

```
last_block = blockrange.BlockHeader.get_latest_block_header(db)
state = statedataset.StateDataset(db, last_block.state_root)
df = state.to_panda_dataframe()
plt.figure()
df[df.nonce>1000]['nonce'].plot.hist()
```

The example below shows how to extract data out of a range of blocks and store it into a pandas series.

```
factor = 10 ** 18
accAddress =
df.loc[df[df.key_in_db==True].nonce.idxmax()].account
aRange= blockrange.BlockRange.date_range(db, '7/1/2018 15:30:00',
'7/1/2018 18:00:00')
s = pd.Series(np.zeros(aRange.upper_blk_nbr -
aRange.lower_blk_nbr + 1))
i = 0
for blk in aRange:
    st = statedataset.StateDataset(db,
decode_hex(blk.state_root))
    acc = st.get_account(accAddress)
    s[i] = acc.balance / factor
    i = i + 1
s.plot()
```