UNIVERSITY OF THESSALY

MASTER THESIS

Inspecting and Analyzing Blockchain Applications

Author:
Dimitrios Greasidis

Supervisor:
Dr. Manolis VAVALIS

A thesis submitted in fulfillment of the requirements for the degree of Master

in the

Department of Electrical and Computer Engineering

June 20, 2018

Declaration of Authorship

I, Dimitrios Greasidis, declare that this thesis titled, "Inspecting and Analyzing Blockchain Applications" and the work presented in it are my own. I, Dimitrios Greasidis, confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Πανεπιστήμιο Θεσσαλίας

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Μεταπτυχιακό Δίπλωμα

Έλεγχος και ανάλυση **blockchain** εφαρμογών

Δημήτριος Γρεασίδης

Η σημερινή εποχή αδιαμφισβήτητα χαρακτηρίζεται από το μεγάλο πλήθος πληροφορίας, η οποία είναι πλήρως κεντρικοποιημένη. Αυτό το γεγονός καθιστά αναγκαστική την τυφλή εμπιστοσύνη προς τους οργανισμούς - αρχές που κατέχουν αυτές τις πληροφορίες, χωρίς την δυνατότητα ελέγχου και αμφισβήτησης των ενεργειών τους στις προσωπικές μας πληροφορίες. Με βασικό ερέθισμα το προηγούμενο πρόβλημα τα τελευταία χρόνια έχουν αναπτυχθεί συστήματα που προσπαθούν να αποκεντροποιήσουν όλες τις διαδικασίες με σκοπό η εμπιστοσύνη του συστήματος να διασφαλίζεται μέσω της ίδιας της αρχιτεκτονικής του. Η αρχιτεκτονική αυτών των συστημάτων είναι γνωστή ως blockchain και ουσιαστικά αναπαριστά ένα δημόσιο κατάστιχο, όπου όλες οι κινήσεις είναι φανερές. Αρχικά, αυτή η τεχνολογία χρησιμοποιήθηκε για την δημιουργία κρυπτονομισμάτων με σκοπό την αποκεντροποίηση του χρήματος, αλλά με την πάροδο του χρόνου βλέπουμε ότι όλο και περισσότερες ιδέες βασίζονται πάνω σε αυτή την αρχιτεκτονική. Σε κάποια από τα κρυπτονομίσματα που δημιουργήθηκαν εισήχθη για πρώτη φορά η έννοια του smart contract , το οποίο ουσιαστικά είναι ένα ¨συμβόλαιο¨, σε κάποια γλώσσα προγραμματισμού. Όταν εκπληρωθούν οι συνθήκες του εκτελούνται κάποιες ενέργειες, χωρίς την ανάγκη ύπαρξης κάποια κεντρικής αρχής. Το πιο γνωστό κρυπτονόμισμα αλλά και το πρώτο που εισήγαγε αυτή την έννοια είναι το Ethereum. Εξαιτίας του μεγάλου εύρους των προβλημάτων που μπορεί να λύσει η τεχνολογία των smart contracts, όλο και περισσότεροι προγραμματιστές υλοποιούν συστήματα και εφαρμογές βασισμένες σε αυτά και κατ' επέκταση στο blockchain. Αυτή η τεχνολογία είναι αρκετά πολύπλοκη ώστε να δουλεύει σωστά και να λύνει τα περισσότερα προβλήματα που μπορούν να λυθούν αποκεντροποιημένα. Για αυτό το λόγο ελλοχεύουν πολλοί κίνδυνοι κατά την διάρκεια υλοποίησης μιας εφαρμογής, οι οποίοι μπορεί να οδηγήσουν κάποια στιγμή στην απώλεια χρημάτων και την κατάρρευση μιας εταιρείας. Στόχος αυτής της διπλωματικής είναι η εύρεση και ταυτοποίηση αυτών των κινδύνων πρώτα θεωρητικά, μέσω της μελέτης της αρχιτεκτονικής αυτού του συστήματος, και στην συνέχεια πρακτικά, μέσω της δημιουργίας ενός εργαλείου για την ανίχνευση λαθών αλλά και την επίβλεψη τέτοιων εφαρμογών.

University of Thessaly

Abstract

Department of Electrical and Computer Engineering

Master

Inspecting and Analyzing
Blockchain Applications

Dimitrios Greasidis

Nowadays the amount of information is enormous and fully centralized. Agencies and authorities that have that information are being trusted blindly and no one can question their actions. The previous problem is the key stimulus that motivates people to design and develop systems that are decentralized. In such a system the trust is guarded by its own architecture. The technology behind those systems is known as **blockchain** which essentially represents a public record where all “movements” are available. Initially, this technology was used to create cryptocurrencies in order to decentralize money, but over time more and more ideas are based on this architecture. In some cryptocurrencies, the concept of **smart contracts** was introduced for the first time, which is essentially a "contract" in a programming language. In that “contract”, when certain conditions are fulfilled, some actions are carried out without the need of a central authority. The most well-known cryptocurrency, but also the first that introduced this concept is Ethereum. Due to the wide range of problems that **smart contracts** can solve, more and more developers are implementing systems and applications based on them, and consequently on blockchain. This technology is quite complex in order to function properly and solve most of the problems in a decentralized way. For this reason, there are many risks during an implementation of a decentralized application (**Dapp**), which can lead to a big loss of money and probably the collapse of a company. The target of this Master Thesis is to locate and identify these risks theoretically, through the study of the architecture of this system, and then practically through the creation of a tool to detect errors and inspect such applications.

Contents

List of Figures

Dedicated to my family

Chapter 1

Introduction

1.1 Motivation

Day after day decentralized applications are preferred from people around the world. The basic advantage of those applications is the absence of a central authority that people must trust even if they don't want to.

That simple reason is establishing more and more over time the decentralized applications in the software development sector. This architecture is adopted to solve many different problems that have the exact same disadvantage, a central authority that everyone must trust.

An example that suffers from that disadvantage is the energy market. Putting it simple, in an energy market there are bids in an auction of energy where the actors are the consumers and the generators of energy. After a specific time that has been defined by the central authority, a clearing price is exported by using the bids that were submitted. The window time is usually 15 minutes. After that, the consumers will buy energy with the price that was exported for the next 15 minutes. The basic questioning that occurs is if the central authority is trustworthy enough and will export a clearing price by getting into consideration all the bidders. In that scenario a decentralized market energy could solve this problem of trustworthiness. Probably the best blockchain that is the best fit for this case is the Ethereum blockchain which is open-source and supports smart contracts, which we will analyze later.

After participating in an experiment that ought to prove that an energy market can be supported by such a network, there was a pretty obvious problem. Before analyzing this problem it should be noticed that a private Ethereum network was created for testing and to achieve a lower cost for the experiment. The problem was that there wasn't enough tools to monitor, inspect and debug an experiment of this type. The basic libraries to create such tools were provided by the company but a tool that was using fully the capabilities of that libraries didn't exist. Thus, it was compulsory to create our tool to debug and monitor the experiment and check that everything was going as expected. This was the incentive to continue the development of that tool further and provide more functions that could help developers to transit on the era of decentralized applications.

1.2 Contribution of this Thesis

During the development of the tool the importance of decentralized applications was noticeable enough. To expand this knowledge and provide an easy start to create such applications the need of concentrated knowledge is almost compulsory. This Thesis oughts to specify the vulnerabilities of a system - application based on blockchain and Ethereum Smart Contracts and create a prototype that detects them and makes easier the implementation and debugging of such applications.

This is really important to create a world with much less control over our actions and data, which is a dream that everyone tries to fulfill.

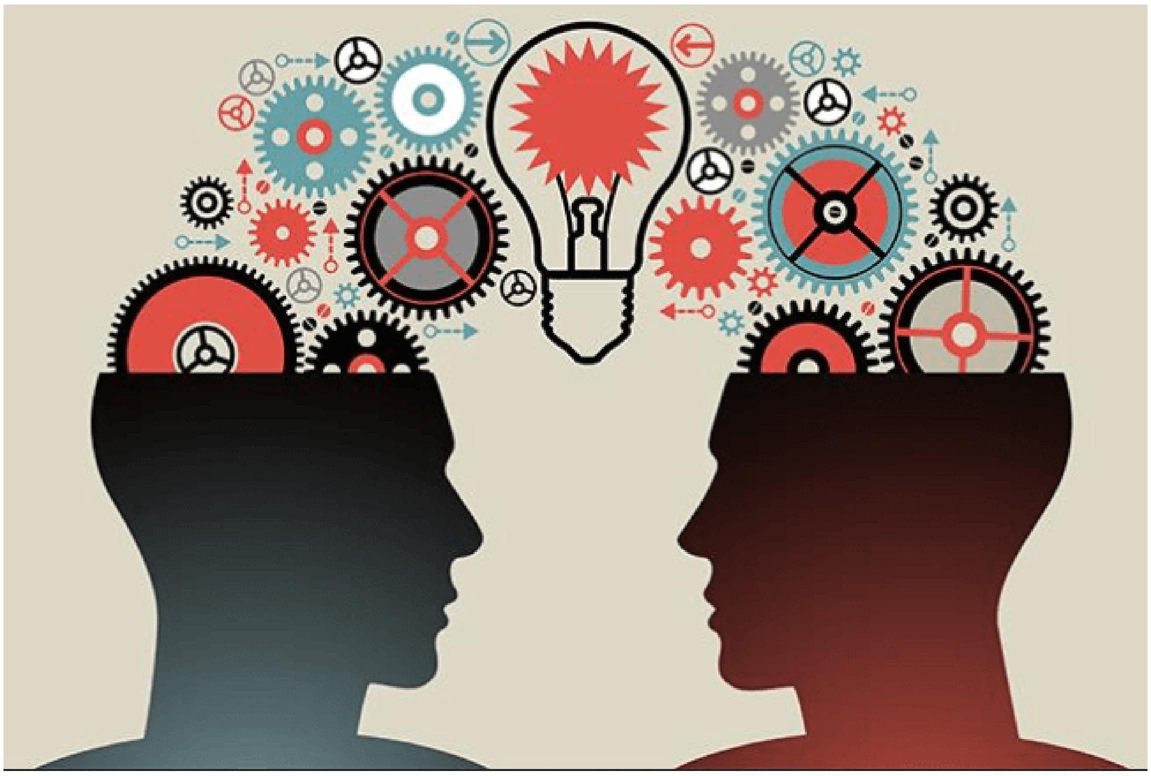


FIGURE 1.1: Knowledge - The next web

Source: https://thenextweb.com/contributors/2017/12/01/recognized-appreciated-knowledge-economy-needs-blockchain/

Chapter 2

Basic Concepts and Enabling Technologies

2.1 Architecture of usual systems

At the begging of the software development history, everything was centralized and in one core-thread. Obviously, this was due to the low quality of technology then. Nevertheless, even when the technology got much better the architecture didn't change to much. The basic step forward was to create all the previous software platforms in a distributed way. The purpose of that was to solve the scaling problem because of the continues increasing data.

Distributed systems solved some basic problems and we are using them until today. The basic concept was to create a lot of similar servers, so that the platform could serve more requests in the same time which implies more users. Nevertheless, in such systems the user must trust the platform with his data without having any way to check the validity of the platform's actions. This problem was emerging more and more over the years, the problem of centralization alongside with the big power of data.

An indirect suggestion came up from Satoshi Nakamoto who was the creator of the first decentralized platform [1]. The platform successfully implemented this logic, known also as blockchain architecture, and since then is the state of the art technology to create a decentralized platform. The picture below demonstrates the basic differences between the previous architectures of systems.

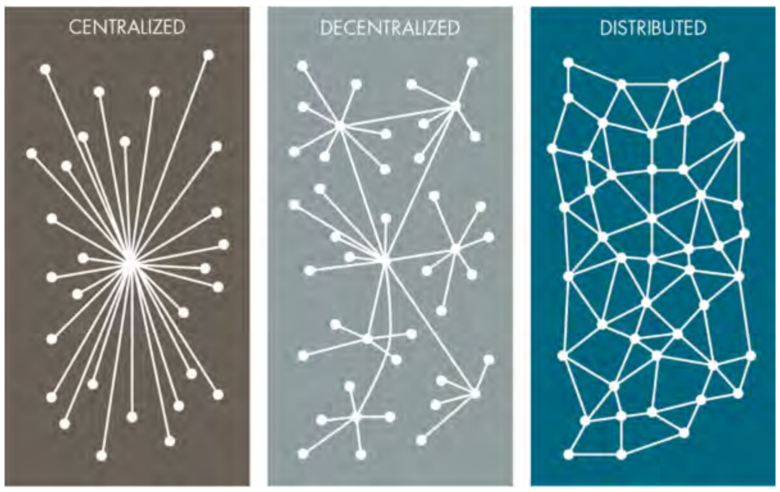


FIGURE 2.1: Types of networks.

Source: https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality

2.2 Blockchain

The blockchain is an undeniably ingenious invention – the brainchild of a person or group of people known by the pseudonym, Satoshi Nakamoto [2]. But since then, it has evolved into something greater, and the main question every single person is asking is: What is Blockchain?

By allowing digital information to be distributed but not copied, blockchain technology created the backbone of a new type of internet. Originally devised for the digital currency, Bitcoin, the tech community is now finding other potential uses for the technology. Bitcoin has been called "digital gold," and for a good reason. To date, the total value of the currency is close to $136,806,493,619 USD [3] [4].

A distributed database. Thus, what blockchain really is? Picture a spreadsheet that is duplicated thousands of times across a network of computers. Then imagine that this network is designed to regularly update this spreadsheet and you have a basic understanding of the blockchain [1] [2].

Information held on a blockchain exists as a shared — and continually reconciled — database. This is a way of using the network that has obvious benefits. The blockchain database isn't stored in any single location, meaning the records it keeps are truly public and easily verifiable. No centralized version of this information exists for a hacker to corrupt. Hosted by millions of computers simultaneously, its data is accessible to anyone on the internet [1] [2].

Blockchain Durability and robustness. Blockchain technology is like the internet in that it has a built-in robustness. By storing blocks of information that are identical across its network, the blockchain cannot:

- Be controlled by any single entity.
- Has no single point of failure.

Bitcoin was invented in 2008. Since that time, the Bitcoin blockchain has operated without significant disruption. To date, any of problems associated with Bitcoin have been due to hacking or mismanagement. In other words, these problems come from bad intention and human error, not flaws in the underlying concepts. The internet itself has proven to be durable for almost 30 years. It's a track record that bodes well for blockchain technology as it continues to be developed [1] [2].

Transparent and incorruptible. The blockchain network lives in a state of consensus, one that automatically checks in with itself every ten minutes. A kind of self-auditing ecosystem of a digital value, the network reconciles every transaction that happens in ten-minute intervals. Each group of these transactions is referred to as a "block". Two important properties result from this:

- **Transparency data** is embedded within the network as a whole, by definition it is public.
- **It cannot be corrupted** altering any unit of information on the blockchain would mean using a huge amount of computing power to override the entire network.

In theory, this could be possible. In practice, it's unlikely to happen. Taking control of the system to capture Bitcoins, for instance, would also have the effect of destroying their value [1] [2].

A network of nodes. A network of so-called computing "nodes" make up the blockchain. Together they create a powerful second-level network, a wholly different vision for how the internet can function.

Every node is an "administrator" of the blockchain, and joins the network voluntarily (in this sense, the network is decentralized). However, each one has an incentive for participating in the network: the chance of winning Bitcoins.

Nodes are said to be "mining" Bitcoin, but the term is something of a misnomer. In fact, each one is competing to win Bitcoins by solving computational puzzles. Bitcoin was the purpose of the blockchain as it was originally conceived. It's now recognized to be only the first of many potential applications of the technology.

The idea of decentralization. By design, the blockchain is a decentralized technology. Anything that happens on it is a function of the network as a whole. Some important implications stem from this. By creating a new way to verify transactions aspects of traditional commerce could become unnecessary. Stock market trades become almost simultaneous on the blockchain, for instance — or it could make types of record keeping, like a land registry, fully public. And decentralization is already a reality [1] [2].

A global network of computers uses blockchain technology to jointly manage the database that records Bitcoin transactions. That is, Bitcoin and other similar cryptocurrencies are managed by its network, and not any one central authority. Decentralization means the network operates on a user-to-user (or peer-to-peer) basis. The forms of mass collaboration this makes possible are just beginning to be investigated.

FIGURE 2.2: Blockchain.

Source: http://www.documentarytube.com/articles/what-is-blockchain-the-technology-explained

Who will use the blockchain? As web infrastructure, you don't need to know about the blockchain for it to be useful in your life.

Currently, finance offers the strongest use cases for the technology. International remittances, for instance. The World Bank estimates that over $430 billion US in money transfers were sent in 2015. And at the moment there is a high demand for blockchain developers.

The blockchain potentially cuts out the middleman for these types of transactions. Personal computing became accessible to the general public with the invention of the Graphical User Interface (GUI), which took the form of a "desktop". Similarly, the most common GUI devised for the blockchain are the so-called "wallet" applications, which people use to buy things with Bitcoin, and store it along with other cryptocurrencies.

Transactions online are closely connected to the processes of identity verification. It is easy to imagine that wallet apps will transform in the coming years to include other types of identity management.

The Blockchain & Enhanced security. By storing data across its network, the blockchain eliminates the risks that come with data being held centrally.

Its network lacks centralized points of vulnerability that computer hackers can exploit. Today's internet has security problems that are familiar to everyone. We all rely on the "username/password" system to protect our identity and assets online. Blockchain security methods use encryption technology [1] [2].

The basis for this are the so-called public and private "keys". A "public key" (a long, randomly-generated string of numbers) is a users' address on the blockchain. Bitcoins sent across the network gets recorded as belonging to that address. The "private key" is like a password that gives its owner access to their Bitcoin or other digital assets. Store your data on the blockchain and it is incorruptible. This is true, although protecting your digital assets will also require safeguarding of your private key by printing it out, creating what's referred to as a paper wallet [1] [2].

A second-level network With blockchain technology, the web gains a new layer of functionality. Already, users can transact directly with one another — Bitcoin transactions in 2016 averaged over $200,000 US per day. With the added security brought by the blockchain new internet business are on track to unbundle the traditional institutions of finance [3].

2.3 Networks

As mentioned previously cryptocurrencies is the effort to decentralize the currency itself and solve the problem of the manipulation through blockchain. Bitcoin was the first one and probably for that reason one of the main cryptocurrencies right now. Although Bitcoin introduced a decentralized and robust currency system, which was fully accepted, a lot of restrictions and problems came along with its architecture. A simple result of the Bitcoin architecture is the huge amounts of energy that this system needs to function, due to the so-called mining process. For that reason a lot of new cryptocurrencies were introduced, which solved efficiently some problems of Bitcoin and added even more functionalities. Below we will analyze and compare the architecture of some of the most well known and innovative cryptocurrencies.

2.3.1 Bitcoin

As mentioned previously Bitcoin was the first of the networks built on blockchain architecture. This network enabled transactions, transfers of money, between people across the world. The first problem that Bitcoin solved was the slow transfer of money over continents with the usual bank systems that required several days to complete a transfer [1]. Bitcoin also provided transactions with low cost of money instead of the several dollars that were needed to transfer money between different countries [2].

Obviously, the decentralization of money was the main target of Bitcoin, based on the blockchain technology. This was achievable because of the nature of the network, which means that no government or no rich person can shut down or ban Bitcoin. The network itself is powerful and stable because of the nodes that participate in the mining process. We can visualize it as a really big network of computers, that communicate and sync with each other.

The mining process represents a problem that each node has to solve to generate the next block on the chain. The first to solve this problem is rewarded with a static price from the network (it is changing over the years) and with the transaction fees from the transactions. The transactions fees is the price that every user has to pay if he wants to make a transaction. The problem is random, pointless (from the aspect of a useful solution) and with a very high power and computational cost. This meaningless process ensures the safety and decentralization of the network because the miner that will be selected for the next block is completely random. The mining process also includes the validation of the transactions that will be inserted in the block. After that the block is published to the network and when the other nodes sync with it, they try to validate the block and then add it to their local chain. If the block is not valid it is rejected. This whole process is also known as the Consensus algorithm that ensures the stability of the network, solves the double spending problem and protects the network from malicious attacks. The consensus algorithm of Bitcoin is "proof of work", which we analyzed briefly. The main concept is that the validity of a node is ensured through the proof of work to solve a certain problem, which is "mining". However, the generation of a new block is happening every 10 minutes and each block contains in average 2020 transactions, which number in a global scale is very small. Thus, to support a global cryptocurrency a lot of solutions were proposed to increase the number of transactions per block, such as doubling the size of each block or implement a new Consensus algorithm which could ensure the safety of the network but also make it a lot faster.

With that in mind, we can visualize a chain of blocks that each one is generated from a random node and contains a number of transactions which are transfers of money between accounts. This chain is copied on every node and it doesn't exist on a certain server. This ensures that even if the power went out it in America the rest world could keep this network alive. Thus, a blockchain network is protected from such attacks.

FIGURE 2.3: Bitcoin.

Source: https://wccftech.com/

Nevertheless, the consensus algorithm of Bitcoin has a very high cost, at a point where a year of mining in Bitcoin consumes as much power as a big European country such as Denmark.

2.3.2 Ethereum

Launched in 2015, Ethereum is the largest and most well-established, open-ended decentralized software platform that enables **Smart Contracts** and Distributed Applications (**ĐApps**) to be built and run without any downtime, fraud, control or interference from a third party. Ethereum is not just a platform but also a programming language (Turing complete) running on a blockchain, helping developers to build and publish distributed applications [5].

Ethereum is a peer-to-peer network of virtual machines that any developer can use to run distributed applications (**Dapps**). These computer programs could be anything, but the network is optimized to carry out rules that mechanically execute when certain conditions are met, like a contract. Ethereum uses its own decentralized public blockchain to cryptographically store, execute, and protect these contracts.

Each computer on their network downloads a small virtual machine to sync with the Ethereum blockchain and remains available to execute contracts. This distributed network of computers conveniently provides the security, reliability, and computing power necessary for carrying out designed arrangements. Of course, this consensus network isn't free or private, so developers only use it for consensus on outcomes and when their data can be public.

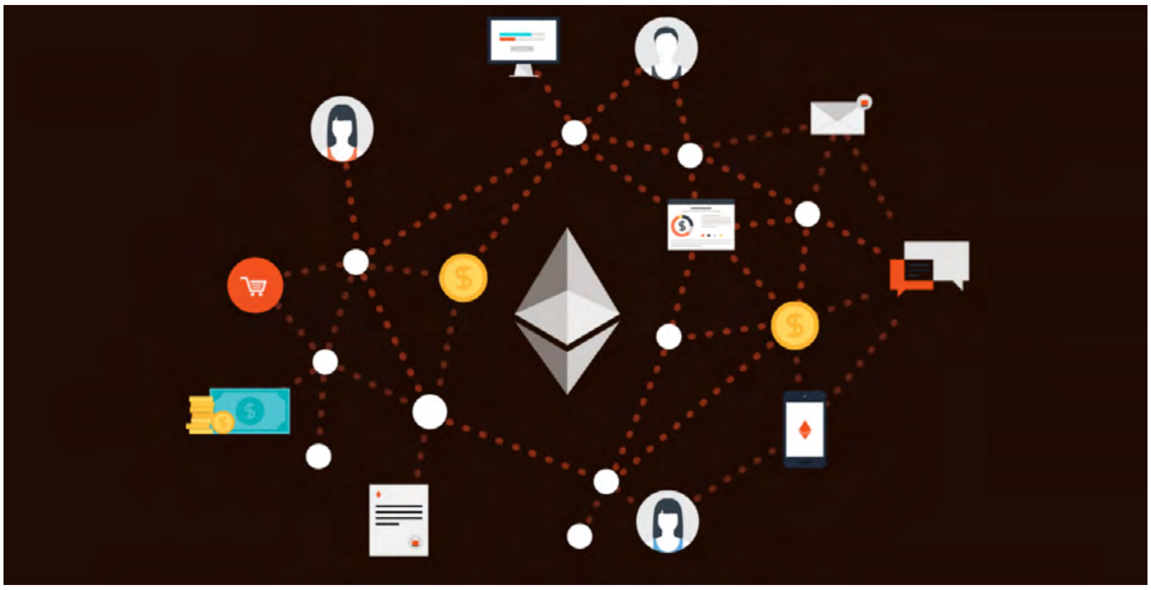


FIGURE 2.4: Ethereum.

Source: https://blockgeeks.com/guides/ethereum/

While most examples of these contracts depict various human interactions, the technology is currently preferred for industrial use-cases like strict business logic between organizations or machine-to-machine communication. For instance, some energy companies are researching ways to create a smarter grid, where houses can automatically buy and sell power. Another example is the International Business Machines Corp. (IBM)/Samsung Group "Internet of Things" collaboration. However, they used a fork of the Ethereum's code – for the sake of privacy – to run how these "things" communicate with each other.

The financial industry loves the promise of controllable blockchain, but not the lack of privacy. So, consulting companies like Eris have forked Ethereum to sell it, bundled with their consulting services, to help banks construct their own private networks.

Ethereum is certainly a novel implementation of virtual machines with amazing possibilities. However, we don't yet know how large the unforked Ethereum network will grow nor how scalable this network could be.

Bitcoin Vs Ethereum. While both Bitcoin and Ethereum are powered by the principle of distributed ledgers and cryptography, the two differ in many technical ways. For example, the programming language used by Ethereum is Turing complete whereas Bitcoin is in a stack based language. Other differences include block time (Ethereum transaction is confirmed in seconds compared to minutes for Bitcoin) and their basic builds (Ethereum uses ethash while Bitcoin uses secure hash algorithm, SHA-256).

However, from a general point of view, Bitcoin and Ethereum differ in purpose. While Bitcoin is created as an alternative to regular money and is thus a medium of payment transaction and store of value, Ethereum is developed as a platform which facilitates peer-to-peer contracts and applications via its own currency vehicle. While Bitcoin and Ether are both digital currencies, the primary purpose of Ether is not to establish itself as a payment alternative (unlike Bitcoin) but to facilitate and monetize the working of Ethereum to enable developers to build and run distributed applications (**ÐApps**).

2.3.3 Stellar Lumens

Stellar Lumens advertises itself as an open-sourced, distributed payments infrastructure, built on the premise that the international community needs "a worldwide financial network open to anyone." Stellar Lumens will fill this need, connecting individuals, institutions, and payment systems through its platform [6].

In doing so, the Stellar Lumen's team wants to make monetary transactions cheaper, quicker, and more reliable than they are under current systems. In addition, their protocol would connect people from all over the world by allowing for more efficient cross-border payments.

Like (most) all other cryptocurrencies, Stellar Lumens bears that beautiful buzzword that has become the hallmark of blockchain technology: decentralization. The Stellar network runs on a web of decentralized servers supported by an international consortium of individuals and entities. These servers support the distributed ledger that keeps track of the network's data and transactions.

In practice, the Stellar protocol will function like a more inclusive, more flexible PayPal. To start using Stellar, you would need to upload funds to an anchor on the network. Much like a bank or PayPal, this anchor then holds your money and issues credit to your virtual wallet in its stead [7].

FIGURE 2.5: Stellar Lumens.

Source: https://www.mycryptopedia.com/stellar-lumens-xlm-explained/

The Stellar Consensus Protocol white paper [6] can seem complex because it's not blockchain. The protocol is more like an evolution of blockchain and that's a good thing. The paper highlights key features: decentralized control, low latency, flexible trust and asymptotic security. Of these four features, proof of work blockchains are only capable of decentralized control and even that can be compromised as we have seen in Bitcoin.

The protocol works through the use of quorums, which are a set of nodes used to reach an agreement. The Stellar network is able to process a large number of transactions quickly and for fractions of a penny because there is no mining involved. The quorums securely reach consensus by exchanging signatures. The result is a platform that solves blockchain's shortfalls right now.

The main feature that Stellar is unique, along with Ripple which may be called his brother from another mother, is that it enables intermediate transactions through other cryptocurrencies. Thus, anyone can buy other cryptocurrencies with Stellar or Ripple with ease.

It is worth mentioning that Stellar was based on Ripple. Those two cryptocurrencies are very similar but Stellar is fully decentralized instead of Ripple, which is centralized. Ripple was adopted initially from banks. For that reason it has a much bigger market capitalization from Stellar, but it is the opposite of what it should be (decentralized) and what blockchain champions.

Chapter 3

Ethereum

3.1 Basic Concept

As mentioned previously Ethereum is one of the top cryptocurrencies with a market capitalization of $41,093,186,257 USD. The basic feature that makes Ethereum very important, useful and innovative is the introduction of Smart Contracts. In a more abstract view a smart contract provides the possibility to add complex logic on a transaction system. It can replace a conveyancer in an easy and effective way.

Because of blockchain technology Smart Contracts cannot be corrupted and cannot be used with any other way than the one that were designed. The basic requirement is a correct architecture for your smart contract and the other requirements are fulfilled by the Ethereum network. This innovation enables the implementation of many different problems in a decentralized way, which many times is the key to the solution.

3.2 Background on Ethereum smart contracts

Ethereum [8] is a decentralized virtual machine, which runs programs — called contracts — upon request of users. Contracts are written in a Turing-complete bytecode language, called EVM bytecode [9]. Roughly, a contract is a set of functions, each one defined by a sequence of bytecode instructions. A remarkable feature of contracts is that they can transfer ether (a cryptocurrency similar to Bitcoin [10]) to/from users and to other contracts.

Users send transactions to the Ethereum network in order to:

- create new contracts;
- invoke functions of a contract;
- transfer ether to contracts or to other users.

All the transactions are recorded on a public, append-only data structure, called blockchain. The sequence of transactions on the blockchain determines the state of each contract, and the balance of each user.

Since contracts have an economic value, it is crucial to guarantee that their execution is performed correctly. To this purpose, Ethereum does not rely on a trusted central authority: rather, each transaction is processed by a large network of mutually untrusted peers — called miners. Potential conflicts in the execution of contracts (due e.g., to failures or attacks) are resolved through a consensus protocol based on “proof-of-work” puzzles. Ideally, the execution of contracts is correct whenever the adversary does not control the majority of the computational power of the network [11].

The security of the consensus protocol relies on the assumption that honest miners are rational, i.e. that it is more convenient for a miner to follow the protocol than to try to attack it. To make this assumption hold, miners receive some economic incentives for performing the (time-consuming) computations required by the protocol. Part of these incentives is given by the execution fees paid by users upon each transaction. These fees bound the execution steps of a transaction, so preventing from denial-of-service attacks where users try to overwhelm the network with time-consuming computations [11].

Programming smart contracts. We illustrate contracts through a small example (AWallet, in 3.1), which implements a personal wallet associated to an owner. Rather than programming it directly as EVM bytecode, we use Solidity, a JavaScript-like programming language which compiles into EVM byte-code. Intuitively, the contract can receive ether from other users, and its owner can send (part of) that ether to other users via the function pay. The hashtable outflow records all the addresses to which it sends money, and associates to each of them the total transferred amount. All the ether received is held by the contract. Its amount is automatically recorded in balance: this is a special variable, which cannot be altered by the programmer [11].

```
 1 contract AWallet{
 2     address owner;
 3 
 4     mapping (address => uint) public outflow;
 5 
 6     function AWallet(){ owner = msg.sender; }
 7 
 8     function pay(uint amount, address recipient) returns (bool){
 9         if (msg.sender != owner || msg.value != 0) throw;
10         if (amount > this.balance) return false;
11         outflow[recipient] += amount;
12         if (!recipient.send(amount)) throw;
13         return true;
14     }
15 }
```

LISTING 3.1: Wallet Contract

Contracts are composed by fields and functions. A user can invoke a function by sending a suitable transaction to the Ethereum nodes. The transaction must include the execution fee (for the miners), and may include a transfer of ether from the caller to the contract. Solidity also features exceptions, but with a peculiar behavior. When an exception is thrown, it cannot be caught: the execution stops, the fee is lost, and all the side effects — including transfers of ether — are reverted.

The function **AWallet** at line 6 is a constructor, run only once when the contract is created. The function **pay** sends **amount** wei **(**$1wei = 10^{-18}$ **ether)** from the contract to **recipient**. At line 9 the contract throws an exception if the caller **(msg.sender)** is not the owner, or if some ether **(msg.value)** is attached to the invocation and transferred to the contract. Since exceptions revert side effects, this ether is returned to the caller (who however loses the fee). At line 10, the call terminates if the required amount of ether is unavailable; in this case, there is no need to revert the state with an exception. At line 11, the contract updates the outflow registry, before transferring the ether to the recipient. The function send used at line 12 to this purpose presents some quirks, e.g. it may fail if the recipient is a contract (see Section 3.3).

Execution fees. Each function invocation is ideally executed by all miners in the Ethereum network. Miners are incentivized to do such work by the execution fees paid by the users which invoke functions. Besides being used as incentives, execution fees also protect against denial-of-service attacks, where an adversary tries to slow down the network by requesting time-consuming computations.

Execution fees are defined in terms of gas and gas price, and their product represents the cost paid by the user to execute code. More specifically, the transaction which triggers the invocation specifies the gas limit up to which the user is willing to pay, and the price per unit of gas. Roughly, the higher is the price per unit, the higher is the chance that miners will choose to execute the transaction. Each EVM operation consumes a certain amount of gas [9], and the overall fee depends on the whole sequence of operations executed by miners.

Miners execute a transaction until its normal termination, unless an exception is thrown. If the transaction terminates successfully, the remaining gas is returned to the caller, otherwise all the gas allocates for the transaction is lost. If a computation consumes all the allocated gas, it terminates with an “out-of-gas” exception — hence the caller loses all the gas. An adversary wishing to attempt a denial-of-service attack (e.g. by invoking a time-consuming function) should allocate a large amount of gas, and pay the corresponding ether. If the adversary chooses a gas price consistently with the market, miners will execute the transaction, but the attack will be too expensive; otherwise, if the price is too low, miners will not execute the transaction.

The mining process. Miners group the transactions sent by users into blocks, and try to append them to the blockchain in order to collect the associated fees. Only those blocks which satisfy a given set of conditions, which altogether are called validity, can be appended to the blockchain. In particular, one of these conditions requires to solve a moderately hard “proof-of-work” puzzle, which depends on the previous block and on the transactions in the new block. The difficulty of the puzzle is dynamically updated so that the average mining rate is 1 block every 12 seconds.

When a miner solves the puzzle and broadcasts a new valid block to the network, the other miners discard their attempts, update their local copy of the blockchain by appending the new block, and start “mining” on top of it. The miner who solves the puzzle is rewarded with the fees of the transactions in the new block (and also with some fresh ether).

It may happen that two (or more) miners solve the puzzle almost simultaneously. In this case, the blockchain forks in two (or more) branches, with the new blocks pointing to the same parent block. The consensus protocol prescribes miners to extend the longest branch. Hence, even though both branches can transiently continue to exist, eventually the fork will be resolved for the longest branch. Only the transactions therein will be part of the blockchain, while those in the shortest branch will be discarded. The reward mechanism, inspired to the GHOST protocol in [12], assigns the full fees to the miners of the blocks in the longest branch, and a portion of the fees to those who mined the roots of the discarded branch. E.g., assume that blocks A and B have the same parent, and that a miner appends a new block on top of A. The miner can donate part of its reward to the miner of the “uncle block” B, in order to increase the weight of its branch in the fork resolution process.

Level	**Cause of vulnerability**
Solidity	Call to the unknown
	Gasless send
	Exception disorders
	Type casts
	Reentrancy
	Keeping secrets
EVM	Immutable bugs
	Ether lost in trasfer
	Stack size limit
Blockchain	Unpredictable state
	Generating randomness
	Time constraints

TABLE 3.1: Taxonomy of vulnerabilities in Ethereum smart contracts.

Compiling Solidity into EVM bytecode. Although contracts are rendered as sets of functions in Solidity, the EVM bytecode has no support for functions. Therefore, the Solidity compiler translates contracts so that their first part implements a function dispatching mechanism. More specifically, each function is uniquely identified by a signature, based on its name and type parameters. Upon function invocation, this signature is passed as input to the called contract: if it matches some function, the execution jumps to the corresponding code, otherwise it jumps to the fallback function. This is a special function with no name and no arguments, which can be arbitrarily programmed. The fallback function is executed also when the contract is passed an empty signature: this happens e.g. when sending ether to the contract.

Solidity features three different constructs to invoke a contract from another contract, which also allow to send ether. All these constructs are compiled using the same bytecode instruction. The result is that the same behavior can be implemented in several ways, with some subtle differences detailed in Section 3.3.

3.3 Vulnerabilities in smart contracts

In this section we systematize the security vulnerabilities of Ethereum smart contracts [11]. We group the vulnerabilities in three classes, according to the level where they are introduced (Solidity, EVM bytecode, or blockchain). Further, we illustrate each vulnerability at the Solidity level through a small piece of code. All these vulnerabilities can be (actually, most of them have been) exploited to carry on attacks which e.g. steal money from contracts. Table 3.1 summarizes our taxonomy of vulnerabilities.

Call to the unknown. Some of the primitives used in Solidity to invoke functions and to transfer ether may have the side effect of invoking the fallback function of the callee/recipient. We illustrate them below.

- **call** invokes a function (of another contract, or of itself), and transfers ether to the callee. E.g., one can invoke the function ping of contract c as follows:

```
c.call.value(amount)(bytes4(sha3("ping(uint256)")),n)
```

LISTING 3.2: Invoke function

where the called function is identified by the first 4 bytes of its hashed signature, amount determines how many wei have to be transferred to c, and n is the actual parameter of ping. Remarkably, if a function with the given signature does not exist at address c, then the fallback function of c is executed, instead.

- **send** is used to transfer ether from the running contract to some recipient r, as in r.send(amount). After the ether has been transferred, send executes the recipient's fallback. Others vulnerabilities related to send are detailed in "**exception disorders**" and "**gasless send**".

- **delegatecall** is quite similar to call, with the difference that the invocation of the called function is run in the caller environment. For instance, executing

```
c.delegatecall(bytes4(sha3("ping(uint256)")),n)
```

LISTING 3.3: DelegateCall

if ping contains the variable this, it refers to the caller's address and not to c, and in case of ether transfer to some recipient d — via d.send(amount) — the ether is taken from the caller balance.

- besides the primitives above, one can also use a direct call as follows:

```
contract Alice { function ping(uint) returns (uint)}
contract Bob {
function pong(Alice c){c.ping(42);} }
```

LISTING 3.4: Multiple Cotracts

The first line declares the interface of **Alice's** contract, and the last two lines contain Bob's contract: therein, **pong** invokes **Alice's ping** via a direct call. Now, if the programmer mistypes the interface of contract **Alice** (e.g., by declaring the type of the parameter as int, instead of uint), and **Alice** has no function with that signature, then the call to **ping** actually results in a call to **Alice's** fallback function.

The fallback function is not the only piece of code that can be unexpectedly executed: other cases are reported in the vulnerabilities "type cast" at 3.3 and "unpredictable state" at 3.3.

Exception disorder. In Solidity there are several situations where an exception may be raised, e.g. if

- the execution runs out of gas
- the call stack reaches its limit
- the command throw is executed

However, Solidity is not uniform in the way it handles exceptions: there are two different behaviors, which depend on how contracts call each others. For instance, consider:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob {
  uint x=0;
    function pong(Alice c) {
      x=1; c.ping(42); x=2;
    }
}
```

LISTING 3.5: Calling other Contract

Now, assume that some user invokes **Bob's pong**, and that **Alice's ping** throws an exception. Then, the execution stops, and the side effects of the whole transaction are reverted. Therefore, the field x contains 0 after the transaction. Now, assume instead that **Bob** invokes **ping** via a **call**. In this case, only the side effects of that invocation are reverted, the call returns false, and the execution continues. Therefore, x contains 2 after the transaction.

More in general, assume that there is a chain of nested calls, when an exception is thrown. Then, the exception is handled as follows:

- if every element of the chain is a direct call, then the execution stops, and every side effect (including transfers of ether) is reverted. Further, all the gas allocated by the originating transaction is consumed;
- if at least one element of the chain is a **call** (the cases **delegatecall** and **send** are similar), then the exception is propagated along the chain, reverting all the side effects in the called contracts, until it reaches a **call**. From that point the execution is resumed, with the **call** returning false. Further, all the gas allocated by the **call** is consumed.

To set an upper bound to the use of gas in a call, one can write:

```
c.call.gas(g)(bytes4(sha3("ping(uint256)")),n)
```

LISTING 3.6: Set upper bound for gas use

In case of exceptions, if no bound is specified then all the available gas is lost; otherwise, only g gas is lost.

The irregularity in how exceptions are handled may affect the security of contracts. For instance, believing that a transfer of ether was successful just because there were no exceptions may lead to attacks. The quantitative analysis in [13] shows that not control the return value of call/send invocations (note however that the absence of these checks does not necessarily imply a vulnerability).

Gasless send. When using the function send to transfer ether to a contract, it is possible to incur in an out-of-gas exception. This may be quite unexpected by programmers, because transferring ether is not generally associated to executing code. The reason behind this exception is subtle. First, note that **c.send(amount)** is compiled in the same way of a call with empty signature, but the actual number of gas units available to the callee is always bound by 2300. Now, since the **call** has no signature, it will invoke the callee's fallback function. However, 2300 units of gas only allow to execute a limited set of bytecode instructions, e.g. those which do not alter the state of the contract. In any other case, the **call** will end up in an out-of-gas exception.

We illustrate the behavior of send through a small example, involving a contract C who sends ether through function pay, and two recipients D1, D2.

```
contract C {
  function pay(uint n, address d){
    d.send(n);
  }
}

contract D1 {
  uint public count = 0;
  function() { count++; }
}
contract D2 { function() {} }
```

LISTING 3.7: Simple Example

There are three possible cases to execute pay:

- $n \neq 0$ and $d = D1$. The send in C fails with an out-of-gas exception, because 2300 units of gas are not enough to execute the state-updating D1's fallback.
- $n \neq 0$ and $d = D2$. The send in C succeeds, because 2300 units of gas are enough to execute the empty fallback of D2.
- $n = 0$ and $d \in \{D1, D2\}$. For compiler versions $< 0.4.0$, the send in C fails with an out-of-gas exception, since the gas is not enough to execute any fallback, not even an empty one. For compiler versions $\geq 0.4.0$, the behavior is the same as in one of the previous two cases, according whether $d = D1$ or $d = D2$.

Summing up, sending ether via send succeeds in two cases: when the recipient is a contract with an unexpensive fallback, or when the recipient is a user.

Type casts. The Solidity compiler can detect some type errors (e.g., assigning an integer value to a variable of type string). Types are also used in direct calls: the caller must declare the callee's interface, and cast to it the callee's address when performing the call. For instance, consider again the direct call to ping:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

LISTING 3.8: Type casts example

The signature of **pong** informs the compiler that c adheres to interface **Alice**. However, the compiler only checks whether the interface declares the function **ping**, while it does not check that:

1. **c** is the address of contract **Alice**;
2. the interface declared by **Bob** matches **Alice's** actual interface.

A similar situation happens with explicit type casts, e.g. **Alice(c).ping()**, where **c** is an address.

The fact that a contract can type-check may deceive programmers, making them believe that any error in checks (1) and (2) is detected. Furthermore, even in the presence of such errors, the contract will not throw exceptions at run-time. Indeed, direct calls are compiled in the same EVM bytecode instruction used to compile call (except for the management of exceptions). Hence, in case of type mismatch, three different things may happen at run-time:

- if c is not a contract address, the call returns without executing any code 12;
- if c is the address of any contract having a function with the same signature as **Alice's ping**, then that function is executed;
- if c is a contract with no function matching the signature of **Alice's ping**, then **c**'s fallback is executed.

In all cases, no exception is thrown, and the caller is unaware of the error.

Reentrancy. The atomicity and sequentiality of transactions may induce programmers to believe that, when a non-recursive function is invoked, it cannot be reentered before its termination. However, this is not always the case, because the fallback mechanism may allow an attacker to re-enter the caller function. This may result in unexpected behaviors, and possibly also in loops of invocations which eventually consume all the gas. For instance, assume that contract Bob is already on the blockchain, when the attacker publishes **Mallory** contract:

```
contract Bob {
  bool sent = false;
  function ping(address c) {
    if (!sent) {
    c.call.value(2)();
    sent = true;
}}}
contract Bob { function ping(); }

contract Mallory {
  function() {
    Bob(msg.sender).ping(this);
  }
}
```

LISTING 3.9: Reentrancy example

The function ping in **Bob** is meant to send exactly **2wei** to some address **c**, using a call with empty signature and no gas limits. Now, assume that ping has been invoked with **Mallory**'s address. As mentioned before, the call has the side effect of invoking **Mallory**'s fallback, which in turn invokes again **ping**. Since variable sent has not already been set to true, **Bob** sends again **2wei** to **Mallory**, and invokes again her fallback, thus starting a loop. This loop ends when the execution eventually goes out-of gas, or when the stack limit is reached (see the "**stack size limit**" vulnerability at 3.3), or when **Bob** has been drained off all his ether. In all cases an exception is thrown: however, since call does not propagate the exception, only the effects of the last call are reverted, leaving all the previous transfers of ether valid.

This vulnerability resides in the fact that function **ping** is not **reentrant**, i.e. it may misbehave if invoked before its termination. Remarkably, the **"DAO Attack"** [14], which caused a huge ether loss in June 2016, exploited this vulnerability.

Keeping secrets. Fields in contracts can be public, i.e. directly readable by everyone, or private, i.e. not directly readable by other users/contracts. Still, declaring a field as private does not guarantee its secrecy. This is because, to set the value of a field, users must send a suitable transaction to miners, who will then publish it on the blockchain. Since the blockchain is public, everyone can inspect the contents of the transaction, and infer the new value of the field.

Many contracts, e.g. those implementing multi-player games, require that some fields are kept secret for a while: for instance, if a field stores the next move of a player, revealing it to the other players may advantage them in choosing their next move. In such cases, to ensure that a field remains secret until a certain event occurs, the contract has to exploit suitable cryptographic techniques, like e.g. timed commitments [15] [16].

Immutable bugs. Once a contract is published on the blockchain, it can no longer be altered. Hence, users can trust that if the contract implements their intended functionality, then its runtime behavior will be the expected one as well, since this is ensured by the consensus protocol. The drawback is that if a contract contains a bug, there is no direct way to patch it. So, programmers have to anticipate ways to alter or terminate a contract in its implementation [17] — although it is debatable the coherency of this with the principles of Ethereum.

The immutability of bugs has been exploited in various attacks, e.g. to steal ether, or to make it unredeemable by any user. In all these attacks, there was no possibility of recovery. The only exception was the recovery from the **"DAO attack"**. The countermeasure was an hardfork of the blockchain, which basically nullified the effects of the transactions involved in the attack [18]. This solution was not agreed by the whole Ethereum community, as it contrasted with the “code is law” principle claimed so far. As a consequence, part of the miners refused to fork, and created an alternative blockchain [19].

Ether lost in transfer. When sending ether, one has to specify the recipient address, which takes the form of a sequence of 160 bits. However, many of these addresses are orphan, i.e. they are not associated to any user or contract. If some ether is sent to an orphan address, it is lost forever (note that there is no way to detect whether an address is orphan). Since lost ether cannot be recovered, programmers have to manually ensure the correctness of the recipient addresses.

Stack size limit. Each time a contract invokes another contract (or even itself via this.f()) the call stack associated with the transaction grows by one frame. The call stack is bounded to 1024 frames: when this limit is reached, a further invocation throws an exception.

Until October 18th 2016, it was possible to exploit this fact to carry on an attack as follows. An adversary starts by generating an almost-full call stack (via a sequence of nested calls), and then he invokes the victim’s function, which will fail upon a further invocation. If the exception is not properly handled by the victim’s contract, the adversary could manage to succeed in his attack. This vulnerability could be exploited together with others: e.g., the **"exception disorder"** and **"stack size limit"** vulnerabilities.

This cause of vulnerability has been addressed by an hard-fork of the Ethereum blockchain [20]. The fork changed the cost of several **EVM** instructions, and redefined the way to compute the gas consumption of **call** and **delegatecall**. After the fork, a caller can allocate at most 63/64 of its gas: since, currently, the gas limit per block is 4,7M units, this implies that the maximum reachable depth of the call stack is always less than 1024 [21].

Unpredictable state. The state of a contract is determined by the value of its fields and balance. In general, when a user sends a transaction to the network in order to invoke some contract, he cannot be sure that the transaction will be run in the same state the contract was at the time of sending that transaction. This may happen because, in the meanwhile, other transactions have changed the contract state. Even if the user was fast enough to be the first to send a transaction, it is not guaranteed that such transaction will be the first to be run. Indeed, when miners group transactions into blocks, they are not required to preserve any order; they could also choose not to include some transactions.

There is another circumstance where a user may not know the actual state wherein his transaction will be run. This happens in case the blockchain forks (see Section 3.2). Recall that, when two miners discover a new valid block at the same time, the blockchain forks in two branches. Some miners will try to append new blocks on one of the branches, while some others will work on the other one. After some time, though, only the longest branch will be considered part of the blockchain, while the shortest one will be abandoned. Transactions in the shortest branch will then be ignored, because no longer part of the blockchain. Therefore, believing that a contract is in a certain state, could be determinant for a user in order to publish new transactions (e.g., for sending ether to other users). However, later on such state could be reverted, because the transactions that led to it could happen to be in the shortest branch of a fork.

In some cases, not knowing the state where a transaction will be run could give rise to vulnerabilities. E.g., this is the case when invoking contracts that can be dynamically updated. Note indeed that, although the code of a contract cannot be altered once published on the blockchain, with some forethinking it is possible to craft a contract whose components can be updated at his owner's request. At a later time, the owner can link such contract to a malicious component, which e.g. steals the caller's ether.

Generating randomness. The execution of EVM bytecode is deterministic: in the absence of misbehavior, all miners executing a transaction will have the same results. Hence, to simulate non-deterministic choices, many contracts (e.g. lotteries, games, etc.) generate pseudo-random numbers, where the initialization seed is chosen uniquely for all miners.

A common choice is to take for this seed (or for the random number itself) the hash or the timestamp of some block that will appear in the blockchain at a given time in the future. Since all the miners have the same view of the blockchain, at run-time this value will be the same for everyone. Apparently, this is a secure way to generate random numbers, as the content of future blocks is unpredictable. However, since miners control which transactions are put in a block and in which order, a malicious miner could attempt to craft his block so to bias the outcome of the pseudo-random generator. The analysis in [22] on the randomness of the Bitcoin blockchain shows that an attacker, controlling a minority of the mining power of the network, could invest 50 bitcoins to significantly bias the probability distribution of the outcome; more recent research [23] proves that this is also possible with more limited resources.

Alternative solutions to this problem are based on timed commitment protocols [16] [15]. In these protocols, each participant chooses a secret, and then communicates to the others a digest of it, paying a deposit as a guarantee. Later on, participants must either reveal their secrets, or lose their deposits. The pseudo-random number is then computed by combining the secrets of all participants [24] [25]. Also in this case an adversary could bias the outcome by not revealing her secret: however, doing so would result in losing her deposit. The protocol can then set the amount of the deposit so that not revealing the secret is an irrational strategy.

Time constraints. A wide range of applications use time constraints in order to determine which actions are permitted (or mandatory) in the current state. Typically, time constraints are implemented by using block timestamps, which are agreed upon by all the miners.

Contracts can retrieve the timestamp in which the block was mined; all the transactions within a block share the same timestamp. This guarantees the coherence with the state of the contract after the execution, but it may also expose a contract to attacks, since the miner who creates the new block can choose the timestamp with a certain degree of arbitrariness. If a miner holds a stake on a contract, he could gain an advantage by choosing a suitable timestamp for a block he is mining.

Chapter 4

Design and Implementation

4.1 Debug and Inspect

Debug In usual systems the debug procedure is well known and very similar between different languages. In a more abstract view, Debugging is the routine process of locating and removing computer program bugs, errors or abnormalities, which is methodically handled by software programmers via debugging tools.

Debugging checks, detects and corrects errors or bugs to allow proper program operation according to set specifications. Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as debuggers.

Usual and well known debuggers are gdb [26] for **C** and pdb [27] for **python**. Such Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, run the program line by line, and change values in memory. In that process a developer can easily detect the problem at the code and fix it. The debugging process hasn't been changed dramatically over the years because the architecture of the systems and programming languages is more or less the same.

Inspect In blockchain the debugging process is very different and probably more difficult. The program developed in a blochchain application is basically the smart contract which is written in **solidity**. The first step to debug and check if the smart contract works as expected is to use the remix webpage [28] which is the official site to compile and run step by step your smart contract before deploying it to the blockchain. After some manual tests you can deploy it but the difficult part is after the deployment. When a smart contract is deployed and you start to use it for your system, you cannot change it. It is immutable. Thus, if a bug comes up you may need to deploy a new smart contract and pause your system for a while. Nevertheless, such problems are usually solved but the main problem is understanding how the blockchain system works [29] [30].

Lets say that we have a UI that enables the users of our system to make actions on our system and essentially interact with the smart contract, which is the basic logic behind the system. To submit a transaction and interact with the smart contract, there are some requirements to be fulfilled. We will state those requirements briefly below.

- An account should exist which will be the sender of the transaction. If an account doesn't exist the transaction will fail and the desired action will not be accomplished.
- The address of the recipient, in our case the address of the smart contract, must be valid. If not the gas from the transaction will be lost and the action obviously will not be executed.
- The input value, which refers to the money an account wants to send to another, must be lower or equal to the remaining amount.
- The gas parameter, should have higher value from the expected cost of the transaction, which is the most times unknown. A lower gas value will roll back the execution of the code and the gas will be spent, even though the code didn't execute.
- The **gasPrice** argument should be empty or higher than the current value of the blockchain. If it is lower the transaction will be rejected.
- The input data, which refers to the address of the function and the arguments for that function must be valid.

Those are the basic requirements so that a transaction is valid. As mentioned previously there are pretty much a lot vulnerabilities in smart contracts and also in system that is based on the blockchain. The basic reason of that, is the plethora of the different subsystems you need to use to create your blockchain application.

The basic subsystems that must function correctly are listed below.

- The smart contract.
- The transactions sent from user interface of the application.
- The fine tuning of the transactions to the current values of the blockchain and to the expected cost of each action.

To accomplish all the previous a more general tool than a debugger is needed, which will help to monitor and detect such abnormalities in a system based on blockchain. The hardware requirements to accomplish this task are a few. The basic component obviously is a mid range computer with a satisfying amount of ram, (about 8GB) and the proper tool. Probably the best words to describe such tools are, **Inspector - Visualizer**.

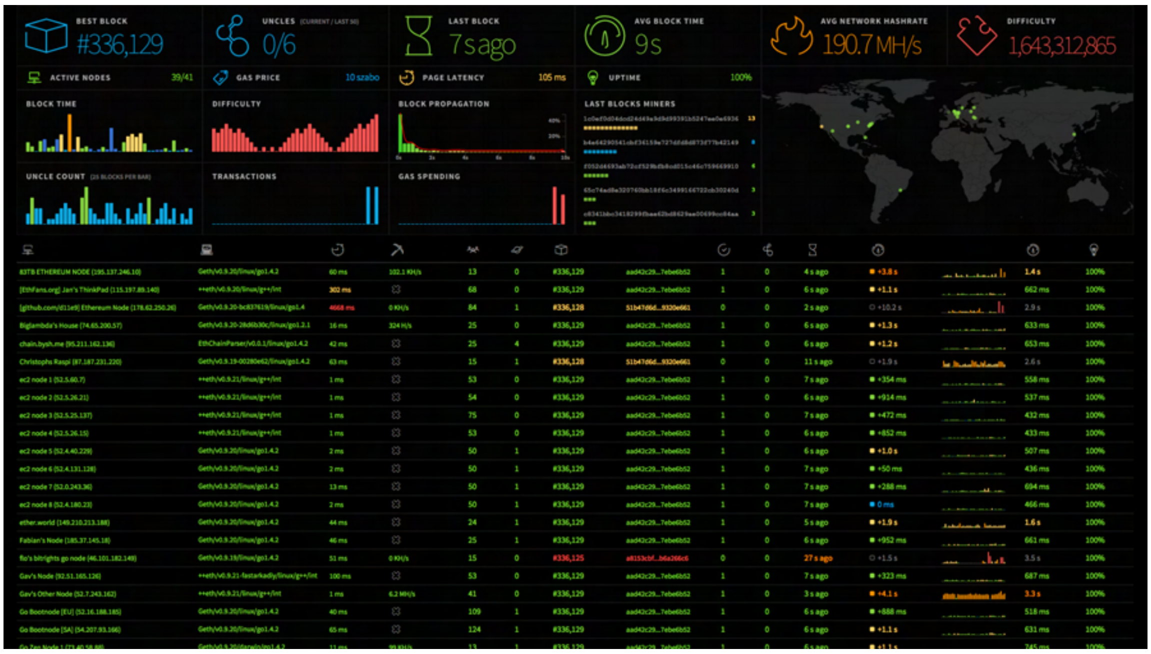
Nevertheless, anyone who wants to create an application/system that is based on the blockchain must test his system locally by creating a private Ethereum network. The setup of such a system is not very difficult but it is time consuming. The basic requirements are personal computers which will act as the miners - nodes of the network and provide the functionality of the public Ethereum network. The basic reason that we need to create a private network instead of testing our system in the public one is the money. If you want to run your application in the public network you must pay Ethers, which can be bought with real money. Thus, a private network is needed to create a simulation of the application with no cost. The number of tools that can inspect the public network is really big, but the ones that can be used to monitor a private network are really a few [29] [30].

4.2 Previous Work

To inspect a system based on blockchain, we need a tool that can firstly connect easily to any network, private or public, and explore the blockchain. This means, that a user can get a desired block and check all the informations inside that block, like the size of the block, from which account it was mined, the total gas that this block can contain, the difficulty of the block, the number of the transactions and many more. Each information may or may not be important depending on the nature of the system that was implemented. Another important feature is to check the transactions that were mined in a specified block and their arguments. This basic functionality is important if you suspect where the possible error occurred but it doesn't help if you are trying to find the error.

There are several Blockchain - Ethereum explorers that provide this basic functionality and we will analyze some of them briefly below.

Ethereum Network Stats [31] The ethstats tool is one of the most well known tools to monitor the activity of the public Ethereum blockchain network. It provides several metrics and charts to represent the state of the network.



83TB ETHEREUM NODE (195.137.246.10)	Geth/v0.9.20/linux/go1.4.2	60 ms	102.1 KH/s	13	0	#336,129	aad42c29...7ebe6b52	1	0	4 s ago	+3.8 s	1.4 s	100%
[EthFans.org] Jan's ThinkPad (115.197.89.140)	++eth/v0.9.20/linux/g++/int	302 ms	⊠	68	0	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+1.1 s	662 ms	100%
[github.com/d11e9] Ethereum Node (178.62.250.26)	Geth/v0.9.20-bc837619/linux/go1.4	4668 ms	0 KH/s	84	1	#336,128	51b47d6d...9320e661	0	0	2 s ago	+10.2 s	2.9 s	100%
Biglambda's House (74.65.200.57)	Geth/v0.9.20-28d6b30c/linux/go1.2.1	16 ms	324 H/s	25	0	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+1.3 s	633 ms	100%
chain.bysh.me (95.211.162.136)	EthChainParser/v0.0.1/linux/go1.4.2	42 ms	⊠	25	4	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+1.2 s	653 ms	100%
Christophs Raspi (87.187.231.220)	Geth/v0.9.19-00280e62/linux/go1.4.2	63 ms	⊠	15	1	#336,128	51b47d6d...9320e661	0	0	11 s ago	+1.9 s	2.6 s	100%
ec2 node 1 (52.5.60.7)	++eth/v0.9.21/linux/g++/int	1 ms	⊠	53	0	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	+354 ms	558 ms	100%
ec2 node 2 (52.5.26.21)	++eth/v0.9.21/linux/g++/int	1 ms	⊠	54	0	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+914 ms	537 ms	100%
ec2 node 3 (52.5.25.137)	++eth/v0.9.21/linux/g++/int	1 ms	⊠	75	0	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	+472 ms	432 ms	100%
ec2 node 4 (52.5.26.15)	++eth/v0.9.21/linux/g++/int	1 ms	⊠	53	0	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+852 ms	433 ms	100%
ec2 node 5 (52.4.40.229)	Geth/v0.9.20/linux/go1.4.2	2 ms	⊠	50	1	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+1.0 s	507 ms	100%
ec2 node 6 (52.4.131.128)	Geth/v0.9.20/linux/go1.4.2	2 ms	⊠	50	1	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	+50 ms	436 ms	100%
ec2 node 7 (52.0.243.36)	Geth/v0.9.20/linux/go1.4.2	13 ms	⊠	50	1	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	+288 ms	694 ms	100%
ec2 node 8 (52.4.180.23)	Geth/v0.9.20/linux/go1.4.2	2 ms	⊠	50	1	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	0 ms	466 ms	100%
ether.world (149.210.213.188)	Geth/v0.9.20/linux/go1.4.2	44 ms	⊠	24	1	#336,129	aad42c29...7ebe6b52	1	0	5 s ago	+1.9 s	1.6 s	100%
Fabian's Node (185.37.145.18)	Geth/v0.9.20/linux/go1.4.2	46 ms	⊠	25	1	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+952 ms	661 ms	100%
flo's bitrights go node (46.101.182.149)	Geth/v0.9.19/linux/go1.4.2	51 ms	0 KH/s	15	0	#336,125	a8153cbf...b6a266c6	0	0	27 s ago	+1.5 s	3.5 s	100%
Gav's Node (92.51.165.126)	++eth/v0.9.21-fastarkadiy/linux/g++/int	100 ms	⊠	53	0	#336,129	aad42c29...7ebe6b52	1	0	7 s ago	+323 ms	687 ms	100%
Gav's Other Node (52.7.243.162)	++eth/v0.9.21/linux/g++/int	1 ms	6.2 MH/s	41	0	#336,129	aad42c29...7ebe6b52	1	0	3 s ago	+4.1 s	3.3 s	100%
Go Bootnode [EU] (52.16.188.185)	Geth/v0.9.20/linux/go1.4.2	40 ms	⊠	109	1	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+888 ms	518 ms	100%
Go Bootnode [SA] (54.207.93.166)	Geth/v0.9.20/linux/go1.4.2	65 ms	⊠	124	1	#336,129	aad42c29...7ebe6b52	1	0	6 s ago	+1.1 s	631 ms	100%

FIGURE 4.1: Ethstats.

Source: http://ethstats.net

The basic charts are live for a window of 20 instances and can be seen on the previous image. We enumerate and analyze some of them below.

- Block Time chart, which represents the time that was needed to generate each block.
- The difficulty of each block.
- The Uncle Count, which represents the number of the side chains (uncles) for each block.

- The number of transactions per block, which is a good metric to watch the load of the network
- The gas spend for each block.

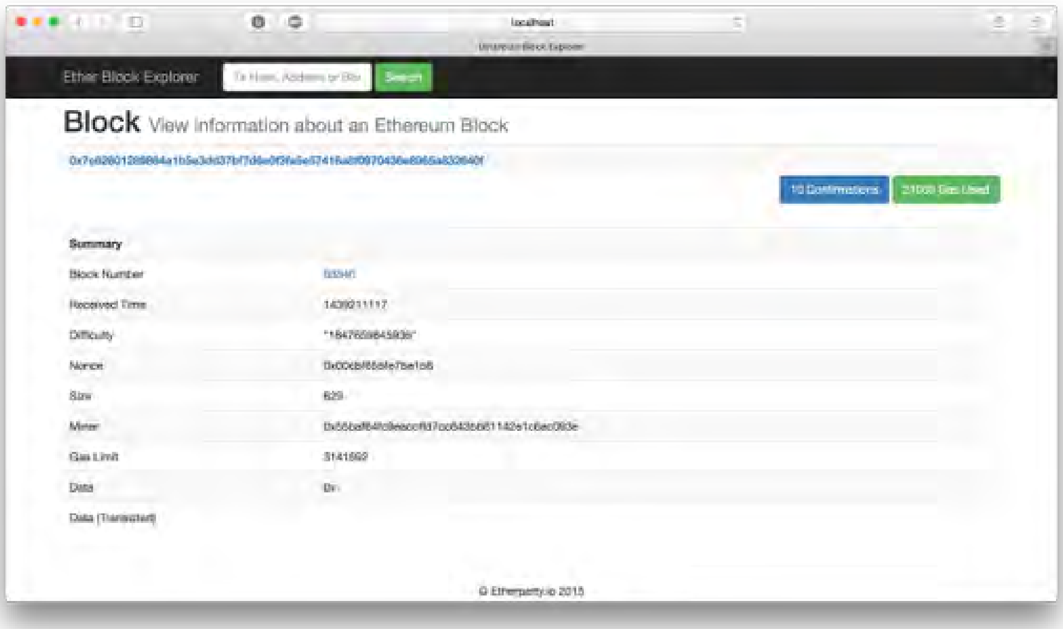
More metrics are provided such as the gas price, which determines the ether that each transaction costs, the average time that a block is generated and many more.

In that tool a user cannot explore the blocks or the transactions of the blockchain but can only watch a representation of the network. Also, there is no support for a smart contract or a UI to interact and search specific information that exists in blockchain. This tool is open source and uploaded at GitHub [32]. Nevertheless, there is not an easy way to contribute to this project because the documentation is insufficient.

As stated from the creators this tool can be used for a private network but the setup is really hard and repulsive. Firstly, you must create a public IP for your local node, which the most times cannot be done or you don't have the time to do so and secondly you must make a skype call with the company to provide you a token so that you can use their tool for your application.

Thus, this tool cannot be easily used for a private network and probably a developer will not add it to his possible options.

EthExplorer [33] This Eth explorer is one of the first explorers for the Ethereum network and functions really good. It has a simple UI and you can get any block you want and inspect its informations and transactions.



Block View information about an Ethereum Block

0x7e626012[illegible]

Summary	
Block Number	[illegible]
Received Time	1439211117
Difficulty	"1847609845936"
Nonce	0x00cbf[illegible]
Size	629
Miner	0x58baf[illegible]
Gas Limit	3141592
Data	Ðr
Data (Translated)	

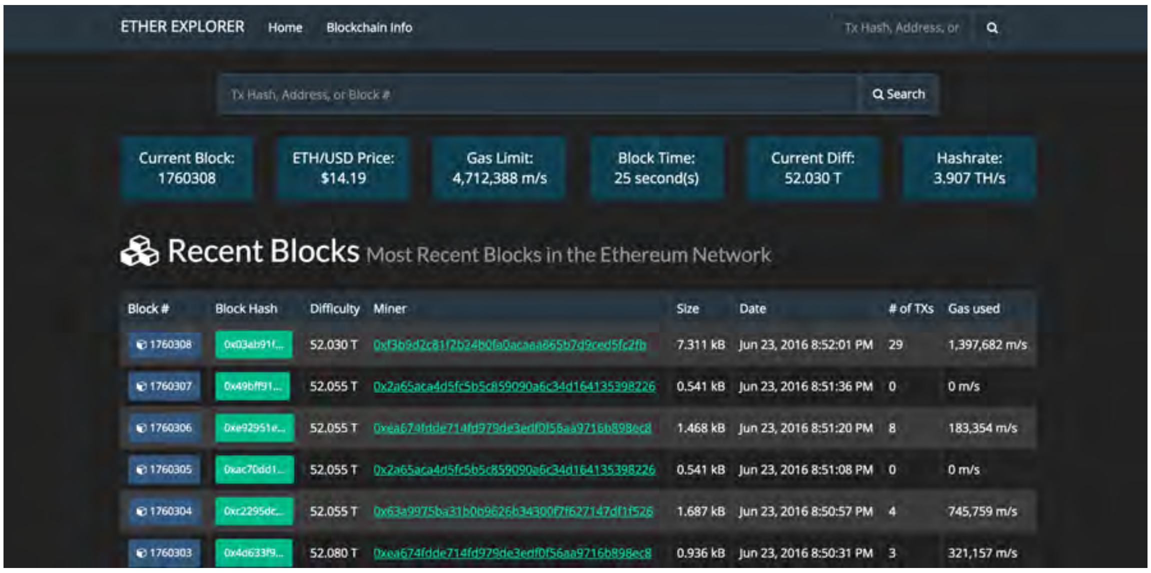
© Etherparty.io 2015

FIGURE 4.2: Eth Explorer.

Source: https://github.com/etherparty/explorer

The basic advantage of this tool is that it is really simple and easy to use but there are also some disadvantages. At first, the tool syncs with the last forty blocks which is a really small number if we consider the size of blockchain. Secondly and most importantly, this tool doesn't provide any scenarios to compare or get a picture of the state of the network. Also, there is no contract support. Lastly, this project is outdated and the last update was nine months ago. Provided that the project is open source it should have a very detailed documentation so that anyone who wants to contribute could start coding easier. Nevertheless, it provides an easy setup and connection with a private network. Thus, the possibility to be used from a developer is high.

Ethereum Block Explorer V2 [34] This project is a fork - replication of the previous explorer with a refreshed UI and some extra features.



Recent Blocks Most Recent Blocks in the Ethereum Network

Block #	Block Hash	Difficulty	Miner	Size	Date	# of TXs	Gas used
1760308	0x03ab91f...	52.030 T	0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb	7.311 kB	Jun 23, 2016 8:52:01 PM	29	1,397,682 m/s
1760307	0x49bff91...	52.055 T	0x2a65aca4d5fc5b5c859090a6c34d164135398226	0.541 kB	Jun 23, 2016 8:51:36 PM	0	0 m/s
1760306	0xe92951e...	52.055 T	0xea674fdde714fd979de3edf0f56aa9716b898ec8	1.468 kB	Jun 23, 2016 8:51:20 PM	8	183,354 m/s
1760305	0xac70dd1...	52.055 T	0x2a65aca4d5fc5b5c859090a6c34d164135398226	0.541 kB	Jun 23, 2016 8:51:08 PM	0	0 m/s
1760304	0xc2295dc...	52.055 T	0x63a9975ba31b0b9626b34300f7f627147df1f526	1.687 kB	Jun 23, 2016 8:50:57 PM	4	745,759 m/s
1760303	0x4d633f9...	52.080 T	0xea674fdde714fd979de3edf0f56aa9716b898ec8	0.936 kB	Jun 23, 2016 8:50:31 PM	3	321,157 m/s

FIGURE 4.3: ETHExplorer V2.

Source: `https://github.com/etherparty/explorer`

This explorer adopts better the sense of an open source project, by providing better documentation, and this is obvious by the usual and relatively large activity of the developers. Even though this explorer is a fork of the previous one, it has a much bigger impact on the open source community, which is obvious from the much bigger number of stars it has. Lastly, the difficulty of the setup and connection to a private network is really low like its fork.

Etherchain Light [35] Etherchain Light is an Ethereum blockchain explorer built with NodeJS, Express and Parity. It does not require an external database and retrieves all information on the fly from a backend Ethereum node.

While there are several excellent Ethereum blockchain explorers available (etherscan, ether.camp and etherchain) they operate on a fixed subset of Ethereum networks, usually the mainnet and testnet. Currently there are no network agnostic blockchain explorers available. If you want to develop Dapps on a private testnet or would like to launch a private / consortium network, Etherchain Light will allow you to quickly explore such chains.

This tool is the probably the best through the rest but it is using the parity Ethereum Node which has less development from the Geth Ethereum Node. It is also not so easy to setup but it uses docker which is a very good advantage.

ethereum-blockchain-explorer [36] Another much simpler tool is the ethereum blockchain explorer. It has no documentation or instructions and we present it here just for reference. It is very similar with the previous explorers but probably at the bottom of the list.

Tools that work only with the public network There are a lot of really good tools that work with the public Ethereum network. These tools are usually whole companies that invest on their products for different reasons. The most famous of them is Etherscan [37]. It is a tool that has the whole Ethereum network saved on a database and provides different useful functions, such as the previous tools, but in a more enterprise way. The basic use of Etherscan is to find all the transactions that are related with an account, through the blockchain and possibly tokens of smart contracts acquired by that account. Very similar with that tool is Etherchain [38] which we will not analyze further. Summarizing, it would be really helpful if there was a tool so much delicate as the ones that exist and work with the Ethereum public network.

QuickBlocks [39] Expect from tools, there also a lot of libraries and one of the most well known is QuickBlocks.

A software system, QuickBlocks, is described that provides user focused, speed optimized, customizable per smart contract data from any blockchain, including public, consortia,and private chains. Through a collection of software libraries, applications, and automatically generated source code, the system improves the quality and accessibility of blockchain data to programmers and end users. Given this improved accessibility, many previously unanticipated functionalities, such as fast delivery of smart contract specific JSON data from RPC, detailed gas usage analysis, live debugging and stress testing from previously recorded blockchain interactions, smart contract control panels, and user local, data rich, fully decentralized desktop and mobile applications become possible.

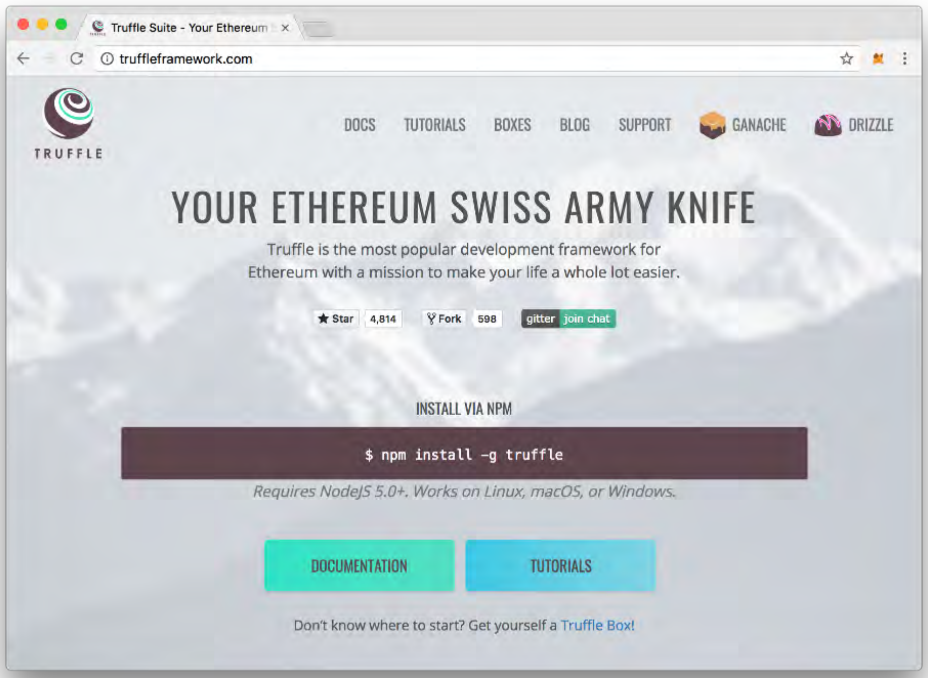
This library is implemented in C++ and has a lot of useful functions which can be used in different simulations on an Ethereum private network. The implementation in C++ may be a drawback for some developers but it also may be really useful in other cases.

smart-contract-watch [40] Smart contract watch is a tool - library created to monitor a smart contract. It can monitor smart contracts activity and interactions based on generated transactions and events. For example, it can be used on a local blockchain explorer that runs locally on your server or machine ,or as an investigation tool that scrapes the blockchain in search for a specific query. This is done by sending requests to an Ethereum node via JSON RPC calls. The fastest way to use this tool is by the CLI after installing it from the official repository. In the documentation there are no instructions on how to use this tool as a library but it should not be difficult because it is basically implemented in JavaScript [41].

Truffle [42] [43] Among the previous tools-libraries there is a bigger project Truffle. This project is much different than the previous listed and we should address it as a framework because of the extensive functionality it provides.

Truffle is a development environment, testing framework and asset pipeline for Ethereum, aiming to make life as an Ethereum developer easier. With Truffle, you get:

- Built-in smart contract compilation, linking, deployment and binary management.
- Automated contract testing with Mocha and Chai.
- Configurable build pipeline with support for custom build processes.
- Scriptable deployment & migrations framework.
- Network management for deploying to many public & private networks.
- Interactive console for direct contract communication.
- Instant rebuilding of assets during development
- External script runner that executes scripts within a Truffle environment.



YOUR ETHEREUM SWISS ARMY KNIFE

Truffle is the most popular development framework for Ethereum with a mission to make your life a whole lot easier.

```
$ npm install -g truffle
```

Requires NodeJS 5.0+. Works on Linux, macOS, or Windows.

Don't know where to start? Get yourself a Truffle Box!

FIGURE 4.4: Truffle Framework.

Source: `http://truffleframework.com/`

The basic target group of Truffle is anyone who wants to dive into Ethereum development and needs a framework so they can better organize their DApp development assets and not have to worry about manually setting up a test environment.

4.3 System Design (Inspector - Visualizer)

4.3.1 Basic Concept

This tool was created because there were obvious scenarios, that a developer of a Dapp could use to debug his application and were not implemented by none so far. Initially, the basic concept of the tool was based on the already existed tools-explorers, which were mentioned previously. A simple way to explore the blocks and their info is obviously necessary to debug a Dapp.

Before diving in the implementation a basic requirement must be met, which is a way to connect and get data from an Ethereum node efficiently. The library web3js [44] implemented by the Ethereum organization is the key component that enables the communication with an Ethereum node via http requests. This library provides a lot of functions and really good documentation on how to use them. The library itself is really good but an Ethereum node is not built to answer in a lot requests. By experimenting with this library it was found out that if multiple requests were made at the Ethereum node some of them would return empty. This was an indication that the node hadn't a built-in queue to serve external requests. As a result, if a user wanted to get a big range of blocks and save them locally, he couldn't retrieve some of them. A simple way to solve this problem is to request small amounts of data each time and sync them. Obviously, this is possible "by hand" but an efficient way was implemented to get the blocks and the transactions receipts of the blockchain without any loss of data. As always, the basic component on syncing http requests, is chaining "promises" [45].

Provided that the first two requirements were met, we could continue with the architecture and the implementation. While looking at the already existed tools about exploring and debugging a Dapp, it was obvious enough that none of them could be installed really easy, fast and straight-forward. Also, most of them could not sync with a range of data and then extract any available info out of them. And lastly there wasn't a good enough and simple implementation so that, any developer could contribute on that tool.

Thus, to achieve and create a useful tool those faults of the previous implementations should be corrected and more features must be added. The first to do, is to create an open source tool in a famous framework, based on web to take advantage of its features and implement the project with a straight forward and easy to understand way. The framework that was selected is NodeJs, which probably is the most well known-used server-backend for web. This framework is implemented in Javascript which is one of the most used programming language and thus it could be easy to start coding. Also, the contribution of any developer could be much easier because Javascript is so widespread. Pure Javascript was used and nothing more such as typescript, to keep the implementation simple and easy to extend from anyone.

The requirements to use this tool are minimal. After cloning the project from GitHub, with a command all the libraries that are needed will be installed. The basic requirement to function this tool, is an Ethereum node. The tool can connect to any address which can be changed in a global var inside the basic configuration. Obviously, a local node is always better for faster responses and easier configuration. Thus, this tool connects on the Ethereum node and uses the web3js library to get any information from the blockchain. Currently there is not a database setup and all the data are being saved at RAM. This is obviously not efficient for big experiments but it enables a fast startup without spending time on installing the right database.

The basic functions to implement are allowing the user to get a block and explore it. He can get many information such as the miner, the transactions that contains, the total gas that was used and many more. From that point he can dig in the transactions and check their details. The basic information of a transaction is the sender, the receiver and the input data. The receiver could be an account or a smart contract and the data can contain the address of a function and the arguments that will be the input.

Thus, after solving those concerns we were ready to implement a lot of scenarios to provide an inspector-visualizer-debugger for the developer.

4.3.2 Basic Functions

Real Time Information Inspecting a network such as blockchain is really difficult and confusing. For that reason the tool gets some basic information from the blockchain and represents it in real time. The real time variables that can be shown in the home screen are:

- Difficulty. The difficulty of the last block generated
- Gas Limit. The maximum gas a block can contain.
- Gas Used. The gas that is contained at the last block.
- Gas Price. The price of the gas that represents a function to the value of ether.
- The last block number.

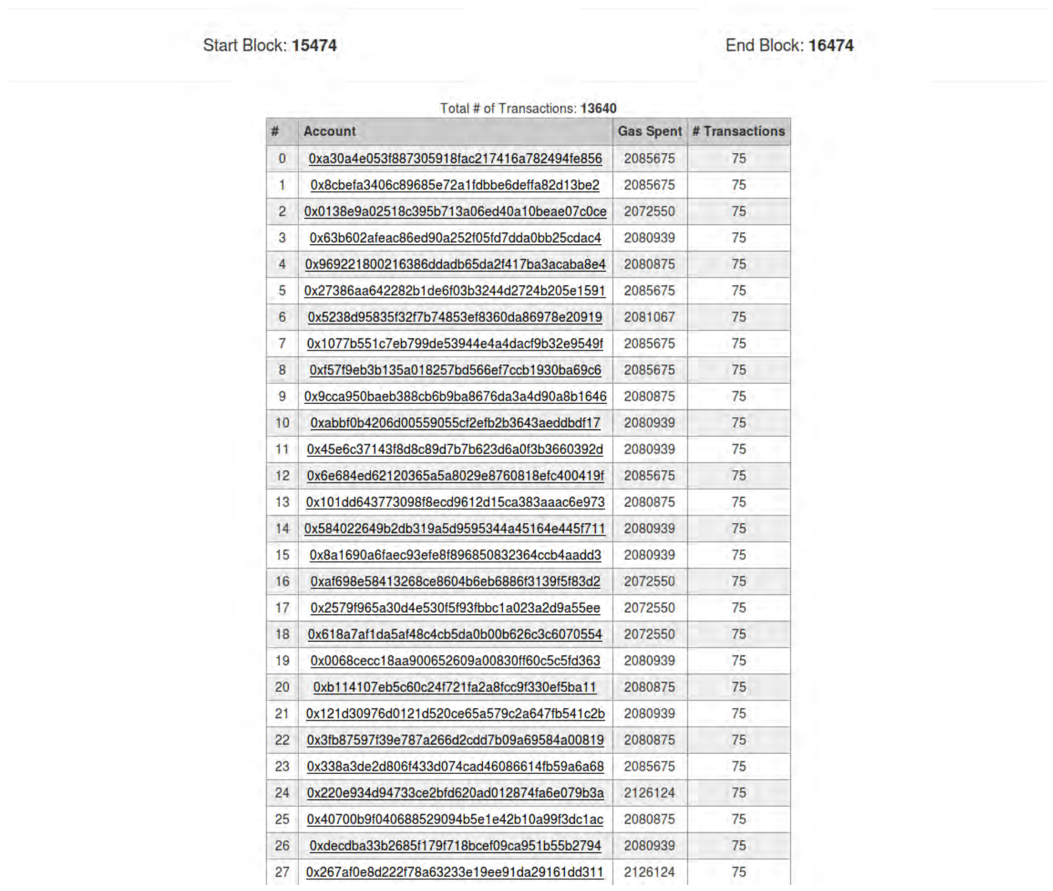
Except from that useful metrics, which can give to the developer a picture at the state of the network, there are 4 real time charts, which show the difficulty, gas limit, number of transactions, gas spend for each block in a range of 10 blocks.

Basic Scenario The first scenario that was implemented was based on a simple logic to inspect the transactions in the network. The basic info that is needed to inspect the correctness of an experiment firstly is the transactions. At first, the user must specify the range of the blocks, start - end, to get the results. If not specified the Inspector will return the results for the last 1000 blocks.

It is useful enough in a blockchain application to get feedback for a small range of blocks in a few seconds. In that manner you can find a bug quickly in an experiment and monitor small parts of it. For that reason it is recommended to specify a small fractions of blocks. Nevertheless, the "Inspector" was implemented to get results for a big range of blocks. The restrictions of the web3.js library and of the Ethereum node architecture made this task really hard and it will be analyzed in a section below.

So, after getting the transactions and obviously the blocks that contain the transactions we have all the needed information to create a simple and straightforward table. The table will contain rows of the following object: (account hash, gas spent on transactions, number of transactions) and the first column will enumerate this objects for better monitoring. It is a really simple table but it contains really valuable information. With that table is easy enough to understand if an experiment is working correctly and that all the sent transactions are mined into blocks. When designing your experiment you will obviously know the number of transactions that each account will sent. With this basic scenario you will know if the transactions are actually sent or not and it will be easy enough to find that something is going wrong. Furthermore, you can see the total number of the transactions through the specified range of blocks which is also a good metric to understand the state of the experiment.

A simple example of an experiment that was one of the incentives to create this tool, consisted from 100 accounts that each one was sending one transaction per 10 blocks. Thus, in a range of 1000 blocks we should expect 10000 transactions. If this number is different then something is wrong and usually it is really hard to find, but it is better to know that something is not working as expected than not.


Start Block: **15474**

End Block: **16474**

Total # of Transactions: **13640**

#	Account	Gas Spent	# Transactions
0	0xa30a4e053f887305918fac217416a782494fe856	2085675	75
1	0x8cbefa3406c89685e72a1fdbbe6deffa82d13be2	2085675	75
2	0x0138e9a02518c395b713a06ed40a10beae07c0ce	2072550	75
3	0x63b602afeac86ed90a252f05fd7dda0bb25cdac4	2080939	75
4	0x969221800216386ddadb65da2f417ba3acaba8e4	2080875	75
5	0x27386aa642282b1de6f03b3244d2724b205e1591	2085675	75
6	0x5238d95835f32f7b74853ef8360da86978e20919	2081067	75
7	0x1077b551c7eb799de53944e4a4dacf9b32e9549f	2085675	75
8	0xf57f9eb3b135a018257bd566ef7ccb1930ba69c6	2085675	75
9	0x9cca950baeb388cb6b9ba8676da3a4d90a8b1646	2080875	75
10	0xabbf0b4206d00559055cf2efb2b3643aeddbdf17	2080939	75
11	0x45e6c37143f8d8c89d7b7b623d6a0f3b3660392d	2080939	75
12	0x6e684ed62120365a5a8029e8760818efc400419f	2085675	75
13	0x101dd643773098f8ecd9612d15ca383aaac6e973	2080875	75
14	0x584022649b2db319a5d9595344a45164e445f711	2080939	75
15	0x8a1690a6faec93efe8f896850832364ccb4aadd3	2080939	75
16	0xaf698e58413268ce8604b6eb6886f3139f5f83d2	2072550	75
17	0x2579f965a30d4e530f5f93fbbc1a023a2d9a55ee	2072550	75
18	0x618a7af1da5af48c4cb5da0b00b626c3c6070554	2072550	75
19	0x0068cecc18aa900652609a00830ff60c5c5fd363	2080939	75
20	0xb114107eb5c60c24f721fa2a8fcc9f330ef5ba11	2080875	75
21	0x121d30976d0121d520ce65a579c2a647fb541c2b	2080939	75
22	0x3fb87597f39e787a266d2cdd7b09a69584a00819	2080875	75
23	0x338a3de2d806f433d074cad46086614fb59a6a68	2085675	75
24	0x220e934d94733ce2bfd620ad012874fa6e079b3a	2126124	75
25	0x40700b9f040688529094b5e1e42b10a99f3dc1ac	2080875	75
26	0xdecdba33b2685f179f718bcef09ca951b55b2794	2080939	75
27	0x267af0e8d222f78a63233e19ee91da29161dd311	2126124	75

FIGURE 4.5: Basic Scenario

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

As shown in the picture above, each account is a hyperlink that when pressed will redirect to another scenario, which shows the transactions of a specified account through a range of blocks. This scenario will be analyzed later more exhaustively.

Another usual problem that is really disturbing exists when interacting with smart contracts. To interact and change the state, the values of variables, of smart contracts the only way is to send transactions to their addresses and call a function by passing the hash and the input of the function. This seems really straightforward and it probably is but only if there was no blockchain. The blockchain architecture comes with some requirements to ensure stability, security and other important things, but those requirements are a lot. It is really obvious that at the begging of an experiment and if the developer has no previous experience, a lot of the requirements will not be fulfilled. This will result unwanted and unexpected errors. The most common and disturbing problem that can occur is from sending a "gas-less" transaction. As previously mentioned a transaction must have gas as argument when sent to be mined. A transaction that will execute some code in a smart contract will need more or less gas from the default value, which is 90000, depending on the complexity and length of the code. Thus, if you don't specify the gas you will send with the transaction the default value will be used. When you send the transaction it will be mined, but after that it will try to execute the code in the smart contract and there will probably occur something that cannot be easily detected. If the gas send was less than the required of the function that was called, which cannot be calculated if there are for-loops, then the code will execute until it spends all the available gas that was sent and then will roll back the execution of the code because of the lack of gas. This will have as a result a transaction that was mined but didn't change the state of a variable as expected. At first by only looking the transactions you will be surprised to see that the state of the contract wasn't changed even if the needed transactions exist.

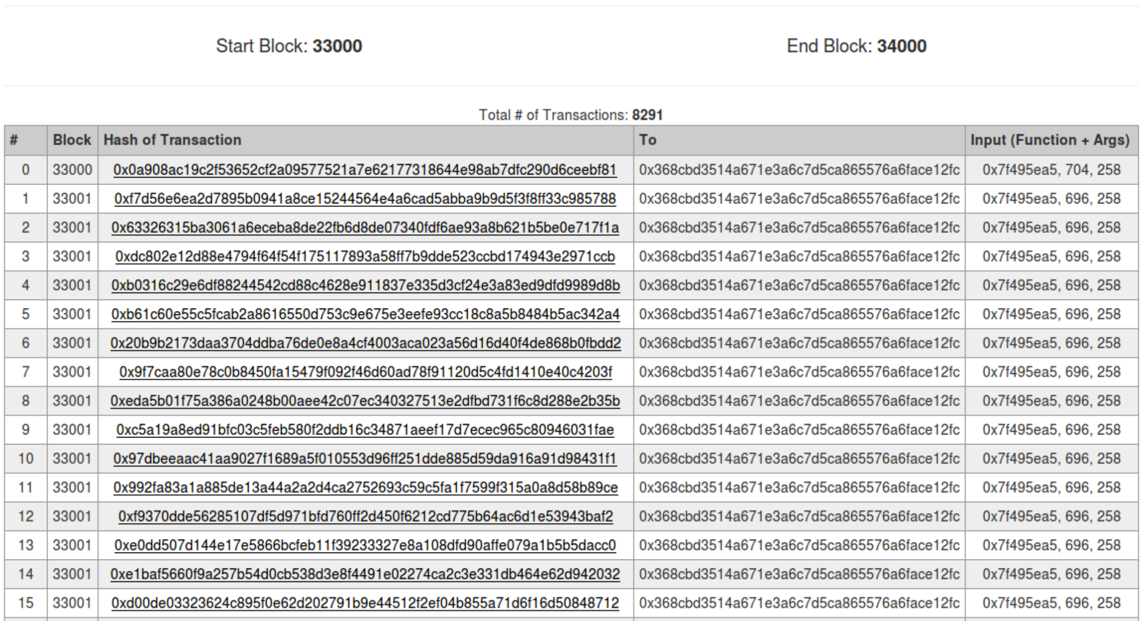
To solve this problem, some of the solutions was to search in the documentation which wasn't any help and the obvious one was testing. All this to understand how we can identify a roll-back transaction [11]. By checking the information of a mined transaction it was odd in some of the transactions that the sent gas was the same as the spend gas. And this was the key to identify such problematic transactions. Thus, when getting all the transactions in this basic scenario if there are any rolled-back transaction a new table with red font will be displayed with the needed information and hyperlinks to identify those transactions. This scenario identifies the following vulnerabilities 17 18 19 21.

This really simple to solve problem, will take a lot of time to find by hand and the basic cause of that is a wrong argument value. Ideally we should know the exact cost of a function in a smart contract but this is not achievable with the current architecture of smart contracts and Solidity. A simple solution to that problem is to send a much higher value of gas from which the remaining gas will be returned to the user.

This basic scenario solves some relatively easy problems, but hard for the human eye to detect, and its simplicity helps to solve problems much faster and more efficiently.

Get Transactions A much simpler and obvious function that any tool of this type should have is a transactions explorer. This function generates a table that contains in each row, a hyperlink of the transaction hash, the block in which the transaction was mined, the receiver address and the input variable, which may contain a function address of a contract and its arguments. A demonstration of this function can be seen in the following figure 4.6.

Sync with Blocks A user can specify a range of blocks to sync, so that he can use that information on the different provided scenarios. When a provided function is called, the system checks if the range of blocks is already synced or not and adapts appropriately.


Start Block: **33000**

End Block: **34000**

Total # of Transactions: **8291**

#	Block	Hash of Transaction	To	Input (Function + Args)
0	33000	0x0a908ac19c2f53652cf2a09577521a7e62177318644e98ab7dfc290d6ceebf81	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 704, 258
1	33001	0xf7d56e6ea2d7895b0941a8ce15244564e4a6cad5abba9b9d5f3f8ff33c985788	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
2	33001	0x63326315ba3061a6eceba8de22fb6d8de07340fdf6ae93a8b621b5be0e717f1a	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
3	33001	0xdc802e12d88e4794f64f54f175117893a58ff7b9dde523ccbd174943e2971ccb	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
4	33001	0xb0316c29e6df88244542cd88c4628e911837e335d3cf24e3a83ed9dfd9989d8b	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
5	33001	0xb61c60e55c5fcab2a8616550d753c9e675e3eefe93cc18c8a5b8484b5ac342a4	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
6	33001	0x20b9b2173daa3704ddba76de0e8a4cf4003aca023a56d16d40f4de868b0fbdd2	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
7	33001	0x9f7caa80e78c0b8450fa15479f092f46d60ad78f91120d5c4fd1410e40c4203f	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
8	33001	0xeda5b01f75a386a0248b00aee42c07ec340327513e2dfbd731f6c8d288e2b35b	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
9	33001	0xc5a19a8ed91bfc03c5feb580f2ddb16c34871aeef17d7ecec965c80946031fae	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
10	33001	0x97dbeeaac41aa9027f1689a5f010553d96ff251dde885d59da916a91d98431f1	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
11	33001	0x992fa83a1a885de13a44a2a2d4ca2752693c59c5fa1f7599f315a0a8d58b89ce	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
12	33001	0xf9370dde56285107df5d971bfd760ff2d450f6212cd775b64ac6d1e53943baf2	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
13	33001	0xe0dd507d144e17e5866bcfeb11f39233327e8a108dfd90affe079a1b5b5dacc0	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
14	33001	0xe1baf5660f9a257b54d0cb538d3e8f4491e02274ca2c3e331db464e62d942032	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258
15	33001	0xd00de03323624c895f0e62d202791b9e44512f2ef04b855a71d6f16d50848712	0x368cbd3514a671e3a6c7d5ca865576a6face12fc	0x7f495ea5, 696, 258

FIGURE 4.6: Get Transactions

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

4.3.3 Charts

Get Blocks Info A blockchain block has a lot of metrics which vary from block to block and a developer can extract precious information just by inspecting them and compare them through time. The best way to do that is a chart with multiple traces which can provide the values of the various metrics through a specified range of blocks. The metrics that were chosen, after taking into consideration the value of the information that each one has, are:

- Gas limit
- Gas sent
- Gas spent
- Block size

All the previous metrics are related each other and sometimes it is essential to compare them. From the **gas spent** we can understand the cost of an experiment which is compulsory during the development of a blockchain system.

Get Gas Spent of Account Another useful chart demonstrates the gas that an account spent through a range of blocks. With that chart the developer can easily monitor and inspect the activity of any account and check if it is the expected that he designed. Picture 4.7 demonstrates the previous description.

This chart has another line that shows the gas limit through the specified range of blocks. As stated previously the gas limit, is the upper limit of gas that can be contained in a block.

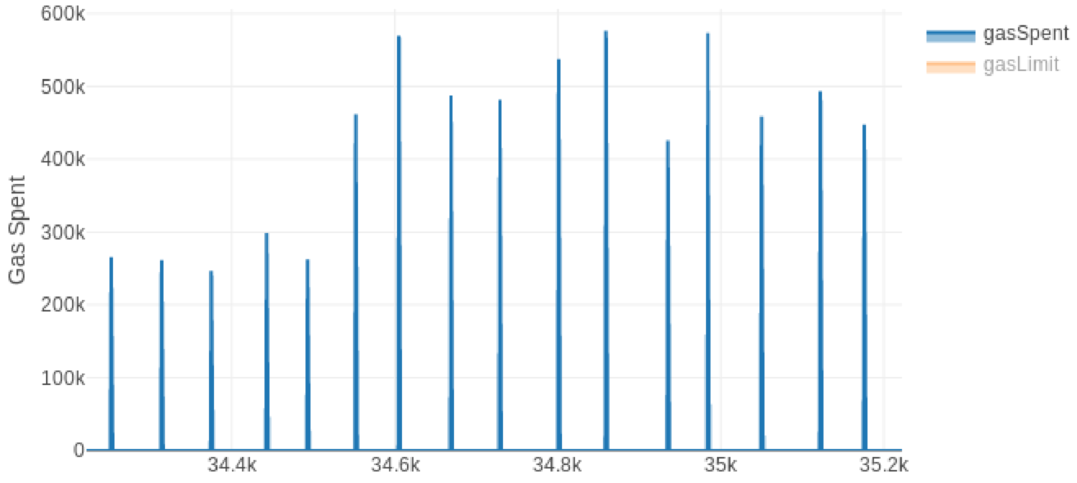
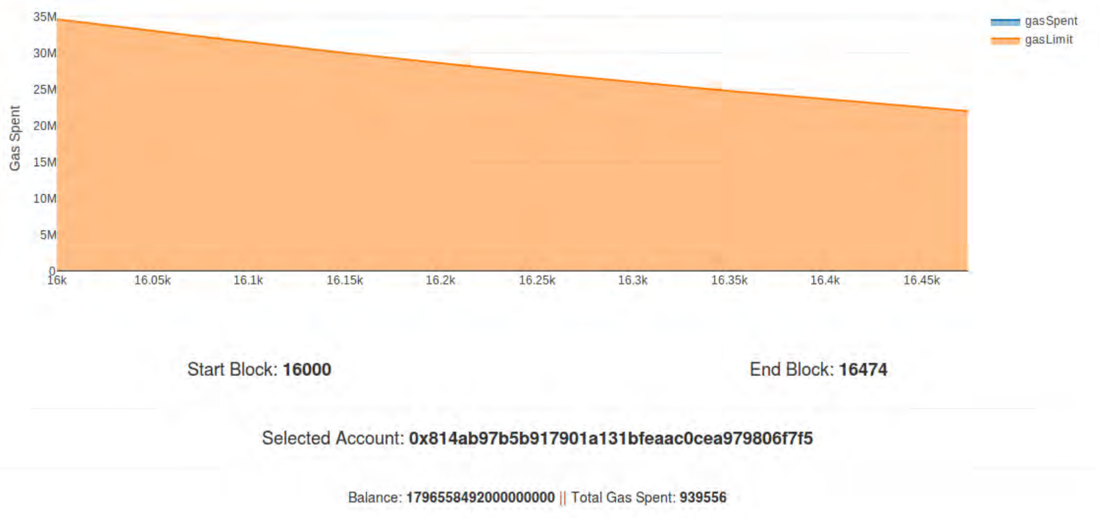


FIGURE 4.7: Gas Spent of account

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

Thus, if we want all the transactions to be valid and pass to the blockchain we need to set this argument high enough for our needs. In the chart, if the gas spent from the account is near the gas limit then it is possible that some of the transactions were aborted. In that scenario, if a call to function costs more than the gas limit then this transaction will never be mined. Thus, this chart will demonstrate the course of the gas limit which will help the developer to adjust his experiment and set the gas limit according. Picture 4.8 demonstrates the previous description.



Start Block: **16000**

End Block: **16474**

Selected Account: **0x814ab97b5b917901a131bfeaac0cea979806f7f5**

Balance: **1796558492000000000** || Total Gas Spent: **939556**

FIGURE 4.8: Gas Limit

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

Get Balance of Account Per Block An easy way to show the profit of a miner is to get his balance over a number of blocks. This functionality is implemented and represented in a chart that shows the balance of the specified through a number of blocks. Easy enough anyone can understand from the chart if the miner benefits from the network if we consider that he spends gas to send his transactions, and what is the coefficient of his benefit. Picture 4.9 demonstrates the previous description.

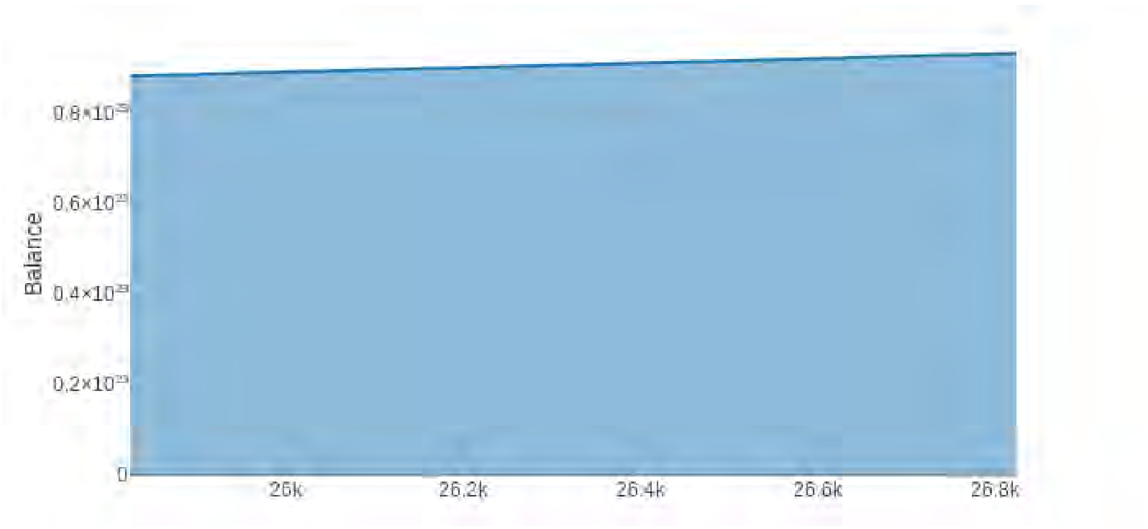


FIGURE 4.9: Balance of account per block

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

Get Transactions Per Blocks A simple chart to monitor the traffic of a blockchain network contains the number of transactions per block. This chart will be generated with input the start-end block number. With that oversimplified chart a developer can understand if everything works on the experiment-application he is implementing. It may seem unneeded but it can reveal important bugs on a simulation. Picture 4.10 demonstrates the previous description.

Get Time to Mine Block One of the most known restrictions that blockchain has is the time restriction. This is due to the time needed to mine a block, which contains the transactions. Thus, any system must wait for the next block to be mined and after that its transactions will be executed. This constrain demands a more specific architecture for each use case system. To do so the developer must easily observe the time between the mined blocks and design his system accordingly. Thus, a very useful chart is provided, which contains the previously mentioned information.

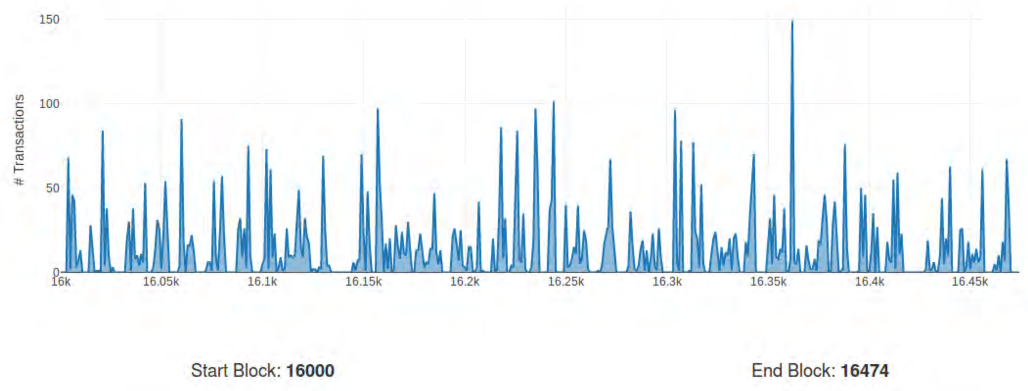


FIGURE 4.10: Transactions per block

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

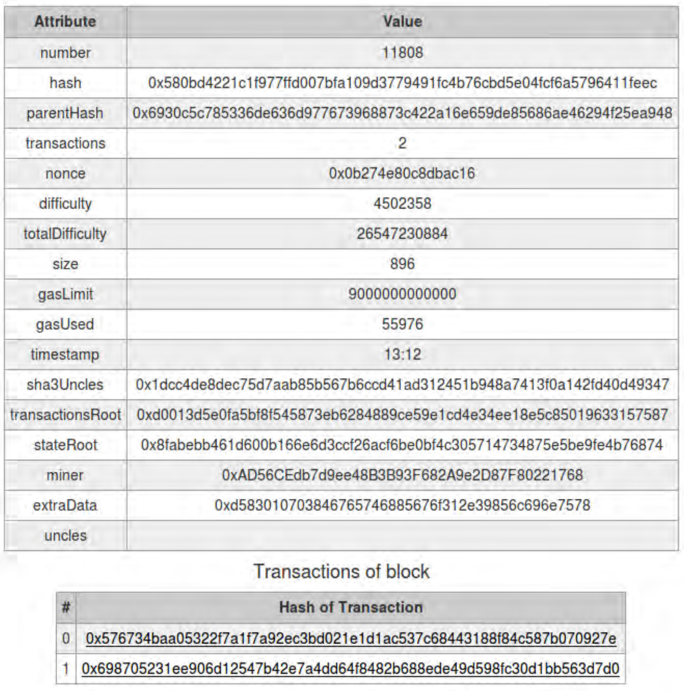
4.3.4 Get Specific Entities

Get Block This function doesn't need a lot of explanation. The user can specify a block and then the basic information of that block will be returned. It is really simple but it is a lot faster from doing it manually from a the console of an Ethereum node. Some of the info that is returned, is the timestamp of the block that indicates the exact time the block was mined, the number of transactions that this block contains, the gas used in that block, the global gas limit, the miner and many more. This function also displays the hyperlinks of the transactions that are contained in the specified block. This function is embedded in other scenarios to achieve a higher usability. Picture 4.11 below demonstrates the previous description.

Get Transaction This function is also embedded in previous scenarios but it may be useful from time to time to get a specific transaction and its information. Some of the basic information are the block number that this transaction was mined, which is a hyperlink to the block, the transaction index inside that block, the sender and the receiver of the transaction and the gas cost of the transaction. Picture 4.12 below demonstrates that function.

Get Account Info As mentioned in the first scenario-function, it is really useful to get the transactions and some other basic information about an account through a specified range of blocks. This function does exactly that, it has as input the range of blocks and the hash of the account. It returns a table of the transactions sent from that account, which are hyperlinks to get further information about each transaction. This function also returns the current balance of the account and the total number of transactions from that account for the whole blockchain. Picture 4.13 below demonstrates that function.

Get Number of Peers The web3.js library which is being used to get all the information from blockchain and represent it on a web UI. It has a lot of functions about the transactions and the accounts but there are minimal functions to get information about the state of the network. The only function that returns information related to the network is the "getPeerCount()", which



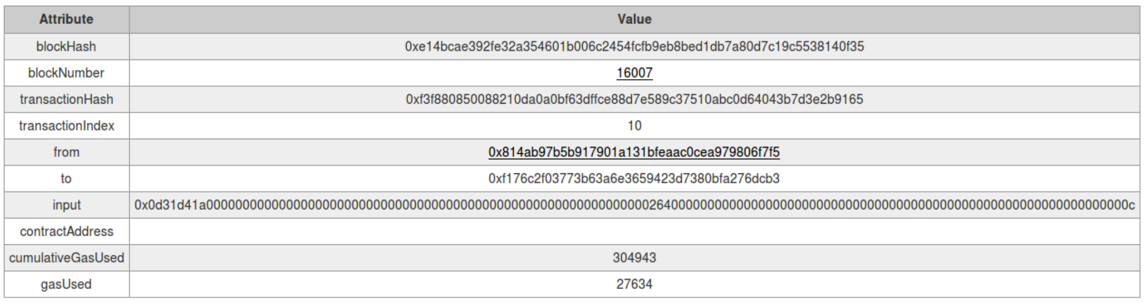
Attribute	Value
number	11808
hash	0x580bd4221c1f977ffd007bfa109d3779491fc4b76cbd5e04fcf6a5796411feec
parentHash	0x6930c5c785336de636d977673968873c422a16e659de85686ae46294f25ea948
transactions	2
nonce	0x0b274e80c8dbac16
difficulty	4502358
totalDifficulty	26547230884
size	896
gasLimit	9000000000000
gasUsed	55976
timestamp	13:12
sha3Uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
transactionsRoot	0xd0013d5e0fa5bf8f545873eb6284889ce59e1cd4e34ee18e5c85019633157587
stateRoot	0x8fabebb461d600b166e6d3ccf26acf6be0bf4c305714734875e5be9fe4b76874
miner	0xAD56CEdb7d9ee48B3B93F682A9e2D87F80221768
extraData	0xd583010703846765746885676f312e39856c696e7578
uncles	

Transactions of block

#	Hash of Transaction
0	0x576734baa05322f7a1f7a92ec3bd021e1d1ac537c68443188f84c587b070927e
1	0x698705231ee906d12547b42e7a4dd64f8482b688ede49d598fc30d1bb563d7d0

FIGURE 4.11: Block Info

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

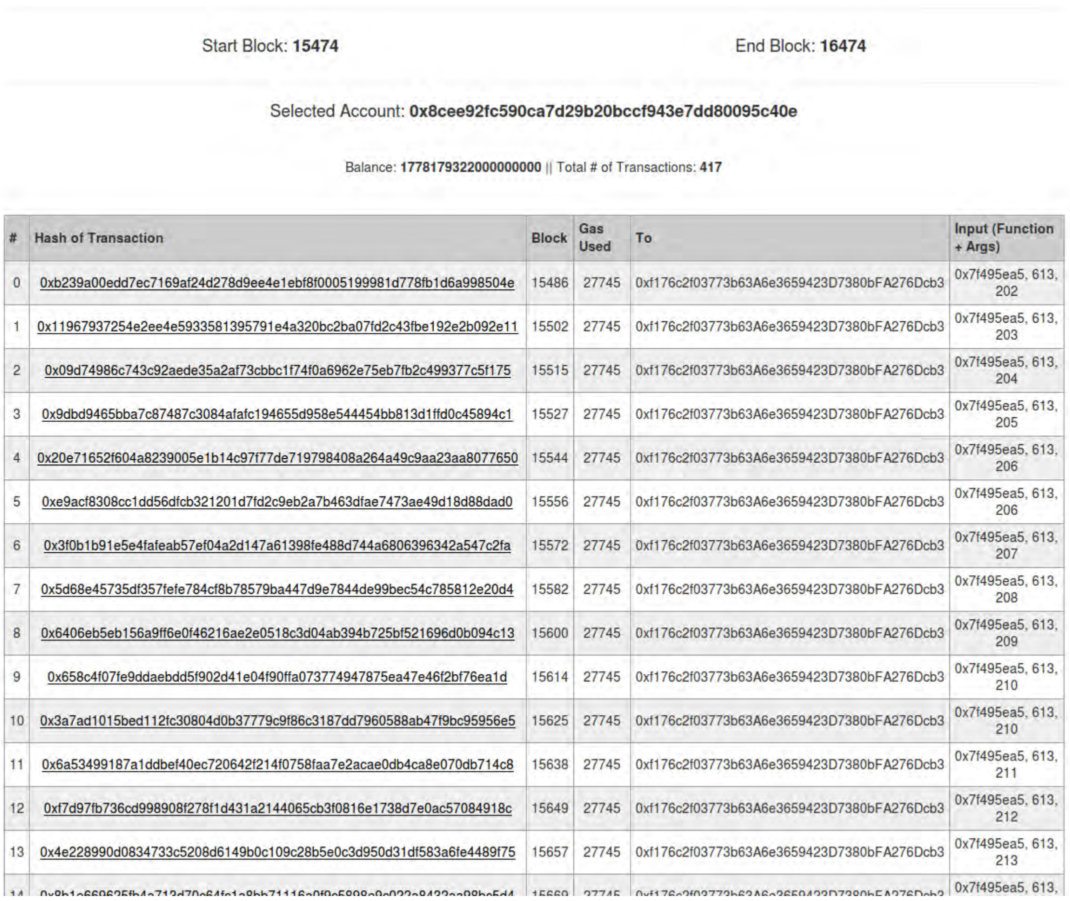


Attribute	Value
blockHash	0xe14bcae392fe32a354601b006c2454fcfb9eb8bed1db7a80d7c19c5538140f35
blockNumber	16007
transactionHash	0xf3f880850088210da0a0bf63dffce88d7e589c37510abc0d64043b7d3e2b9165
transactionIndex	10
from	0x814ab97b5b917901a131bfeaac0cea979806f7f5
to	0xf176c2f03773b63a6e3659423d7380bfa276dcb3
input	0x0d31d41a0000000000000000000000000000000000000000000000000000000000000264000000000000000000000000000000000000000000000000000000000000000c
contractAddress	
cumulativeGasUsed	304943
gasUsed	27634

FIGURE 4.12: Transaction Info

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

returns the number of Ethereum nodes that are connected to the node you are using to get the data. It would be really helpful to implement more functions about the network state, such as one that would return the hash of the neighbor nodes and theirs too. This would make possible to represent the blockchain network on a chart with connected nodes, which could help a lot in debugging a network that depends so much at the neighbors of each node, like blockchain.



Start Block: **15474** End Block: **16474**

Selected Account: **0x8cee92fc590ca7d29b20bccf943e7dd80095c40e**

Balance: **177817932200000000** || Total # of Transactions: **417**

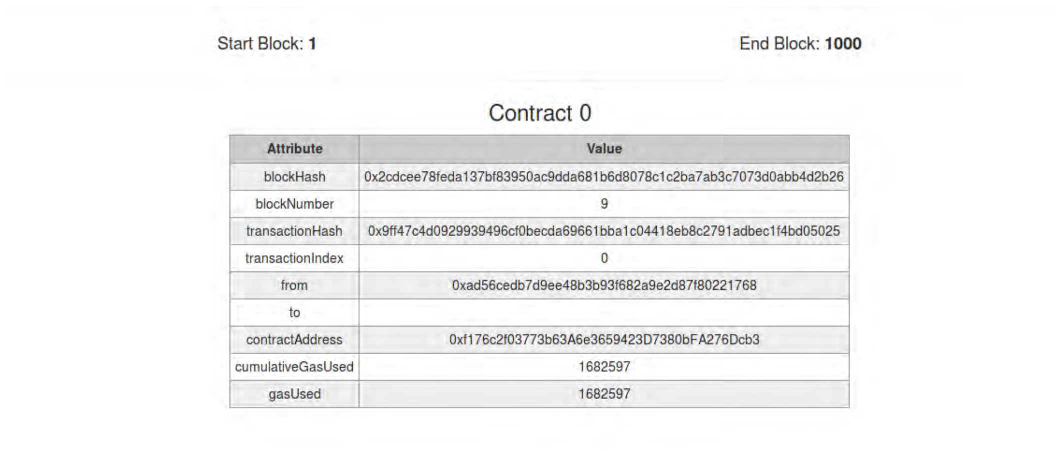
#	Hash of Transaction	Block	Gas Used	To	Input (Function + Args)
0	0xb239a00edd7ec7169af24d278d9ee4e1ebf8f0005199981d778fb1d6a998504e	15486	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 202
1	0x11967937254e2ee4e5933581395791e4a320bc2ba07fd2c43fbe192e2b092e11	15502	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 203
2	0x09d74986c743c92aede35a2af73cbbc1f74f0a6962e75eb7fb2c499377c5f175	15515	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 204
3	0x9dbd9465bba7c87487c3084afafc194655d958e544454bb813d1ffd0c45894c1	15527	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 205
4	0x20e71652f604a8239005e1b14c97f77de719798408a264a49c9aa23aa8077650	15544	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 206
5	0xe9acf8308cc1dd56dfcb321201d7fd2c9eb2a7b463dfae7473ae49d18d88dad0	15556	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 206
6	0x3f0b1b91e5e4fafeab57ef04a2d147a61398fe488d744a6806396342a547c2fa	15572	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 207
7	0x5d68e45735df357fefe784cf8b78579ba447d9e7844de99bec54c785812e20d4	15582	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 208
8	0x6406eb5eb156a9ff6e0f46216ae2e0518c3d04ab394b725bf521696d0b094c13	15600	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 209
9	0x658c4f07fe9ddaebdd5f902d41e04f90ffa073774947875ea47e46f2bf76ea1d	15614	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 210
10	0x3a7ad1015bed112fc30804d0b37779c9f86c3187dd7960588ab47f9bc95956e5	15625	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 210
11	0x6a53499187a1ddbef40ec720642f214f0758faa7e2acae0db4ca8e070db714c8	15638	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 211
12	0xf7d97fb736cd998908f278f1d431a2144065cb3f0816e1738d7e0ac57084918c	15649	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 212
13	0x4e228990d0834733c5208d6149b0c109c28b5e0c3d950d31df583a6fe4489f75	15657	27745	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3	0x7f495ea5, 613, 213
14	[illegible]	15669	27745	[illegible]	0x7f495ea5, 613,

FIGURE 4.13: Account Info

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

4.3.5 General Contracts

Get Contracts When creating a smart contract the first step, after debugging, is to compile it and then deploy it at blockchain. After that anyone can make use of that smart contract and its logic. Nevertheless, in a private network there would be a lot of versions of smart contracts for the cause of testing and debugging. Thus, to get the appropriate data from blockchain the developer should know when each experiment started. The only way to distinguish between the transactions of the different experiments without getting all the data of blockchain, is to find the different pieces of blocks that contain that transactions. A flag of that pieces is the block that the smart contract of each test was mined. Thus, by finding that block we know when the transactions of that test started and we can save time by getting only the needed blocks. This function takes as input the start-end number of blocks and searches for contracts mined in that range. If there are any contracts, they will be returned with some information that describes them. Figure 4.14 demonstrates the previous description.


Start Block: **1** End Block: **1000**

Contract 0

Attribute	Value
blockHash	0x2cdcee78feda137bf83950ac9dda681b6d8078c1c2ba7ab3c7073d0abb4d2b26
blockNumber	9
transactionHash	0x9ff47c4d0929939496cf0becda69661bba1c04418eb8c2791adbec1f4bd05025
transactionIndex	0
from	0xad56cedb7d9ee48b3b93f682a9e2d87f80221768
to	
contractAddress	0xf176c2f03773b63A6e3659423D7380bFA276Dcb3
cumulativeGasUsed	1682597
gasUsed	1682597

FIGURE 4.14: Get Contracts

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

Interact with Smart Contracts The functionality of a Dapp exists in the smart contract. To interact with it you need the ABI, which is the description of the smart contract and the only way to communicate with it. The ABI is the result of the compilation of the smart contract. For example it contains the inputs and their types, and it is compulsory to encode and decode the information to and from the blockchain. Thus, through the UI the user can submit the code of a smart contract which will be compiled and generate the ABI. This ABI can be used through the UI to call functions that don't cost gas. A simple example, is to call a function that returns a global variable, so that the developer can monitor any smart contract after providing the address of the corresponding mined contract. Nevertheless, when monitoring a system you need a real time interface. Therefore, it was implemented a real time chart that each dot contains the return value of the aforementioned function. This enables an easy monitoring of any smart contract.

4.3.6 Custom Contracts

Get state of Contract (Now - Through Blocks) Through an implementation of a blockchain application it is utmost importance for a developer, the ability to obtain the values of a smart contract for past blocks. In that way, it is easier to monitor and debug a smart contract. All the explorers available now for the Ethereum network, don't support such an important function. The web3.js library provides such a function and we used that to create a scenario to return a much bigger range of values.

The first simple function implemented is returning the current value of the specified variable of the smart contract.

To implement that functions the information of the compiled smart contract was compulsory. After compiling the smart contract either at an on-line tool like remix, or locally with the web3.js library, the first value to use is the hash of the mined smart contract. After that, we need to keep the generated hashes for each function. If the developer has implemented "get" functions this will return the variables he want. After that, it is very simple to obtain that values just by calling that functions from the web3.js function. Nevertheless, if we want to obtain the values of the variables at previous blocks, then we need to get the so called "state of the contract" in each block.

This part is quite difficult at first and different for each implementation. To get the state of the contract the arguments needed are the contract hash, the block number and the index position of the storage. The last one cannot be specified with a deterministic way because the index of the variable is based on the declaration series of the variables. Thus, by testing after reading the official documentation anyone can find what is the index of the wanted variables [46].

Such data are very useful in a chart but also in a table of data to better monitor and inspect the function of the developed smart contract. An example chart is following at 4.15.

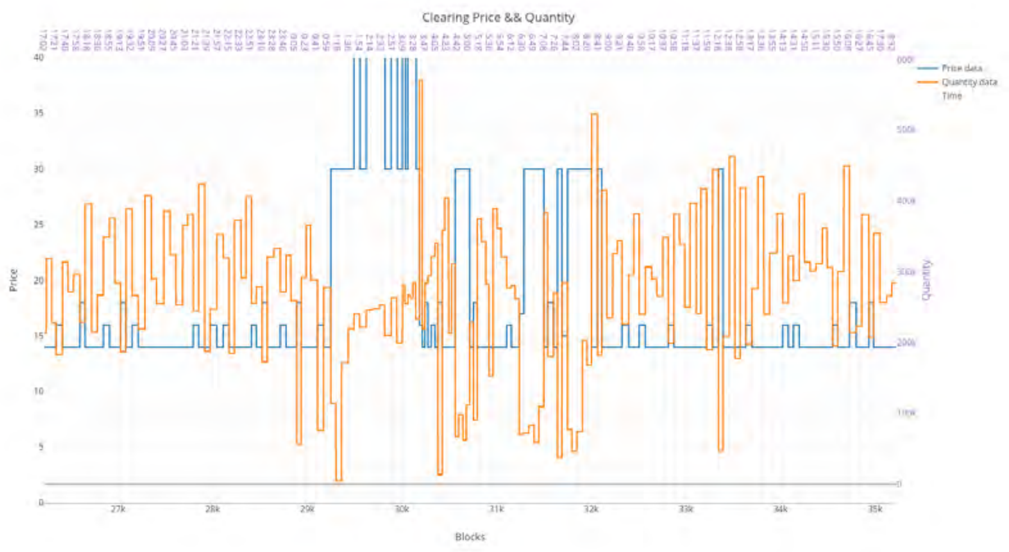


FIGURE 4.15: Get Contracts

Source: https://github.com/Temeteron/Ethereum_analytics_debugger

At 4.15 chart two values of a smart contract are presented. The bottom xx' axis contains the number of block and the top axis contains the time of day. Thus, by inspecting that chart a developer can very easily detect bugs and wrong values at his implementation.

Currently, this function is hard coded and not general enough to use right away. Nevertheless, there are specific instructions in the repository of the inspector [47], so that any developer can change the arguments for his needs and test the contract. Obviously, this is not the logic way to implement that function and we have already begun the implementation of a more general and user friendly function to get the state of the contract through blocks.

Get Chart from transactions Most of the transactions in an Ethereum network contain arguments for the called function of a contract. Those arguments have different meaning for each system, depending on the logic that was implemented on the contract. Nevertheless, these values have some meaning and it could be helpful to inspect those arguments in a chart. For that reason we created a chart, which decodes each transaction's arguments and represents them graphically.

4.3.7 Comparison Table of Tools

The following table presents the features of the Inspector and of the previous presented tools in short. This table can be used as a reference to evaluate and compare such tools.

#	Features	Inspector	Other Tools
1	Get Block	✓	2 3 4 5 6
2	Get Transaction	✓	2 3 4 5 6
3	Explore 1,2	✓	2 3 4 5
4	Get specified range of blocks	✓	
5	Sync with a big number of blocks	✓	
6	Table (accounts, # ts, gas spent)	✓	
7	Get Ts through Blocks	✓	
8	Chart Block Information	✓	1
9	Chart Gas Spent of Account	✓	
10	Chart Balance of Account	✓	
11	Chart Transactions Per Block	✓	1
12	Chart Time to Mine Block	✓	1
13	Get Account Detailed Info	✓	
14	Get # Peers of Node	✓	
15	Live Monitoring	✓	1
16	Find Mined Contracts	✓	
17	Compile Contract - Get ABI	✓	8
18	Call get Functions of Contract	✓	
19	Support Private Networks	✓	2 3 5 6 7
20	Support Public Networks	✓	1 2 3 5 6 7
21	Fast Setup	✓	
22	Good Documentation	✓	

4.3.8 Development Issues

Always, while developing a tool or a platform a lot of issues may come up and make things much harder and slower. In case that other tools and libraries are involved, at which a developer has no access and edit permissions, the issues that will come up may be a huge time loss. At a tool of this nature, which purpose is to monitor another really big platform - architecture, many issues may be unveiled that occur from different parts of this platform.

During the development of this tool, the issues emerged from the Ethereum Geth Node and the web3.js library. Obviously those two components were compulsory to use, and there weren't any other choices, to connect to an Ethereum network and monitor it in a web page. The issues that emerged will be enumerated below.

Library web3.js

- It is unoptimized. When making requests to the Ethereum node it opens and closes a new socket every time instead of using a previous one. This is obviously a step that increases the response time. This implies that the limit of opened files will be exceeded with a medium number of requests (1000). The default limit in linux distribution is 1024 opened files.
- By making synced requests the web3.js is even slower, but the syncing is needed for a lot of important reasons.

Ethereum Node

- The node cannot answer multiple requests. If a lot of requests are submitted, then it returns null because there is no queue implemented in the node. This is probably a way to keep the implementation of an Ethereum Node minimal.
- The node responds on requests really slowly, as a results all the requests are delayed.
- The node sometimes returns 'Null' values without any reason. If we remake the requests the return values are valid.

As mentioned previously the solutions on these issues weren't obvious enough. Nevertheless, some of them were as simple as changing the limit of open files in our distribution to solve the problem of the unneeded created sockets. All of the rest issues could not be solved totally because we could not and didn't want to change the core code of the Ethereum Node.

4.3.9 Architecture

By implementing this tool an interesting architecture decanted, which is demonstrated in the following picture.

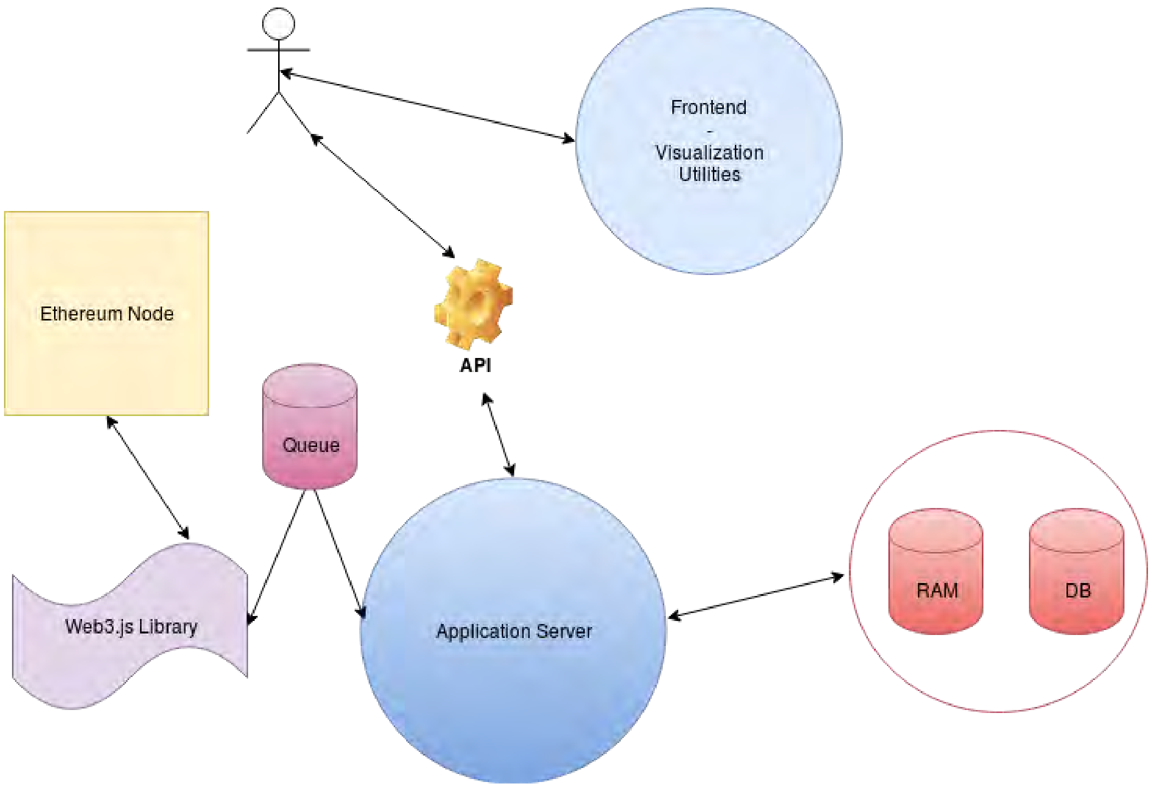


FIGURE 4.16: Architecture of Inspector

The basic use case of the above architecture is the following. After initiating the Ethereum Node and the Inspector, the NodeJs server will try to connect with the Ethereum Node via the web3.js library and will start getting the live information that were previously mentioned. All the information will be represented in the frontend of the tool which was implemented with bootstrap and Handlebars. To demonstrate beautiful and easy to understand charts the Plotly.js library was used, which can redirect the user from the chart in our page to the official page of **Plotly**, where a lot of functions are ready for use on the previous chart. The actor uses the tool through the frontend and requests any information he needs. This information most of the times needs pre-sync with the blocks and the transactions of the requested range. Those data are stored in the RAM. The reason that a database wasn't implemented, was to give a developer a quick start without a lot of libraries to install. Thus, because of the previously mentioned problems, we try to minimize the number of requests by storing data in the RAM to decrease the response time. The NodeJs server will create a queue with a bottleneck for the requests after splitting them into chunks of smaller requests. As mentioned this was obligatory to get valid responses from the Ethereum Node. All these requests are made through the web3.js library which is the official Ethereum library to communicate with an Ethereum Node.

Target of the tool and prospects In conclusion, i believe it is really significant to point out the purpose of the tool that was created.

- Create an open source, easily expandable, lightweight web tool to debug and visualize an Ethereum private network.
- Provide a lot of functions by using the web3.js API, about the transactions, blocks, accounts, cost, and extra useful scenarios.
- Alongside with the functions of the block explorer, the user can access and interact with his smart contract, so it can easily debug it.
- Create useful scenarios to debug an experiment that doesn't only depend on a smart contract but also on the network itself.
- Use Plotly.js for Visualization with charts (No previous tool has charts)
- Solve common problems that previous tools didn't. An example is the creation of a queue to get a lot of blocks and transactions. Ethereum nodes doesn't support it.

Chapter 5

Synopsis and Future Work

5.1 Synopsis

The aim of this thesis was the development, the implementation and the analysis of the blockchain technology. This research on blockchain technology in theoretical but also practical level was really helpful to understand the capabilities of this new system, which are really big and can be applied almost everywhere.

Initially, some of the most well known blockchain architecture were analyzed to create a theoretical background. After that a more detailed analysis of the Ethereum network was presented. Alongside the capabilities and advantages of this network, a brief introduction in the smart contracts took place, which is one of the most important features of Ethereum.

Furthermore, the basic vulnerabilities of an Ethereum blockchain network were presented and analyzed. This chapter was important, so that the reader could understand the problem that the Inspector ought to solve. As mentioned there many problems, pitfalls that cannot be detected without the needed knowledge and understanding of a blockchain network. For that reason a tool such as the Inspector, which was the result of that research, could help a lot of developers that may not have the required knowledge to debug a decentralized application.

All the previous projects-tools that already exist were stated. An empirical comparison took place by contradicting the basic features of each tool. All the pros and cons of each tool were marked up, so that the design and implementation of the Inspector could be efficient and with plenty of functions.

After that, the implemented tool was analyzed in depth by showing all the possible use cases and useful functions. Inspector oughts to be the fastest way to monitor and debug decentralized applications with minimal requirements and installs in your system.

5.2 Future Work

Tools that help developers at their implementations must always be updated and refreshed. When a new technology such as blockchain rises, there is a lot of space to create new tools or enrich the existed ones, so that a transition or simple tests could be more easy.

The plethora of blockchain variants are really interesting and i ought to explore them in depth and find the best for each case. This is obviously a difficult task thinking the vast technology-knowledge that is produced every day, and even more time consuming. There are a lot of different architectures, each designed for different problems with the same basic concept, decentralization.

As far as the tool that was created, the first feature that should be noted about the Inspector is that it will always be an open-source project with the previously mentioned targets-purposes. A lot of additions could be implemented in the aspect of the functions and about the scaling of the tool. Probably one of the first to implement should be a database, for the already synced data. Currently, all the data are being saved on the ram of the computer which obviously is not the best if you want to monitor a big range of blocks. For that reason a version that will use a proper database will be implemented. The version that depends on the ram will continue to exist so that anyone could use it for a faster installation with much less requirements.

After that, some of the functions of the Inspector will be materialized again to be more general and efficient. A simple example is the functions that interact with the smart contracts. As mentioned, currently the basic function about a smart contract is to get the values of the specified variables over time. This function will generate a chart which shows in a more user friendly way the variables values variations. To get the needed variables the developer must change some lines of code so that the function point to his inner smart contract functions. To get over this process a function will be implemented that will have as input the smart contract's code and will generate all the hashes (addresses) of the functions. Those functions could be called from the user interface and the tool could be instantly more generic and easy to use.

There are also a lot of functions that could be implemented by using the web3.js library but didn't. An example is the compilation and deployment of a smart contract through the tool. It is obvious enough that this function doesn't belong in this tool, which was created to monitor and inspect, but nevertheless could be a useful addition. The functions that could be added are a lot and for that reason this project is open-source, so that anyone could easily customize it for his needs.

Other really important longterm future prospects concern compatibility and support of other blockchain architectures. Currently the Inspector fully supports the Ethereum and the previous was chosen because of the functionality it provides and the high acceptance by the community. Another emerging and promising blockchain architecture is the Hyperledger [48] [49] [50] [51], which is the enterprise blockchain solution provided by IBM with much more security and privacy features implemented, compared to other blockchain architectures. Thus, supporting that technology could increase the importance and usefulness of the Inspector. To support Hyperledger a few additions are needed. Because of the Inspector's architecture it is achievable to support two different blockchain architectures by using the intermediate API. The basic addition at the server side is to add an intermediate node which will identify the type of the blockchain and use the proper library to interact and monitor the network. After that the monitoring and visualization process has meaning for both architectures because both are blockchain, which means that a lot of features are same.

The last thing that should be noted on this Thesis, is that decentralized applications become more and more usual but the required knowledge to begin with that technology is not so widespread. Tools that make easier and more efficient the development in such new technologies will always be in great need. Thus, such tools help the world to develop faster and transit to new technologies more easily.

Bibliography

[1] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System | Satoshi Nakamoto Institute*, 2008. [Online]. Available: http://nakamotoinstitute.org/bitcoin/.

[2] S Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", 2008. [Online]. Available: http://www.academia.edu/download/32413652/BitCoin_P2P_electronic_cash_system.pdf.

[3] *What is Blockchain Technology?* [Online]. Available: https://blockgeeks.com/guides/what-is-blockchain-technology/.

[4] *CoinMarketCap*. [Online]. Available: https://coinmarketcap.com/currencies/bitcoin/.

[5] "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", 1824. [Online]. Available: http://www.cryptopapers.net/papers/ethereum-yellowpaper.pdf.

[6] *Stellar Lumens (XLM) – Whitepaper*. [Online]. Available: https://whitepaperdatabase.com/stellar-lumens-xlm-whitepaper/.

[7] D. Maz Eres, "The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus", [Online]. Available: https://www.stellar.org/papers/stellar-consensus-protocol.pdf.

[8] V. Buterin, "A next-generation smart contract and decentralized application platform", *White paper*, 2014.

[9] "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", [Online]. Available: https://bravenewcoin.com/assets/Whitepapers/Ethereum-A-Secure-Decentralised-Generalised-Transaction-Ledger-Yellow-Paper.pdf.

[10] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", [Online]. Available: www.bitcoin.org.

[11] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)", in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, Springer-Verlag New York, Inc., 2017, pp. 164–186, ISBN: 978-3-662-54454-9. DOI: 10.1007/978-3-662-54455-6{_}8. [Online]. Available: http://link.springer.com/10.1007/978-3-662-54455-6_8.

[12] Y. Sompolinsky and A. Zohar, "Secure High-Rate Transaction Processing in Bitcoin", [Online]. Available: https://fc15.ifca.ai/preproceedings/paper_30.pdf.

[13] *Scanning Live Ethereum Contracts for the &quot;Unchecked-Send&quot; Bug*. [Online]. Available: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[14] *Understanding The DAO Attack - CoinDesk*. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists/.

[15] D. Boneh and M. Naor, "Timed Commitmen ts", [Online]. Available: https://pdfs.semanticscholar.org/764b/41d1cf0c2c64bec722f0afd4b0a2ce0bee27.pdf.

[16] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure Multiparty Computations on Bitcoin", in *2014 IEEE Symposium on Security and Privacy*, IEEE, May 2014, pp. 443–458, ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.35. [Online]. Available: http://ieeexplore.ieee.org/document/6956580/.

[17] B. Marino and A. Juels, "Setting Standards for Altering and Undoing Smart Contracts", in, 2016, pp. 151–166. DOI: 10.1007/978-3-319-42019-6{_}10. [Online]. Available: http://link.springer.com/10.1007/978-3-319-42019-6_10.

[18] *Hard Fork Completed - Ethereum Blog*. [Online]. Available: https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[19] *Ethereum Classic*. [Online]. Available: https://ethereumclassic.github.io/.

[20] *Announcement of imminent hard fork for EIP150 gas cost changes - Ethereum Blog*. [Online]. Available: https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/.

[21] *Explaining EIP 150 : ethereum*. [Online]. Available: https://www.reddit.com/r/ethereum/comments/56f6we/explaining_eip_150/.

[22] J Bonneau, J Clark, S. G.I. C. ePrint, and u. 2015, "On Bitcoin as a public randomness source.", *Pdfs.semanticscholar.org*, [Online]. Available: https://pdfs.semanticscholar.org/ebae/9c7d91ea8b6a987642040a2142cc5ea67f7d.pdf.

[23] C. Pierrot and B. Wesolowski, "Malleability of the blockchain's entropy", Jul. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01364045.

[24] *makerdao/maker-darts: A random number generating game for Ethereum*. [Online]. Available: https://github.com/makerdao/maker-darts.

[25] *randao/randao: RANDAO: A DAO working as RNG of Ethereum*. [Online]. Available: https://github.com/randao/randao.

[26] I. Free Software Foundation, *GDB: The GNU Project Debugger*, 2009. [Online]. Available: https://www.gnu.org/software/gdb/.

[27] *27.3. pdb — The Python Debugger — Python 3.6.5 documentation*. [Online]. Available: https://docs.python.org/3/library/pdb.html.

[28] *Remix - Solidity IDE*. [Online]. Available: http://remix.ethereum.org/#optimize=false&version=soljson-v0.4.21+commit.dfe3193c.js.

[29] M. Bartoletti, A. Bracciali, S. Lande, and L. Pompianu, "A general framework for blockchain analytics", Jul. 2017. [Online]. Available: http://arxiv.org/abs/1707.01021.

[30] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns", Mar. 2017. [Online]. Available: http://arxiv.org/abs/1703.06322.

[31] *Ethereum Network Status*. [Online]. Available: https://ethstats.net/.

[32] *Ethereum Network Stats*. [Online]. Available: https://github.com/cubedro/eth-netstats.

[33] *A lightweight ethereum block explorer*. [Online]. Available: https://github.com/etherparty/explorer.

[34] *Ethereum Block Explorer (ETHExplorer V2) - Realtime Price Ticker, Shapeshift.io Integration, etc.* [Online]. Available: https://github.com/carsenk/explorer.

[35] *Lightweight Ethereum blockchain explorer*. [Online]. Available: https://github.com/gobitfly/etherchain-light.

[36] *Ethereum Blockchain Explorer*. [Online]. Available: https://github.com/maran/ethereum-blockchain-explorer.

[37] *Ether Scan*. [Online]. Available: https://etherscan.io/.

[38] *etherchain.org - The Ethereum Blockchain Explorer*. [Online]. Available: https://www.etherchain.org/.

[39] *Homepage - QuickBlocks*. [Online]. Available: https://quickblocks.io/.

[40] *A tool to monitor a number of smart contracts and transactions*. [Online]. Available: https://github.com/Neufund/smart-contract-watch.

[41] *Keep Your Private Keys Close and Keep Your Smart Contracts Closer — Introducing The Smart Contract...* [Online]. Available: https://blog.neufund.org/keep-your-private-keys-close-and-keep-your-smart-contracts-closer-introducing-the-smart-contract-e3bd1fcad204.

[42] *Ethereum Overview | Truffle Suite*. [Online]. Available: http://truffleframework.com/tutorials/ethereum-overview.

[43] *Truffle Suite - Your Ethereum Swiss Army Knife*. [Online]. Available: http://truffleframework.com/.

[44] *web3.js - Ethereum JavaScript API — web3.js 1.0.0 documentation*. [Online]. Available: https://web3js.readthedocs.io/en/1.0/.

[45] *Promise - JavaScript | MDN*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[46] *Layout of State Variables in Storage*. [Online]. Available: https://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-variables-in-storage.

[47] *Temeteron/Ethereum_analytics_debugger: A NodeJs project to get various analytics and debug a private ethereum network.* [Online]. Available: https://github.com/Temeteron/Ethereum_analytics_debugger.

[48] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains", Jan. 2018. DOI: 10.1145/3190508.3190538. [Online]. Available: http://arxiv.org/abs/1801.10228http://dx.doi.org/10.1145/3190508.3190538.

[49] *Home - Hyperledger*. [Online]. Available: https://www.hyperledger.org/.

[50] *Hyperledger - Open source blockchain for business – IBM Blockchain*. [Online]. Available: https://www.ibm.com/blockchain/hyperledger.html.

[51] J. Sousa, A. Bessani, and M. Vukolić, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform", Sep. 2017. [Online]. Available: http://arxiv.org/abs/1709.06921.