Università degli Studi di Cagliari

Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Matematica e Informatica
Ciclo XXXIII

PHD DEGREE

# A User-Centered Perspective for the blockchain Development

S.S.D. INF/01

Candidate
Giuseppe Antonio Pierro

Supervisor
Prof. Michele Marchesi
Prof. Roberto Tonelli
Dr. Stéphane Ducassse

PhD Coordinator
Prof. Michele Marchesi

Final examination academic year 2020/2021
September 1, 2021

# Acknowledgements

I first wish to thank my supervisors, Proff. Roberto Tonelli and Michele Marchesi, for their constant encouragement and valuable suggestions when working on the research presented in my dissertation, and my colleagues Gavina Baralla, Luisanna Cocco, Giacomo Ibba, Simona Ibba, Ilaria Lunesu, Lodovica Marchesi, Katiuscia Mannaro, Marco Ortu, Andrea Pinna, Raffaele Porcu and Nicola Uras at the Department of Mathematics and Computer Science at the University of Cagliari. I warmly thank the FlossLab team where I spent about nine months of my Phd Program, and especially Dr. Marco Di Francesco for supervising my work there.

I am especially indebted to Dr. Stéphane Ducasse, for his invaluable critical thinking and deep insights on my work work, and the RMoD project team at the INRIA Institute (Lille, France), where I spent part of my PhD program. I especially thank Nicolas Anquetil, Alexander Bergel, Christopher Fuhrman, Henrique Rocha and Oleksandr Zaitsev, to dedicate their time in discussing with me some of the main issues of this dissertation. Many suggestions on software development come from the pair program activity with Pharo, and especially from Guillermo Polito, Cyril Ferlicot and Santiago Bragagnolo: they helped me in becoming a better software developer and to build some of the tools presented in this dissertation. A special thanks goes to my colleagues Mahugnon Honore, Allex Oliveira and Andy Amoordon for sharing research time and interests.

Finally, my deep and sincere gratitude goes to my wife for all her love and support, and my kids, Leonardo for waking up early in the morning on Sunday saying "Papà sveglia, c'è il sole!" when I could sleep more and Ilaria for waking up during the night because she wanted to sleep with her mama.

# Abstract

Blockchain technology is regarded as one of the most important digital innovations in the last two decades. Its applicability beyond cryptocurrencies has been a growing topic of research interest not only in computer science but also in other areas, such as marketing, finance, law, healthcare, etc. However blockchain is far from reaching the population on a larger scale. The dissertation evaluates the causes that are preventing successful implementation and adoption of blockchain technology at a larger scale, supporting infrastructure for public and private companies.

The latest academic research suggests that the blockchain services are still in an early stage, and standards for developing blockchain-based applications have not been defined yet. Moreover the interaction with the blockchain technology is still complex, especially for non expert users, because it requires many technical skills. The dissertation focuses on this knowledge gap as a cause for the blockchain missing reach on society at a larger scale. This work aims to fill the gap by presenting innovative methodologies and user-centered models that could help the adoption of the blockchain technology by a larger number of private/public companies and individuals. Based on these models, specific tools for both expert and non-expert users are developed and discussed in the dissertation.

First, tools for expert users, i.e., software developers, are proposed to analyze the smart contracts' source code, to collect the smart contracts in a reasoned repository, and to identify code clones and boost the use of open source libraries for a better collective practice in developing and maintaining the blockchain. Second, tools for non-expert users, i.e. people with no technical knowledge, are proposed to suggest them the fairest fees to pay to have their transactions executed according to the price and waiting times they are willing to spend, and to identify malicious smart contracts that can deceive them, thus preventing them to trust the blockchain and use it again. Finally, visualization models for users with expertise in different disciplines are proposed to provide them with graphical representations that can foster the understanding of the blockchain underlying mechanisms.

# Contents

# Chapter 1

# Introduction

In the last two decades, multiple innovative technologies have substantially impacted most of the world's population and its economy. The most notable innovative technologies of them are: Cloud Computing, Big Data, the Internet of Things (IoT), Augmented Reality, and, the last in order of time, Blockchain.

The advantages and opportunities for the use of blockchain technology in different industrial and technical areas were clear from the very beginning to the researchers, who wrote that nowadays the blockchain technology could have an impact similar to the World Wide Web (WWW) in the nineties (**?????**). Indeed, the blockchain technology was initially introduced as a technology that allowed to track cryptocurrencies transactions, but then a set of new and promising features were developed in the Ethereum blockchain to apply its advantages to other fields (**?????**).

Governments and companies around the world started wondering about the possible implementation of blockchain technologies in many areas of life, not directly associated with the use of cryptocurrencies (**?**). One of the most promising implementation of the Ethereum blockchain technology is its use to create fully automated contracts, i.e., agreements that are performed without human involvement. Such agreements in the information technology environment are frequently referred to as "Smart" contracts (**??**).

However, nowadays only few public organization and companies are using this technology, and it has been estimated that blockchain-based applications will be available to the wider public only in 10–15 years (**?**). The blockchain technology is not yet ready for mass adoption. It is indeed not mature enough for large scale application and use by a large number of (autonomous) objects, individuals, and organizations (**?**).

This dissertation is specifically dedicated to the research and development of innovative technologies for the Ethereum blockchain, that can foster and empower its application on a larger scale. To this aim new methodologies, models and user-centered tools for both expert and non-expert users are proposed to encourage the adoption of the blockchain technology by a larger number of private and public companies, institutions and/or individuals. After providing an introductory pre-

sentation of the blockchain and possible future scenarios of application, the specific questions and hypotheses leading this research will be presented and discussed.

## 1.1 What is the Blockchain?

A blockchain can be described as a public database that is updated and shared across many computers in a network. "Block" refers to the fact that data are stored in sequential "blocks". If a user sends cryptocurrencies to someone else, the transaction data needs to be added to a block to be successful. "Chain" refers to the fact that each block cryptographically references its parent. A block's data cannot be changed without changing all subsequent blocks, as it would require the consensus of the entire network (**?**).

Blockchain owns specific characteristics that make it quite different from classic centralized systems. First of all, despite its original distributed design, nowadays Internet is highly centralized. Most of the Internet traffic is routed through a few centralized services or platforms, managed and controlled by few large corporations, such as Google, Amazon, Facebook, Microsoft, etc. These centralized platforms are very useful tools and they provide to the users great comfort and convenience. Moreover, most users experience the services provided by those private companies as public goods. However, at the same time, most users do not know that their data are exploited by the very same private companies for their own commercial aims. Thus, many researchers in the fields of political science, law, history, philosophy, sociology pointed out the critical challenges that centralized technology platforms pose to democracy (**???**). They pointed out how the centralized technology platforms are part of a long trend of consolidating power for profit (**??**). In contrast, blockchain has provided the users with a radical alternative to the current market economy managed and influenced by few big players, such as central banking system of few countries or multinational technology companies. A blockchain platform has the following key characteristics:

- Decentralize nature. From a technical perspective, the blockchain has no single point of failure: the network can still function even if a large proportion of participants are attacked or taken out. This characteristic is in contrast with the single point of failure, which is typical of centralized systems, where malicious actors may be able to take down the network by targeting the central authority. Moreover, from an economic perspective, centralized systems hold a large amount of sensitive user data. It is not uncommon for online operators to abuse their dominant position over a centralized platform in order to promote their own (economic) interests, often at the expenses of their user-base. On the contrary, applications based on blockchain technology do not have a central authority. The maintenance of the transactions is performed by a network of nodes communicating to one another and running particular softwares (**?**). Even the blockchain protocol developers do not have control

over the users' transactions. As the relevant code is distributed on the basis of the MIT open-source license, it is available for inspection by any interested person and it is subject to the possibility of modifications, that can become a standard only if accepted by the majority of the community and not by a central authority represented by few persons (**?**).

- Anonymous nature. Cryptocurrencies can be used without any special registration or identification procedure. It is sufficient to install a special wallet application to enable the users' transactions with cryptocurrencies. Each wallet consists of cryptocurrency units, a public key and a private key. The private key is used for the transfer of cryptocurrency amount by its owner to another user's wallet. Without knowledge of the private key, the transaction cannot be signed and the cryptocurrencies cannot be spent. The public key is used by other persons to send cryptocurrencies to the recipient user's wallet, and is used by the blockchain network to verify the transactions. The cryptocurrencies' owners are not explicitly identified, but all transactions on the blockchain are public (**?**).

- Absence of single administrator of transactions. Electronic money can be subject to the risk of double-spending (**?**). Unlike physical coins, electronic money (like any computer data) can be duplicated and thus be used more than once. Traditional electronic money systems prevent double-spending by having a centralized trusted administrator who follows an established process to authorize each transaction. The problem with this practice is that the stability of the money system depends on the company running the administrative function, with every transaction having to go through them, just like a bank. The blockchain technology resolves the double-spending problem by using a peer-to-peer network. All the transactions are included in a publicly available database. Information about a new transaction is distributed through the network, verified by blockchain participants named "miners", and then fixed with the indication of the time it was made (the timestamp) and the unique number of the cryptocurrencies unit. Thus, it is possible to trace the entire history of transactions (**?**).

- Resilience to data manipulations from Outside. Cryptography used in the process of creating records on Bitcoin-related transactions in the blockchain database prevents tampering with the content of such records and ensures their perpetual nature. Whenever two people exchange cryptocurrencies, an encrypted record of the transaction is sent out to all other nodes in the blockchain network. The other nodes verify the transaction by performing complex cryptographic calculations on the data in the record ("mining"), and notify one another each time a new "block" of transactions is confirmed as legitimate. When a majority of the nodes agree that a block passes review, they all add it to the blockchain database and use the updated version as a cryptographic

> basis to encrypt and verify future transactions. Each block is guaranteed to come after the previous block chronologically because the previous block's hash would otherwise not be known. Each block is also computationally impractical to modify once it has been in the chain for a while, because every block after it would also have to be regenerated. Thus, it is not possible to rewrite information about certain transaction once it has been included in the blockchain. Such information will be rejected by the blockchain network, unless the intruder possessed more than 50% of the overall computational power of the blockchain network. As a result, all the members of the blockchain community have a single version of "world-state", i.e., an irreversible blockchain's state of affairs at a given time t. Each participant to a transaction has a copy of the blockchain database, and this is synchronized with the others' copies by the use of a specialized algorithm. All this creates an unprecedented level of distributed trust among the users of the blockchain network, the blockchain being the core element facilitating such trust. The participant at each node of the network can access the blocks shared across that network and can own an identical copy of theirs. Any changes or additions made to the blockchain are communicated to all participants in a few seconds or minutes. The decentralized and distributed data processing prevents post factum alteration of data, for instance, for fraudulent ends. Essentially, blockchain enacts a consensus mechanism that ensures the accuracy of a transaction without the necessity to trust single transacting parties (**??**).

Beyond these general characteristic of all the blockchains, other characteristics are specific of the Ethereum blockchain. The most important one is the ability to write programs, i.e. the "smart" contracts, that automatically generate a transaction. The "smart" contracts have greatly increased the possibility to facilitate and enforce the execution of agreements among participants. While the very idea of smart contracts was proposed by Nick Szabo (**?**), a gradual implementation has only started in 2015 with the launch of the Ethereum blockchain (**?**).

There is no universally agreed definition of "smart" contracts, both because of the very novel nature of this phenomena, and of its complex technological basis. According to the simplest definition, a "smart contract" is an agreement whose performance is automated. A smart contract can be considered as a trusted third party between non-trusting participants. The ability to automatically execute contracts at no cost drastically reduces the need for supervision, while allowing an increasing number of businesses and users to trade more frequently and efficiently.

## 1.2 What are the Blockchain Technology Use Cases?

The exchange of money, service and goods with high economic value, between persons or companies, are usually controlled and operated by a third party organization. For instance, a sales-purchase transaction might need to be authenticated by a notary. Making a digital payment or currency transaction requires a bank or credit card provider as a "middleman" to complete the transaction. The same process occurs in several other cases, such as government and public sector applications, including land registration, identity management, taxation, health care, corporate registration, supply chain traceability, insurance contracts. In addition, a transaction causes a fee from a bank or a notary. The Ethereum blockchain technology has been developed to better cope with these issues. The aim of blockchain technology is to create a decentralized environment where no third physical party controls the transactions and data.

Moreover, the blockchain technology can help to address some challenges in the industrial sphere, such as trust, transparency, security and reliability of data processing (**?**). In different industrial and technical areas, the blockchain technology can indeed be helpful in overcoming specific problems, such as:

- The authorship. The difficulty in protecting and enforcing Intellectual Property (IP) rights has been a major obstacle to share intellectual works in digital forms. The blockchain technology can be used to confirm and preserve the authorship of intellectual works and new ideas in digital forms with no third-party interference (**??**).

- The electronic voting. Building a secure electronic voting system that offers the fairness and privacy of current voting schemes, while providing the transparency and flexibility offered by electronic systems has been a challenge for a long time. The blockchain technology can be used to address some of the limitations in the existing electronic voting system. In particular, it can be used to reduce voters' fraud and increase voters' access (**??**).

- Healthcare organizations. The blockchain technology can be used for managing patient electronic medical records. Health records stored in a blockchain could allow patients to make their structured data available to many specialized people. For instance, the data can be made available to medical researchers to better understand diseases and find ways to prevent, treat and cure them (**?????**).

- Traceability. The blockchain technology can be used to provide traceability. Traceability is becoming an increasingly urgent requirement and a key differentiator in many industries in the supply chain (**??**), including pharmaceutical and medical products (**?**), the agri-food sector (**?**), and high-value goods (**??**).

Traceability is very important in luxury and high-value items whose provenance might otherwise depend on paper certificates and receipts that can be easily lost or altered (**?**).

## 1.3 Does Blockchain reach the society?

Some scientific studies have been conducted to understand whether public and private organizations actually use or are anyway ready to use the blockchain technology (**???**). These studies show that either for economic reasons or for lack of (practical) knowledge, the blockchain has not reach the society in its relevant fields of application (**?**).

According to Savelyev (**?**), organizations are not interested in using the blockchain technology for economic reasons. As the author pointed out, while the blockchain was expected to reduce the costs of transactions and third-party fees, "It would not be correct to conclude that smart contracts are cheaper than regular ones". The infrastructure necessary to implement the smart contracts are expensive and the costs associated with the development of application based on the blockchain architecture are still rather high. Indeed, the company that offers blockchain-based solutions must consider the following costs: paying people with the expertise to translate the contracts' language and legal constraints into the programming language of the smart contracts, to develop interoperability standards, to review the energy and the Information and Communications Technology (ICT) infrastructure, to support the blockchain infrastructure, etc.

Gatteschi, believes that while blockchain has the potential to be disruptive or even transformative in the long term, it will not happen before 2023, because it "needs further technology maturity and hardening, in addition to significant changes to business models, operating processes, societal constructs, and regulatory and governance mechanisms" (**?**). There is a need for a holistic approach to smart contracts that includes business model transformation and adequate governance. As Pradhan, Stevens, and Johnson claim, "Full blockchain development could take five to seven years or longer, or may not occur at all. Early adopters who commit to testing blockchain across the supply chain must be prepared to accept significant levels of risk — and be prepared to fail fast and try again" (**?**).

Confirmation of the fact that the blockchain technology is not ready to be massively adopted by the society comes also from some studies based on online user interviews. The University of Cagliari recently carried out an online survey entitled "Verso un'amministrazione digitale" (Towards a digital administration) within the AIND project "Amministrazioni e Imprese Native Digitali" (Administrations and Digital Native Enterprises). In 2018, the Department of Mathematics and Computer Science conducted the project in collaboration with the Department of Social Sciences (Prof. Paola Piras) and the Department of Economics and Business (Proff. Michela Loi and Chiara Di Guardo) of the University of Cagliari. The AIND survey

aimed to collect the users' requirements to implement the applications to support the digitization process of the Italian public administration.Over 70 participants provided their answers to the survey: 42 of them were expert users having a software development background. However, the online survey highlighted the fact that the blockchain technology is not yet ready for mass adoption (**?**).

The majority of the users (68 of 70) did not have professional experience in blockchain technology, 40 of 70 participants admitted that they do not have any knowledge of blockchain technology. When asked whether they were currently using blockchain applications, only one of them answered that he used blockchain applications several times, 2 of them simply plan to use them, while 68 of 70 participants did not use such applications at all. The questions asked to the participants belonged to different experiential dimensions of the blockchain technologies use. These highly contextual dimensions were: 1) Organizational readiness (the readiness of their organization to accept blockchain as an innovative technology); 2) Change acceptance (the acceptance of the changes that the new blockchain technologies entail); 3) Technology knowledge (the knowledge of the existence of such technologies and eventually their basic features); 4) Business use cases (the knowledge of the use cases where the new blockchain technologies can be applied); 5) Practical experience (the experience as both expert and non expert users).

Table 1.1 summarizes the results of the survey carried out in Sardinia in 2018 in some public administrations. The table is based on the analysis of the responses provided by the participants for each experiential dimension. The total number of mentions about each experiential dimensions is reported. Interestingly, only one participant among over 70 participants claimed that his organization would be ready for the changes entailed by the adoption of the blockchain technology.

Table 1.1: Survey summary to estimate how people perceive blockchain

| Experiential dimensions | Mentions | | |
|---|---|---|---|
| | Blockchain | AI | Cloud computing |
| Organizational readiness | 1 | 20 | 61 |
| Change acceptance | 2 | 20 | 62 |
| Technology knowledge | 4 | 19 | 62 |
| Business use cases | 2 | 10 | 61 |
| Practical experience | 1 | 50 | 65 |

## 1.4 Research Questions

Notwithstanding all the possibilities that the blockchain offers to private and public institutions, still its value as a collective good is underestimated and it remains of limited use in the application to larger-scale targets. It might be argued that it is just matter of time, even though other technologies such as internet and the cloud have been adopted in short time at a world scale. Social and economic reasons can

be provided to explain why the blockchain struggles to become a fully-fledged and extensively used technology, also when compared to other technologies in the past. However, the idea of this dissertation is that this might be due also to a lack of tools to bring the blockchain to the world. More specifically, one of the reasons why the blockchain technologies are so slow to spread is that the users are not provided with tools corresponding to their needs and competences. The need for user-oriented blockchain tools is not just accidental, but rather it depends on the absence of a user-centered model, providing the conceptual and the methodological basis for the implementation of such tools. One of the main objectives of this dissertation is to present and discuss some alternatives to the existing models, i.e. models specifically designed to provide the guidelines for the implementation of user-centered tools.

The current blockchains do not consider the specific characteristics, abilities and interests of the users, which might instead be relevant to make the blockchain technologies within their reach and thus facilitate its introduction on a larger scale. A user-centered model for the blockchain development is instead expected to consider the users' competences and aims. This dissertation discusses two main categories of users: expert users and non-expert users. The expert users are the software developers with a domain-specific knowledge that should be able to develop and design new applications based on the blockchain technology. Non-expert users are users who do not necessarily have technical knowledge in the blockchain domain, but do consume applications based on blockchain technology. Some of these users may be individuals who transfer cryptocurrency funds such as Ether and Bitcoin by using online cryptocurrency wallets. Some of these users might also have other relevant competences in law, economy, finance, etc., and actively participate in the blockchain. These competences might be crucial multidisciplinary resources for a more functional use of the blockchain in different application domains.

In this perspective, the dissertation aims to answer the following research questions:

- From a technical point of view, what are the causes that are preventing successful implementation and adoption of blockchain technology at a large scale?
- In what ways can providing better developer tools improve the confidence of the experts users in the blockchain technology?
- In what ways can improving the existing applications for non-experts users increase their confidence in the blockchain technology?
- What can be a way to ease the multidisciplinary collaboration on the blockchain platform?

Answering these questions is also important to better understand whether and why the blockchain can effectively be a more democratic technology, when designed to address the specific characteristic of its target users. Finally, the answers could tell us to what extent it can encounter the favor of a wider audience.

## 1.5 Hypotheses

To answer the research questions, the dissertation provides the following hypotheses:

- H1: Even though the costs associated with the involvement of an intermediary in the process of contract stipulation are removed in smart contracts, this does not necessarily mean that smart contracts are cheaper than traditional contracts. Infrastructure necessary for their implementation and costs associated with the development ("drafting") of terms of smart contracts are still rather high. Although several efforts are currently carried out in order to reduce and hide the complexity behind the blockchain technology, developers with many technical skills are still needed and the number of professional blockchain developers is small compared to the market demand.

- H2: The tools for the expert users are still in an early stage, and standards for developing blockchain-based applications have not been defined yet. Because smart-contracts' development is prone to errors (**?**), productivity tools for expert users are crucial to improve the developers' work and to ease the adoption of the blockchain technology.

- H3: The interaction with the blockchain technology is still complex for non-expert users. Interacting with the blockchain requires many technical skills (e.g., mastering the concept of installing a wallet, access the Gas Oracles Information, the Gas Price, etc.). Moreover, many non-expert users associated the blockchain technology with a fraudulent investment because the cryptocurrency are very volatile or because there might be fraudulent smart contracts, such as the "smart Ponzi schemes". As a consequence, there is still a lot of misinformation on blockchain among non-expert users, and people could still prefer traditional applications rather than applications based on the blockchain technology. Therefore more user-centered tools should be provided to overcome these barriers that can limit the use of blockchain-based applications.

- H4: The knowledge about the blockchain technology should not reside only with the specialists who write smart contracts but should be distributed also among experts in different areas such as experts in law, finance, digital innovation and so on. There is the need for a holistic approach to the blockchain technology that includes an interdisciplinary model to connect experts in different domains. However, different research areas have different technical jargon used to communicate scientific knowledge: while computer scientists need to understand program code, law experts are not expected to understand the program code. To overcome this problem, graphic representation could be a means to facilitate the understanding of scientific knowledge between different disciplines. Figures and diagrams might be a communication medium among different disciplines' languages and expertises. Figures and diagrams not only

show the relevant data that support key research findings, but also provide visual information on the interactions among different operations required in scientific reasoning (**??**). Being able to adequately and precisely visualize data is also a pillar on which decisions can be made, as proposed by different dashboards in the market.

Overall, the blockchain technology still needs several improvements in different areas before becoming popular and adopted by many people and companies, both private and public, as it happened for the WWW. Based on these hypotheses, the dissertation is focused on the research and development of innovative methodologies, models and user-centered tools for expert and non-expert users that could help adopt the blockchain technology by a larger number of private and public companies. Furthermore, the dissertation proposes a holistic approach that can ease the interaction and collaboration among experts in different disciplines also around the blockchain technology.

## 1.6 Structure of the Dissertation

The dissertation is organized as follows:

Chapter 2 introduces the blockchain in details, presenting the existing technologies, mathematical techniques and algorithms connected to blockchain such as the cryptographic and the consensus algorithm (see Section 2.2.5). The formal definition of the blockchain, and in particular of the Ethereum blockchain, will be provided. Lastly, the concepts related to smart contracts (see Section 2.4) and the Oracles (see Section 2.5) will be presented. The purpose of this chapter is to provide the readers with some concepts useful to better understand the blockchain that might be unfamiliar to most of them. The readers should indeed be aware of the blockchain's ecosystem strengths, as well as its shortcomings to better evaluate the motivations and the hypotheses that guide the research. Also, in the following chapters, the blockchain characteristics will be exploited to design a user-center model (**?**) and to implement specific tools for both expert (**????**) and not-expert users (**???**).

Chapter 3 presents two applications aimed at expert users, i.e. users with specific technical knowledge in the blockchain, that should meet their demands and strengthen their abilities in the development of blockchain-based applications. The first application is named PASO (see Section 3.2), which is a web-based tool supporting static code analysis of the most used programming language to write smart contracts, i.e., Solidity. PASO differs from the existing applications because, just using a web browser, it is able to provide the expert users with software metrics values for smart contracts written in Solidity. The second application is named Smart-corpus (see Section 3.3), which is an organized repository of the Ethereum smart contracts' source codes and metrics. It is aimed at expert users, especially as academic researchers, who analyze smart contracts' code to improve the blockchain security, to find design defects and propose solutions. Based on the Smart-corspus,

the dissertation also proposes an analysis of code clones in Solidity (see Section 3.4), aimed to improve expert users' practices in developing smart contracts. Finally, the advantages and the disadvantages of code duplication are discussed, and some "best practices" are proposed to expert users to develop secure smart contracts.

Chapter 4 presents the most important factors that influence the blockchain transactions and thus the waiting times and the fees paid by non-expert users. First, the chapter focuses on the fees prediction provided by the Gas Oracles (see Sections 4.2 and 4.3), i.e. softwares that are supposed to suggest the best price in Gas units to pay by the users to execute their transactions. The aim of the chapter is to provide especially non-expert users with the most convenient fee to pay based on categories that reflects their needs.To this aim, the Gas Oracles' predictions are verified, showing that the Gas Oracles' predictions is less reliable than advertised by the Gas Oracles themselves. A new model for the Gas price prediction is proposed to meet the real needs of the users in terms of fees to pay and waiting time to execute their blockchain transactions. Second, the chapter analyses malicious smart contracts to provide non-expert users with a means to prevent transaction fraud. The malicious smart contracts are intended to steal money especially from non-expert users that have no technical skills to understand that such smart contracts advertise something other than what they actually do (see Section 4.5). An application is therefore proposed to automatically recognize malicious smart contracts by using AI techniques.

Chapter 5 is specifically dedicated to two models (and relative applications) based on graphical representation that are intended to facilitate the understanding of the blockchain underlying mechanisms. The first one is named Smart-Graph (see Section 5.2) and proposes a model that provides a graphical representations for smart contract on the Ethereum blockchain. The graphical representation provided by Smart-Graph highlights two relevant aspects for the blockchain software developers and companies: the costs and the maintainability of the source code. The second one is named Miró and proposes a comprehensive interdisciplinary model for graphical representation (see Section 5.3). The model integrates a data-driven approach with an approach that guides users with an expertise on a specific domain, to achieve the intended visualization, based on their aims, knowledge and hypotheses. The application can thus provide users with different expertises a means to collaborate in the blockchain environment.

Chapter 6 presents the conclusion of the dissertation. The limitations of the work are discussed, as well as the possibilities opened by the answers provided in each chapter to the main research questions. Finally, some perspectives for future research are described, focusing on what still remains needed for both expert and non-expert users.

## 1.7 List of Publications

This dissertation is an extended, revised, and enhanced version of the papers I authored and/or coauthored during the Phd program, consisting in a research activity that had some publications as an outcome. The list of the original publications is provided below.

For the part regarding the blockchain models to design tools for expert users:

- "An Organized Repository of Ethereum Smart Contracts' Source Codes and Metrics" (**?**). The paper has been published on the Journal "Future Internet", in 2020.
- "PASO: A Web-Based Parser for Solidity Language Analysis" (**?**). The paper has been published in the Conference Proceedings of 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE 2020) (https://www.computer.org/csdl/proceedings/iwbose/2020/1iES4LO1CKc).
- "Analysis of Source Code Duplication in Ethreum Smart Contracts" (**?**). The paper has been published in the Conference Proceedings of the 2021 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE 2021).

For the part concerning the blockchain models to design tools for non-expert users:

- "A User-Oriented Model for Oracles' Gas Price Prediction". The paper has been submitted on July 2020 to the Journal "Future Generation Computer Systems" and it is now in revision.
- "The influence factors on ethereum transaction fees" (**?**). The paper has been published in the Conference Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2019) (https://www.computer.org/csdl/proceedings/wetseb/2019/1d9UmyDgc6I).
- "Are the Gas Prices Oracle Reliable? A Case Study using the EthGasStation" (**?**). The paper has been published in the Conference Proceedings of the International Workshop on Blockchain Oriented Software Engineering (IWBOSE 2020)
- "Evaluating Machine-Learning Techniques for Detecting Smart Ponzi Schemes". The paper has been published in the Conference Proceedings of the 2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).

For the part dedicated to the visualization-based models that can be applied to the blockchain for both expert/non-expert users:

- "An Interdisciplinary Model for Graphical Representation" (**?**). The paper has been published in "Lecture Notes in Computer Science", in the series entitled "Formal Methods for Software Engineering: Languages, Methods, Application Domains" (LNCS, volume 12524) (**??**) and it is also part of the "Programming and Software Engineering" book sub-series (LNPSE, volume 12524).

- "Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain" (**?**). The paper has been published in the Conference Proceedings of the 2021 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE 2021).

# Chapter 2

# Background

## 2.1 Introduction

The chapter aims to provide the reader with the relevant information and background knowledge on blockchain technology required to better understand the topics discussed in this dissertation. Indeed the dissertation is based on the concepts presented in this chapter. It is therefore important especially for the readers who are not familiar with blockchain and blockchain technology.

Previous studies (**???**) have suggested that blockchain technology presents both strengths and shortcomings. Some of the strengths are considered to be the decentralization, tamper-proof, and smart contract. Anyway, some research (**?**) warns that the current blockchain ecosystem is still immature, with known (and possibly also unknown) flaws. The shortcomings of the blockchain technology will be addressed to design tools for both expert (see Chapter 3) and non-expert users (see Chapter 4). Blockchain technology and its use in different areas, such as healthcare (**??**), finance (**?**), government (**?**), and academic research (**???**), are considered in this work to design an interdisciplinary model that enables the users to share and receive knowledge from different sources. The interdisciplinary model is based on graphical representations aimed to clarify, interpret and analyze data coming from different sources (see Chapter 5).

Readers should therefore be aware of blockchain's ecosystem strength and shortcomings to better understand the motivations that guide this research work. The blockchain is a combination of already existing technologies, mathematical techniques and algorithms, such as hash functions (**?**), digital signatures (**?**), cryptography (**?**), Peer-to-Peer (P2P) network (**?**), consensus mechanism (Proof of Work and Proof of Stack) (**?**). First, the technologies connected to blockchain will be presented 2.2. Second, the basic components of blockchain (transaction, block, block header, and the chain), its operations (verification, validation, and consensus model), the underlying algorithms, and essentials of trust (hard fork and soft fork) 2.3 will be introduced. Third, the concepts of smart contracts and Solidity, which is the

most widespread programming language used to write smart contracts, will be presented (see 2.4). Finally, the chapter will present the blockchain oracles, which can be defined as a third-party service that takes raw data either from external sources, such as weather service, news, banking systems, or from internal sources, i.e. the blockchain itself (**??**). The Oracles process the collected raw data to extract information and provide it to smart contracts or the users who execute transactions in the blockchain. 2.5.

## 2.2 Technologies Connected to Blockchain

This section presents the basic concepts one must get acquainted with in order to understand the blockchain technology. Blockchain can be understood as a combination of already known technologies, including cryptographic components such as hashing, private and public key infrastructures, consensus processes, and elements of decentralized systems (**?**).

### 2.2.1 Hash

Blockchain technology relies extensively on hash functions.

**Definition 2.2.1** (Hash)**.** A hash function is a function $h$ with the following properties.

- Compression: $h$ maps an input $x$ of arbitrary finite length, to an output $h(x)$ of fixed length $n$.
- Easy of computation: Given $h$ and an input $x$, $h(x)$ is easy to compute.
- Preimage resistance (one-way): for all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find $x$ such that $y = h(x)$ given $y$ for which $x$ is not known.
- Collision resistance (strong collision resistance): it is computationally infeasible to find any two distinct inputs $x, x'$ which hash to the same output, i.e., such that $h(x) = h(x')$.

The output of such function is called a hash. In Definition 2.2.1, easy is purposefully left undefined as its meaning is context-dependent. It can be defined with regards to time, number of operations, or complexity for instance.

A hash is the result of a transformation of the original information that serves as input. This collision resistance is characterized by the fact that the same hash value is obtained from the same data and even a slight difference in the original data will result in a completely different hash value. Taking advantage of such characteristics, this mechanism is used for the detection of falsification of data. Figure 2.1 illustrates the compression property of hash functions.

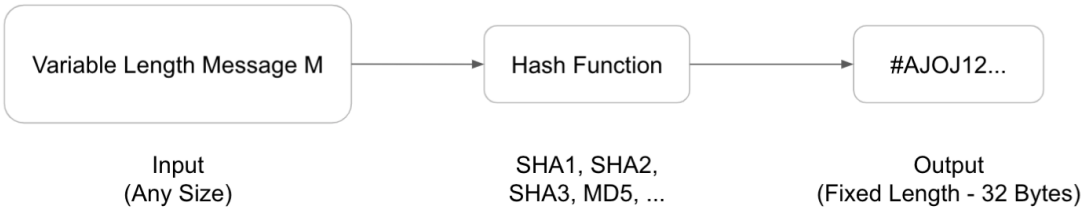



Figure 2.1: Hash function.

### 2.2.2 Digital Signatures

Digital signatures are the equivalent of the handwritten signature one might find at the bottom of a document. Like their paper-based counterparts, digital signatures must be:

- Unforgeable: No one must be able to produce a valid signature for an entity $S$, except $S$ themselves;
- Verifiable: Anyone with the correct information must be able to associate a signature with its emitter;
- Non-repudiable: A signer S cannot successfully dispute the origin of their signature.

A digital signature scheme is composed of three algorithms:

- Key Generation Algorithm: A method for generating a public/private key pair used for signing.
- Signing Algorithm: A method for producing a digital signature.
- Verification Algorithm: A method for verifying that a digital signature is authentic (i.e., was indeed created by the specified entity).

Figure 2.2 shows the digital signature composed of two parts: the signature generation process via the private key and the signature verification process via the public key.

- Key Generation: A digital signature scheme requires two keys. The first key is a private key and it is used to sign the message. The second key is public and it is used to verify the signature. In order to produce a verifiable signature, a signer $S$ must therefore first generate a key pair (private key, public key) and make the public key available to those who will later verify their signature. Anyone that comes into possession of $S$'s private key can sign in their name. The security of the scheme then rests on the security of the private key.

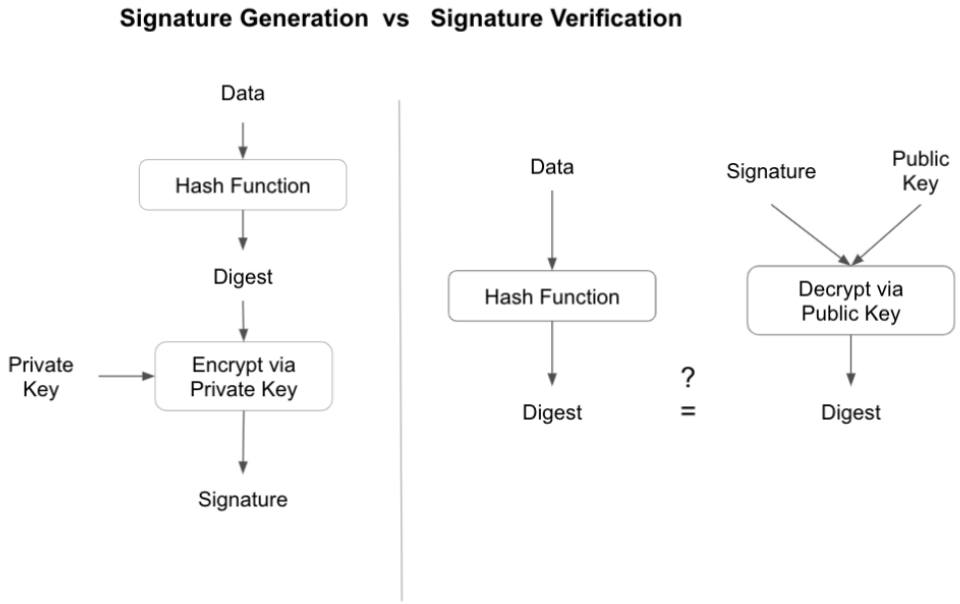



Figure 2.2: Digital signature process.

- Signature: A signing algorithm takes two inputs, the message and the private key, and outputs a signature. An example of signature scheme is to encrypt the cryptographic hash of the message with the private key. Anyone with the public key can decrypt the signature, recompute the message's hash, and compare the two.

- Verification: The verification algorithm takes the public key and the signature as inputs and outputs a boolean. Some schemes enable the retrieval of the original message from the signature. When it is not the case, the verification algorithm requires the original message as an additional input. The verification process confirms that the message has indeed been signed by the private key associated to the inputted public key. Ownership of these keys however must be proven through other means.

### 2.2.3 Merkle Trees

Merkle Tree also known as "hash tree" is a data structure in cryptography in which each leaf node is a hash of a block of data, and each non-leaf node is a hash of its child nodes. The benefit of using the Merkle Tree in blockchain is that instead of downloading every transaction and every block, a "light client" can only download the chain of block headers.

If a user needs to verify the existence of a specific transaction in a block, then he does not have to download the entire block. Downloading a set of a branch of this tree which contains this transaction is enough. We check the hashes which are just

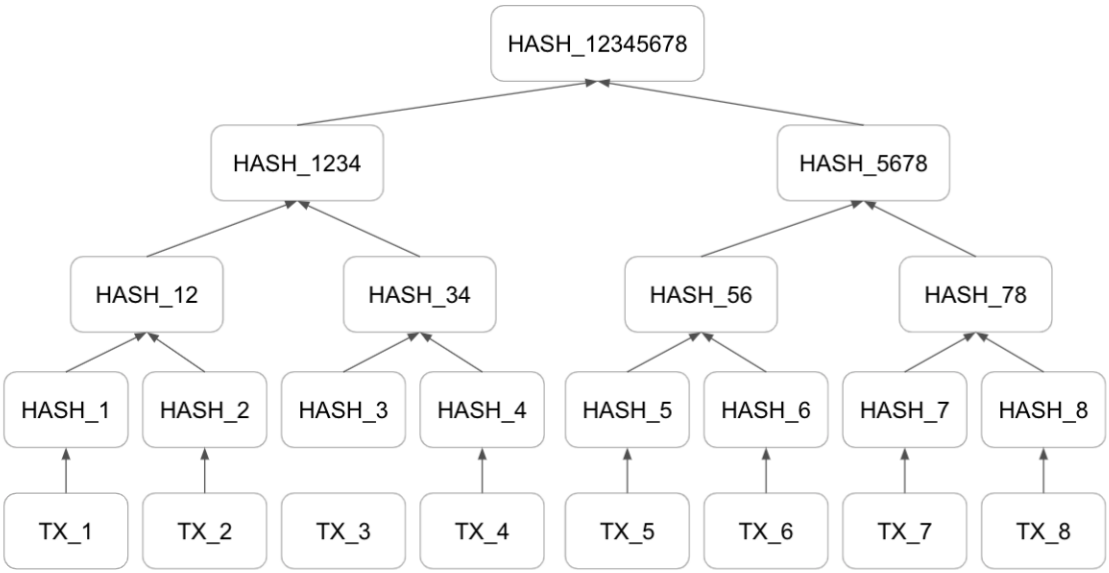



Figure 2.3: Merkle Tree.

going up the branch (relevant to my transaction). If these hashes check out good, then we know that this particular transaction exist in this block.

The tree is then constructed by labelling each non-leaf node with the cryptographic hash of its child nodes. A tree is usually smaller than the data it represents thanks to the compression property of hash functions. Hash trees also encode data in a privacy-preserving fashion as no information from the original data can be recovered from the tree thanks to the preimage resistance of cryptographic hash functions. Figure 2.3 illustrates the encoding of data into a Merkle tree. The original data blocks (the transactions) are not part of the tree. A data block can represent part of a larger file, or be an element in a set.

### 2.2.4 State Machine

A finite-state machine (FSM) or simply a state machine is used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of user-defined states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

Figure 2.4 shows a simple state machine above. It consists of two states, Off and On. On is the initial state here; it is activated when the state machine is executed. The arrows between the states denote the possible state transitions. They define for

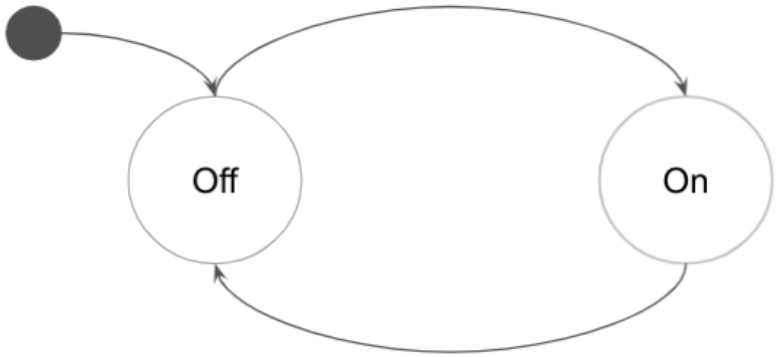



Figure 2.4: A simple state machine.

which input a state change occurs. Here, the active state is changed from On to Off for the input button pressed, and back again to On for the same input.

arrow repersent transitions

Formula 2.1 formally denotes a finite state machine (FSM) represented as a tuple of six values:

$$FSM = (I, O, S, s_0, \delta, \lambda) \tag{2.1}$$

- I: finite set of inputs
- O: finite set of outputs
- S: finite set of states
- $s_0 \in S$ : initial state
- $\delta : S \times I \rightarrow S$: transition function (new state = current state + input value)
- $\lambda$: output function

### 2.2.5 Consensus Algorithm

One of the most important challenges of the blockchain is to keep the different copies of the blockchain consistent with one another. This is achieved through a consensus algorithm.

There exist two main types of such protocols currently in use: Proof of Work (PoW) and Proof of Stake (PoS). All aim at electing a leader that will propose the next block to the network. Other propositions have some merits such as hybriding PoW and PoS to mitigate the shortcomings of both, or basing the protocols on social interactions. All of these protocols are explained below.

### Proof of Work (PoW)

The proof of work (PoW) consensus mechanism is the widest deployed consensus mechanism in existing blockchains. PoW was introduced by Bitcoin (**?**) and assumes that each peer votes with his "computing power" by solving proof of work instances and constructing the appropriate blocks. Bitcoin, for example, employs a hash-based PoW which entails finding a nonce value, such that when hashed with additional block parameters (e.g., a Merkle hash and the previous block hash), the value of the hash has to be smaller than the current target value. When such a nonce is found, the miner creates the block and forwards it on the network layer to its peers. Other peers in the network can verify the PoW by computing the hash of the block and checking whether it satisfies the condition to be smaller than the current target value.

PoW's security relies on the principle that no entity should gather more than 50% of the processing power because such an entity can effectively control the system by sustaining the longest chain. We now briefly outline known attacks on existing PoW-based blockchains. First, an adversary can attempt to double-spend by using the same coin(s) to issue two (or more) transactions—thus effectively spending more coins than he possesses. Recent studies have shown that accepting transactions without requiring blockchain confirmations is insecure (**?**).

To date (May, 2021), the security of the Ethereum blockchain relies on this PoW system, which inherently means that a block cannot be modified without redoing the work spent on it, including the work spent on blocks chained after it. Therefore, an attacker will be outpaced by honest miners as long as majority of the overall computation power participating in the Ethereum network are controlled by honest miners. In this case, a block recorded in the blockchain is almost impossible to modify.

Buterin (**?**) argues that there are some downturns with a PoW consensus protocol, e.g. the risk of a 51% attack and there are high-energy costs of producing one block. Courtious argues that the proof-of-work protocol is heading towards self-destruction. The mining community is getting smaller and more specialized, where big companies with great resources could outwork the individual miner (**?**). This specialization of mining is making the system more centralized to a few big companies and the risk of a 51% attack increases.

### Proof of Stake (PoS)

To reduce the risk of a 51% attack and to reduce energy consumption, a new consensus protocol was introduced within the blockchain community, called proof-of-stake. Instead of proving that a node solved a computational hard task, like one does in the proof-of-work protocol, the node could instead proof it has a certain amount of coins (**?**). In the case of proof-of-stake it takes coins to create a new block, not computational power and the node with the most coins, gets the most influence (**?**).

In its most basic form, PoS works as a lottery. Each second, each account has a certain probability to be chosen to mine a new block that is proportional to its balance. To mine a block, one must simply sign it. Letting a single participant sign is dangerous as we want a system that a single player cannot dominate. PoS therefore requires p validators per block. The lottery system selects more than p winners as some winners may not be online at that moment, or the key associated to the selected funds may have been lost.

This method has some advantages over PoW. First, it is more eco-friendly. Secondly, PoS addresses PoW's centralization concerns evoked above. The block generation can be much faster in PoS. The scheme can arguably be considered more secure as a 51% attack (see Section 2.8.4) requires owning 51% of the currency. But it cannot be used in this simplest form as many issues need to be addressed (**?**).

## 2.3 Blockchain

Blockchain technology is a distributed database, which maintains an immutable public ledger of all the transactions. Blockchain allows for the time stamped recording of all the transactions. These finger prints are saved in groups called "block". The individual blocks are then linked in a chain of blocks and each subsequent block has a digital token from the previous block. Thus, it becomes impossible to modify the information in an old block in the chain without modifying the subsequent blocks. The main idea behind the blockchain technology is to register, confirm and transfer all kinds of contracts and properties without the need of any intermediary (**?**).

### 2.3.1 Transactions and Addresses

Formula 2.2 formally denotes the transaction as a tuple of seven values.

$$T = (nonce, gasPrice, gasLimit, to, value, signature, init) \tag{2.2}$$

The fields have the following meaning:

- nonce: it is a counter that indicates the number of transactions sent from the account. This ensures transactions are only processed once. In a contract account, this number represents the number of contracts created by the account.
- gasPrice: it represents the amount in Ether to be paid for one unit of Gas consumed.
- gasLimit: it is the total amount of "EVM work" that the transactions inside the block can equate to.
- to: It represents the 160-bit address of the message call's recipient.

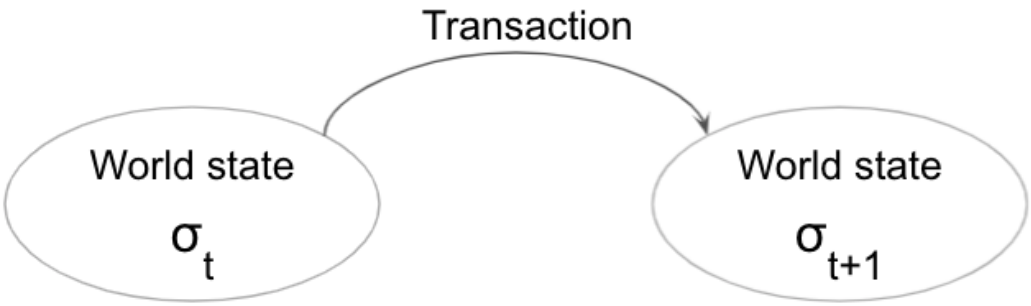



Figure 2.5: Ethereum can be viewed as a transaction-based state machine.

- value: it is a scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.

- signature: the identifier of the sender. This is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction.

- init: it is an unlimited size byte array specifying the EVM-code for the account initialisation procedure.

Ethereum can be viewed as a transaction-based state machine. An Ethereum transaction refers to an action initiated by an account. For example, if Bob sends Alice 1 ETH, Bob's account must be debited and Alice's must be credited. This state-changing action takes place within a transaction. Figure 2.5 depicts a change to world state via a transaction. A transaction represents a valid arc between two states.

In essence, a blockchain is a transaction ordering mechanism. These transactions can describe a transfer of assets (cryptocurrency or other), or an interaction with a smart contract. Transactions must always be signed by their emitter. They can be prepared offline and are then broadcasted to the peer-to-peer network that composes the blockchain. Miners will verify them, before bundling them into a block. A transaction is refused if its digital signature is invalid, if it conflicts with the blockchain history, or if it otherwise breaks blockchain rules. Their specific format is implementation-dependent.

In order to send or be the recipient of transactions, a user must have a blockchain address. To create an address, a public/private key pair is generated for the user. The public key is hashed to create the user's blockchain address. The private key is used to sign outgoing transactions. One can create as many addresses as they want. Each new address acts as a new pseudonym. This increases the user's privacy

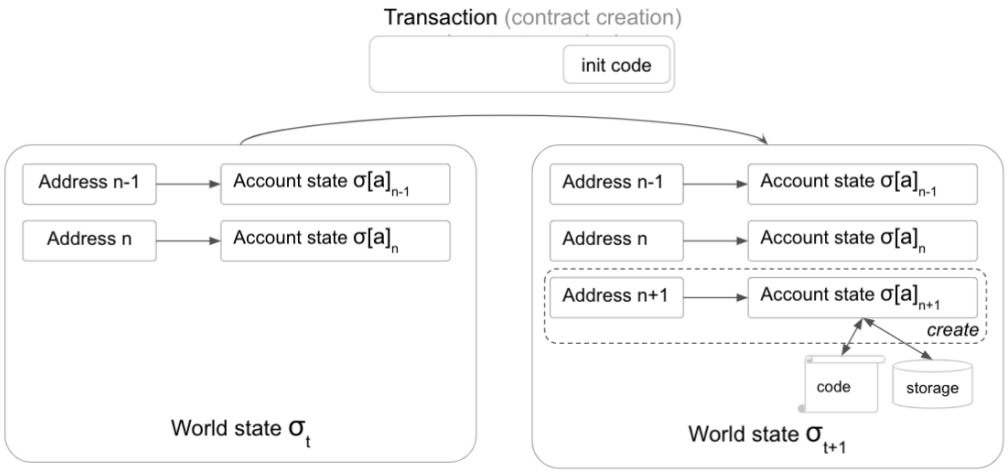



Figure 2.6: Creation of new accounts with associated code (known informally as "contract creation").

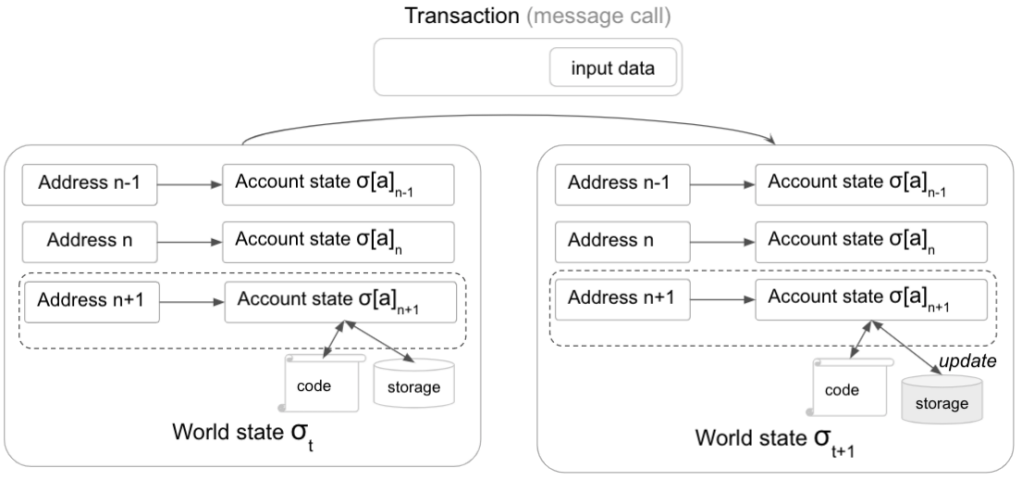



Figure 2.7: Message call transaction which can update the storage.

by making it more difficult for people to trace their activity. Smart contracts have addresses that are used to invoke their functions.

There are two practical types of transaction, contract creation and message call. The "contract creation" changes the world state by adding a new account. The "message call" changes the world state by changing the account state. Figures 2.6, 2.7 illustrates the difference between these two types of transactions.

### 2.3.2 World State

The Ethereum world state is the state of the overall system. The state of all transactions on the blockchain is represented via the "world state". The world state can be thought of as a mapping between address (a 160-bit identifiers) and account state

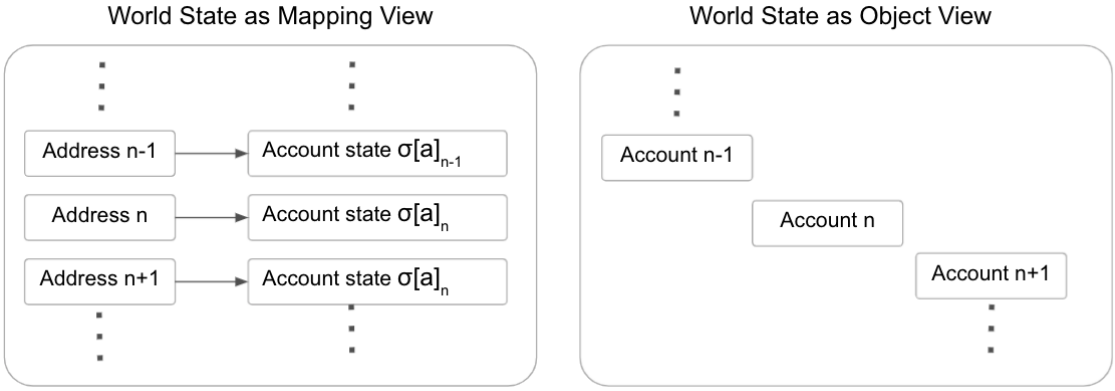



Figure 2.8: "World state" as the mapping view and "world state" as the object view.

or a set of account. By using this view the account is an object in the world state and a mapping between the address and the account state. Figure 2.8 represents two views of world state: the mapping view and the object view.

The account state, $\sigma[a]$, is a tuple holding four information: the balance $\sigma[a]_b$), the nonce $\sigma[a]_n$, the storage hash $\sigma[a]_s$, and the code hash $\sigma[a]_c$. Formula 2.3 formally denotes the account state.

$$\sigma[a] = (\sigma[a]_n, \sigma[a]_b, \sigma[a]_s, \sigma[a]_c) \tag{2.3}$$

The balance ($\sigma[a]_b$) is a scalar value equal to the number of Wei owned by the address $a$.

The nonce ($\sigma[a]_n$) is a scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address "a" in state $\sigma$, this would be formally denoted $\sigma[a]_n$.

The storage hash ($\sigma[a]_s$) is a 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted $\sigma[a]_s$.

The code hash ($\sigma[a]_c$) is hash of the EVM code of this account. This is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted $\sigma[a]_c$, and thus the code may be denoted as b, given that $KEC(b) = \sigma[a]_c$.

The account can be divided in two types: the external owned account (EOA) and the contract account. Figure 2.9 represent the two type of account for the Ethereum blockchain. The external owned account (EOA) is controlled by a private key and

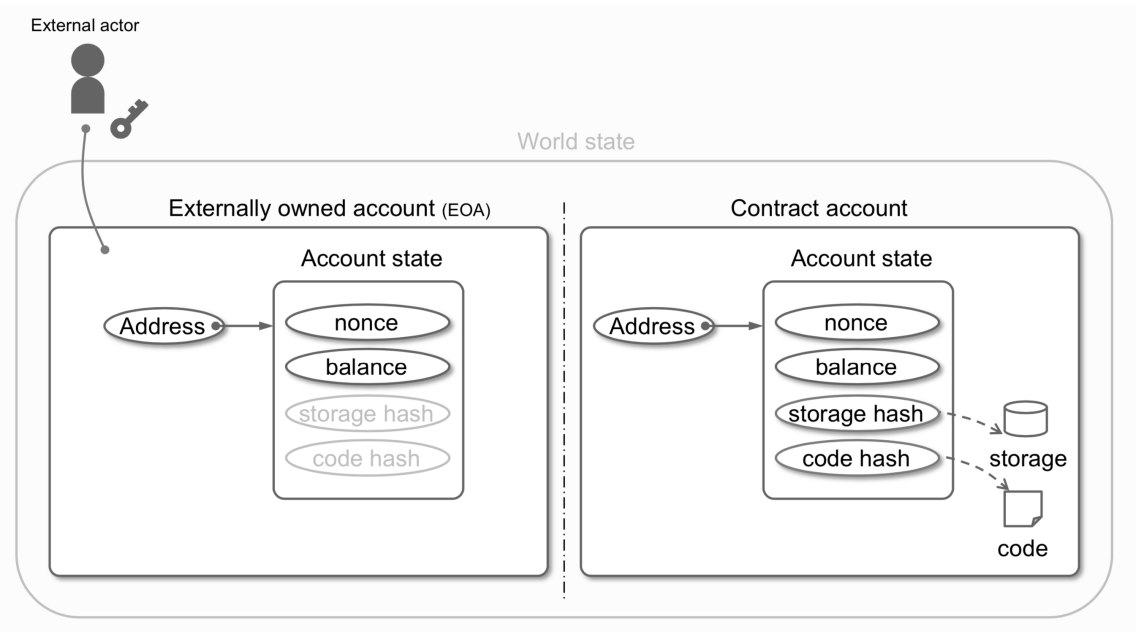



Figure 2.9: The external owned account (EOA) is controlled by a private key and cannot contain EVM code. In contrast, the account contract contains EVM code.

cannot contain EVM code. In contrast, the account contract contains EVM code.

A block is an assembly of ordered transactions. Transactions are collated into blocks. A block can be thought of as a package of data. From the viewpoint of the states, Ethereum can be seen as a state chain. Figure 2.10 display as from the viewpoint of the implementation, Ethereum can also be seen as a chain of blocks, so it is "blockchain".

The blockchain is a linked list of confirmed blocks, such that each block $B_t$ references its immediate predecessor $B_{t-1}$, where the subscript indicates the discrete time index of block confirmation. The only block that does not have a predecessor is the "genesis" block $B_0$. Thus we can schematically refer to a blockchain as:

$$B_0 \cup B_1 \cup \cdots B_t \cup B_{t+1} \cup \cdots$$

A block has a finite capacity for recording transactions. Each Ethereum block has a maximum size, which limits the amount of data that can be included. The current maximum block size is set at 12.5M Gas, where a Gas is a special unit for running a transaction or contract in Ethereum. Currently, the maximum block size is set to $12,500,000$ Gas. If we consider only transactions of $21,000$ Gas, a block can contain at maximum 595 transactions.

It is composed of a header, and a body. The body contains the transactions and a Merkle tree formed by their hashes. The header contains the block's meta data, including the root of the Merkle tree.

Blocks are linked together to form a chain. The first block is called the genesis

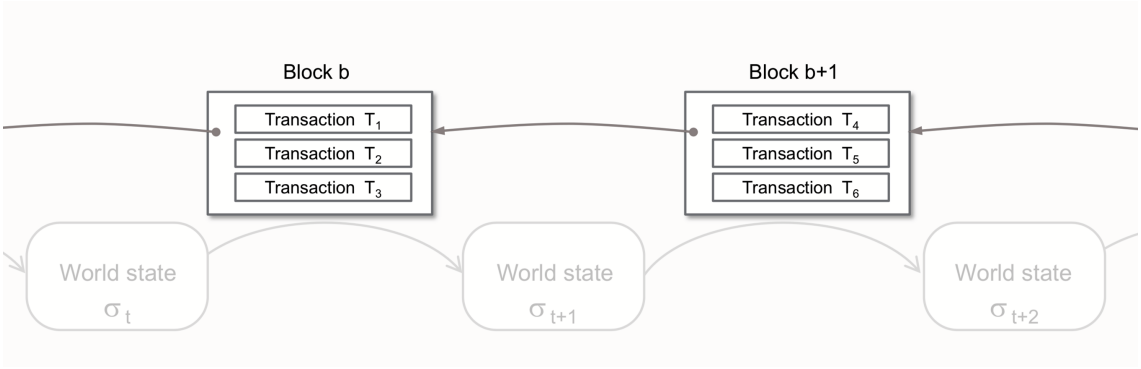



Figure 2.10: Ethereum can be seen as a chain of blocks.

block or block 0. Blocks are identified either by the hash of their header, or by their height, i.e. the distance between them and the genesis block. The chain is created by including the hash of block $n-1$ in the header of block $n$. So in Figure 2.10, block $11's$ header contains the hash of block 10 while block 12 contains the hash of block 11.

Any modification of the content of a block modifies its hash. If an attacker modifies the block $i$, its hash no longer equates the value stored within block $i+1$. One can therefore notice the modification occurred by comparing the two values. To hide this, the attacker needs to also modify block $i+1$, which would force them to modify block $i+2$ to replace the now modified hash, and so on, and so forth, until they reach the most recent block. Blockchains are set up so that producing a valid block is hard. For each block stacked on top, the task becomes even harder for an attacker. This is the heart of the tamper-proof nature of the blockchain, along with its distributed architecture.

A transaction is considered as verified when it has been included in a block. Because the last few blocks are subject to modification (see Section 2.8.1), users should wait until a transaction is buried under several blocks before considering it as definitively integrated into the blockchain. Each new block added to the chain on top of a given block diminishes the chances that this block will be removed.

### 2.3.3 Miners

Some nodes in the network dedicate resources to verifying transactions and maintaining the security of the blockchain. They are called miners. Miners are paid by block rewards. For each new block, a block reward is awarded to a single miner or a small group of them. This reward system is the only way new coins can be created in the system. Transactions can (and usually do) include a transaction fee. The sum of all of the block's transaction fees included in a block are added to that block's reward.

The specifics of block rewards are implementation-dependent. In Bitcoin for

instance, the block reward was originally of 50 BTC per block and halves every 210 000 blocks, which take roughly 4 years to mine. At the moment, it is therefore down to 12.5 BTC per block. Transactions fees supplement this lost in income. In Ethereum, miners also have to run smart contracts. Users must include a fee proportional to the difficulty of the operation. Block rewards are only a few ETH, but blocks are produced every few seconds rather than every 10 minutes.

Miners compete to receive the reward by participating in the consensus protocol. The more miners participate, the more overall resources are poured into the consensus, the harder it is for an individual to single-handedly match that and gain too much influence over the blockchain.

### 2.3.4 Forks and Longest Chains

Since the system is decentralized and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks. In order to form a consensus as to which path, from root (the genesis block) to leaf (the block containing the most recent transactions) through this tree structure, known as the blockchain, there must be an agreed-upon scheme.

Sometimes, two miners will find a block at roughly the same time. These two valid blocks have the same parent block and therefore correspond to the same place in the chain. There is often no logical reason to prefer one over the other. The network is therefore presented with two alternate but equally valid versions of history. This is called a fork.

This is where the notion of longest chain comes into play. Rather than a chain, a blockchain is effectively a tree with the genesis block as its root. Miners only work on the longest path (or longest chain) from root to leaf and it is the only valid version of the blockchain history. The definition of longest chain varies with the blockchain and the consensus protocol it uses. It can simply be the path with the most blocks, or, for Proof of Work (see Section 2.6.1), additionally take the puzzle difficulty of each block into account.

Forks are solved over time. When the next block is created, the block it chooses as its ancestor becomes part of the longest chain and the concurrent blocks are dropped. This is illustrated in Figure 2.11. Forks occurrence rate and resolution time depend on the block rate, network size, and consensus protocol.

### 2.3.5 Ethereum Virtual Machine (EVM)

The EVM handles the computation and state of contracts and is build on a stack-based language with a predefined set of instructions (opcodes) and corresponding arguments. So, in essence, a contract is simply a series of opcode statements, which are sequentially executed by the EVM. The EVM can be thought of as a global decentralized computer on which all smart contracts run. Although it behaves like one giant computer, it is rather a network of smaller discrete machines in constant

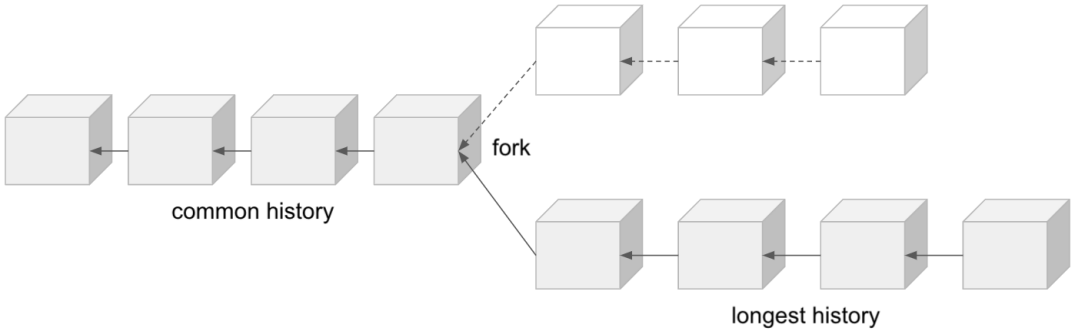



Figure 2.11: Blockchain Fork.

communication. All transactions, handling the execution of smart contracts, are local on each node of the network and processed in relative synchrony. Each node validates and groups the transactions sent from users into blocks, and tries to append them to the blockchain in order to collect an associated reward. This process is called mining and the participating nodes are called miners. To ensure a proper resource handling of the EVM, every instruction the EVM executes has a cost associated with it, measured in units of gas.

### 2.3.6 Ether

In order to incentivize computation within the blockchain network, there needs to be an agreed method for transmitting value. To address this issue, Ethereum has an intrinsic currency, Ether, known also as ETH. Any participant who broadcasts a transaction request must also offer some amount of Ether to the network, as a bounty to be awarded to whoever eventually does the work of verifying the transaction, executing it, committing it to the blockchain, and broadcasting it to the network.

The amount of Ether paid is a function of the length of the computation. This also prevents malicious participants from intentionally clogging the network by requesting the execution of infinite loops or resource-intensive scripts, as these actors will be continually charged.

The smallest sub denomination of Ether, and thus the one in which all integer values of the currency are counted, is the Wei. One Ether is defined as being $10^{18}$ Wei. Table 2.1 shows the names of the currency used within Ethereum.

The names of the currency used within Ethereum correspond to famous mathematicians who contributed to the invention of the blockchain.

- Wei Dai is a computer engineer who formulated the concepts of all modern cryptocurrencies, and is best known as the creator of the predecessor to Bitcoin, B-money.

Table 2.1: Names of the currency used within Ethereum

| Name | Unit | Wei Value | Wei |
|---|---|---|---|
| **Wei** | **wei** | 1 wei | 1 |
| **Babbage** | **Kwei** | $10^3$ wei | 1,000 |
| **Lovelace** | **Mwei** | $10^6$ **wei** | 1,000,000 |
| **Shannon** | **Gwei** | $10^9$ wei | 1,000,000,000 |
| **Szabo** | **microEther** | $10^{12}$ **wei** | 1,000,000,000,000 |
| **Finney** | **milliEther** | $10^{15}$ wei | 1,000,000,000,000,000 |
| **Ether** | **Eher** | $10^{18}$ **wei** | 1,000,000,000,000,000,000 |

- Charles Babbage was a mathematician, philosopher, inventor, and mechanical engineer who designed the first automatic computing engines.
- Ada Lovelace was a mathematician, writer, and computer programmer; she published the first algorithm.
- Claude Shannon was an American mathematician, cryptographer, and cryptoanalysis guru, who is known as "the father of information theory".
- Nick Szabo computer scientist, legal scholar, and cryptographer known for his pioneering research in digital contracts and digital currency.
- Hal Finney, a computer scientist, and cryptographer; he was one of the early developers of Bitcoin, and alleged to be the first human to receive a bitcoin from Satoshi Nakamoto, the named founder of Bitcoin.

Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to successfully conduct a transaction on Ethereum. The Gas fees are paid in Ethereum's native currency, ether (ETH). Gas prices are denoted in GWei, which itself is a denomination of ETH; each Gwei is equal to 0.000000001 ETH ($10^{-9}$ ETH).

By requiring a fee for every computation executed on the network, the Ethreum blockchian prevents users from spamming the network. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "Gas". Although a transaction includes a limit, any gas not used in a transaction is returned to the user.

The Gas limit refers to the maximum amount of Gas a user is willing to consume on a transaction. A higher Gas limit means more computational work can be done while interacting with smart contracts. A standard ETH transfer requires a gas limit of $21,000$ units of gas.

For instance if a user set a Gas limit of $91,000$ for a simple ETH transfer, the EVM would consume $21,000$, and the user would get back the remaining $70,000$.

However, if a user specifies too little Gas say for example, a gas limit of 20,000 for a simple ETH transfer, the EVM will consume 20,000 Gas units attempting to fulfill the transaction, but it will not complete. The EVM then reverts any changes, but since 20,000 Gas units worth of work has already been done by the miner, that Gas is lost by the user.

Table 2.2 shows the Gas cost of some smart contract operations in Ethereum blockchain.

Table 2.2: Gas cost of some smart contract operations in Ethereum blockchain.

| **Operation** | **Gas** | **Description** |
|---|---|---|
| ADD/SUB | 3 | Arithmetic operation |
| MUL/DIV | 5 | |
| ADDMOD/MULMOD | 8 | |
| AND/OR/XOR | 3 | Bitwise logic operation |
| LT/GT/SLT/SGT/EQ | 3 | Comparison operation |
| POP | 2 | Stack operation |
| PUSH/DUP/SWAP | 3 | |
| JUMP | 8 | Unconditional jump |
| JUMPI | 10 | Conditional jump |
| SLOAD | 200 | Storage operation |
| SSTORE | 5000 or 20000 | |

### 2.3.7 Memory-Pool

The Memory-Pool is the name given to the set of valid transactions that the miner is aware of but that have not yet been included in a block. Valid transactions sent to an Ethereum node should enter the Memory-Pool. But there's not actually a single Memory-Pool. Rather, each node has its own Memory-Pool that attempts to remain synchronized with other nodes (peers) over the Ethereum network. Since network communication is not always reliable or timely, each node has a slightly (or sometimes significantly) different Memory-Pool. Also, nodes have different rules about which transactions they accept (e.g. minimum gas price and Memory-Pool size limits).

Ideally transactions leave a node's mempool because they are included in a block. But they can also leave because they are replaced via a speedup/cancel, or dropped due to the node's mempool configuration.

## 2.4 Smart Contract

A smart contract is a digitally signed, computable agreement between two or more parties. A virtual third party, a software agent, can execute and enforce (at least some of) the terms of such agreements. In the context of the blockchain, where it

truly takes it sense, a smart-contract is an event-driven program, with state, that runs on a replicated, shared ledger and which can take custody over assets on that ledger. smart contracts on the blockchain, created by computer programmers, are entirely digital and written using programming code languages. This code defines the rules and consequences in the same way that a traditional legal document would, stating the obligations, benefits and penalties, which may be due to either party in various different circumstances. The big difference is that this code is automatically executed by a distributed ledger system, in a non-repudiable and unbreakable way. Smart contract code has some unique characteristics:

- Deterministic: Since a smart contract code is executed on multiple distributed nodes simultaneously, it needs to be deterministic i.e. given an input; all nodes should produce the same output. That implies the smart contract code should not have any randomness; it should be independent of time (within a small time window because the code might get executed a slightly different time in each of the nodes); and it should be possible to execute the code multiple times.

- Immutable: smart contract code is immutable. This means that once deployed, it cannot be changed. This of course is beneficial from the trust perspective but it also raises some challenges (e.g. how to fix a code bug) and implies that smart contract code requires additional due diligence/governance.

- Verifiable: Once deployed, smart contract code gets a unique address. Before using the smart contract, interested parties can and should view or verify the code.

### 2.4.1 Programming Languages for Smart Contracts

Although ultimately all Ethereum smart contracts are deployed as EVM bytecode, the bytecode is rarely directly written. The most popular programming language Solidity (**?**) has a rich syntax but no specification. The only definition of Solidity is the Solidity compiler implementation, which compiles Solidity programs into EVM bytecode. The Solidity compiler is written in C++.

Because smart contracts work like computer programs, it is very important that they do exactly what the parties want them to do. This is achieved by inputting the proper logic when writing a smart contract. The code behaves in predefined ways and does not have the linguistic nuances of human languages, thus, it has now automated the "if this happens then do that" part of traditional contracts.

### 2.4.2 Solidity Grammar

A programming language is a set of commands, strings of characters that is both readable by programmers and automatically translatable into machine code. It has

grammar and semantic rules. The grammar is a set of rules that define how the commands have to be arranged to make sense and to be correctly translated to the machine code. Semantics is a set of meanings assigned to every command of the language and it is used to correctly translate the program to machine code (**??**).

Figure 18 shows a piece of Solidity grammar according to the ANTLR (ANother Tool for Language Recognition) rules. Each ANTLR rule consists of a name, followed by a colon, followed by its definition, and terminated by a semicolon. The "sourceUnit" symbol is the entry node of the grammar. Nonterminal nodes in ANTLR have to be lowercase. Terminal nodes have capitalized names, like EOF (end-of-file). EOF is a special terminal node, defined by ANTLR, meaning the end of the input. In particular, it stands for the end of the file, even though the input may also come from a string or a network connection rather than just from a single file.

The symbol "|" represents the alternation operator, the symbol "*" is the repetition operator, and parentheses are used for grouping, in the same way we have been using for a natural language grammar reading. Optional parts can be marked with the symbol "?". In ANTLR, terminal nodes can be defined using regular expressions, but fixed strings are not permitted. For example, here are some Terminal patterns used in the Solidity grammar written in ANTLR syntax and with ANTLR naming convention: COMMENT → "/*" ".*?" "*/".

The Solidity grammar definition is available at this web site. [1]

```
1  grammar Solidity;
2
3  sourceUnit
4    : (pragmaDirective | importDirective | contractDefinition)*
       EOF ;
5
6    ...
7
8  functionDefinition
9    : natSpec? 'function' identifier? parameterList modifierList
       returnParameters? ( ';' | block ) ;
10
11 returnParameters
12   : 'returns' parameterList ;
13
14 modifierList
15   : ( modifierInvocation | stateMutability | ExternalKeyword
16     | PublicKeyword | InternalKeyword | PrivateKeyword )* ;
```

Listing 2.1: This code shows how the Solidity grammar looks like as an ANTLR source file

[1] https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

## 2.5 Blockchain Oracles

Blockchain oracles are trusted entities which sign claims about the state of the world. The general function of a blockchain oracle is similar to those that exist in mythology. They were the source of information that was beyond peoples' comprehension. In the same vein, blockchains do not have direct access to information residing outside the chain. Because of this, there is no discernible method of validation for the conditions that smart contracts draw from.

Oracles provide the data that is necessary for triggering smart contracts to execute upon the meeting of the original terms. The association of these conditions with smart contracts can be a variety of things. These include temperature, the completion of payment, changes in price, and numerous others. Basically, oracles are what allow smart contracts to interact with external data. This data comes from software (big-data application) and hardware (Internet-of-Things). As previously mentioned, the conditions can be any data, however, the smart contract does not wait for external data to go into the system. Instead, the contract needs to be enforced, meaning one must spend network resources for calling external data. This consequently activates network transaction costs. When it comes to Ethereum, this is "Gas".

### 2.5.1 Gas Oracles

Ethereum uses the concept of "Gas" to discourage over-consumption of resources (e.g., a contract program that causes miners to loop forever). The user who creates a transaction must spend currency to purchase Gas. During the execution of a transaction, every program instruction consumes some amount of Gas. If the gas runs out before the transaction reaches an ordinary stopping point, it is treated as an exception: the state is reverted as though the transaction had no effect, but the Ether used to purchase the Gas is not refunded. When one contract sends a message to another, the sender can offer only a portion of its available gas to the recipient. If the recipient runs out of Gas, the control returns to the sender, who can use its remaining Gas to handle the exception.

# Chapter 3

# Blockchain Models to Design Tools for Expert Users

## 3.1 Introduction

In the previous chapter, we presented the background information on the blockchain technology used to design and develop the models and tools presented in this dissertation. This chapter is focused on the design and development of blockchain-based applications for expert users.

Based on market studies (**??**) and academic research (**???**), expert users expect to develop high-quality, defect-free, and maintainable software, when interacting with a new technology. Expert users are therefore interested in specific applications that can fit their interests and foster their work. They indeed need applications specific for users with an expertise in software development to increase their productivity, improve the quality of their application, and make the source code of the application easier to maintain (**?**). In other words, the expert users expect applications that facilitate their tasks and make the software development more accessible and satisfying. (**??**).

Among the tools dedicated to expert users, there are those specifically created to analyze the code and software requirements specification for correctness, performance, complexity and modularity (**???**). The Ethereum blockchain is quite recent when compared to other technologies, such as the web. This implies that there are not so many tools to help the developers to build applications within the blockchain technology. Furthermore, bugs in deployed smart contracts can cause severe economic loss (**??**). For instance, in June 2016, a user exploited a software vulnerability and drained millions of Ether – with a theoretical value in tens millions of dollars (**?**). A wallet identified by the community members as the recipient of the stolen funds contains over 3.5 million Ether tokens. At an exchange rate of about $2,500USD$ per unit, the economic scam was worth about 50 million dollars. In an open letter to the Ethereum community, the user claimed that he did nothing

illegal, he was only "making use of this explicitly coded feature as per the smart contract terms". smart contracts are indeed prone to human misuse, and although they are potentially immune to mistakes in legal terminology and drafting, they are still vulnerable to coding errors (**?**). This vulnerability needs to be addressed by new blockchain-based applications (**?**).

When considering the World Wide Web (WWW) in the early years (1990-2000), there were not so many tools to help software developers to build web applications (**?**). The integrated development environments (IDE), i.e., the software application that provides comprehensive facilities to computer programmers for software development, were very primitive compared to the one used today. Part of my research work has been devoted to design and develop tools that could help developers to write better smart contracts. This research allowed me to develop two applications aimed at expert users, namely PASO 3.2 and Smart-Corpus 3.3. Based on the Smart-corspus, the dissertation also proposes an empirical study of source code clone practice in smart contract 3.4.

The chapter is organized as follows. Section 3.2 presents PASO, which is a web-based tool supporting static code analysis. PASO performs static code analysis of Solidity, the most used programming language to write smart contracts. PASO differs from the existing applications because, just using a web browser, it is able to provide the expert users with software metrics values for smart contracts written in Solidity. Section 3.3 presents Smart-Corpus, which is an organized repository of the Ethereum smart contracts' source codes and metrics (**?**). It is aimed at expert users, especially as academic researchers, who analyze smart contracts' code to improve the blockchain security, to find design defects and propose solutions. Section 3.4 presents an empirical study of code clones in Solidity. The empirical study of code clones in Solidity is based on the Smart-Corpus and it is aimed to improve expert users' practices in developing smart contracts. Finally, the advantages and the disadvantages of code duplication are discussed, and some "best practices" are proposed to expert users to develop secure smart contracts.

## 3.2 PASO

### 3.2.1 Introduction

The main motivation of this work was to overcome the lack of web based tools supporting static code analysis of Solidity programming language. On the one hand, static analysis tools are very important because they can help developers to find bugs, or software defects, faster and cheaper than manual inspections. On the other hand, the main challenge is that Solidity is at a very early stage and is changing very fast. So the tool to analyze Solidity need to be updated very frequently.

The outline of this research work is the following: Section 3.2.2 reviews the related work on blockchain analysis platforms. Section 3.2.3 presents the hypothesis

of the work. Section 3.2.4 presents the general components of the online tool PASO and the approach we embraced to define the metrics. Section 3.2.5 describes the threats to validity for the research. Finally, in Section 4.4.9, we make our final remarks and draw some conclusion.

### 3.2.2 Related Work

At the date hereof, a web-based and updatable tools to measure source code metrics of Solidity smart contract do not exist yet. For the most popular programming languages, such as C++, C#, Java, Python, PHP and JavaScript, there are instead a lot of web-based tools to perform static code analysis (**?????**). JSHint [1], for example, is a Static Code Analysis Tool for JavaScript programming langauge that detects errors and potential problems in JavaScript code.

For what regards Solidity there are only standalone applications. SolHint [2] is a command-line tool to analyse the Solidity code for potential errors. It also provides both security and style guide validations. Pharo Solidity Parser uses SmaCC (Smalltalk Compiler-Compiler) and relies on Solidity grammar specification to build a parser that can be used to measure metrics for smart contract written in Solidity (**?**).

Zhang et al. (**?**) proposed metrics for measuring the Web Ontology Language (OWL). Although the OWL language is different from Solidity, the underlying concepts are similar. The paper (**?**) is also inspired by the concept of software metrics. The proposed metrics were analytically evaluated against Weyuker's criteria (**?**). It also performed empirical analysis on public domain ontologies to show the characteristics and usefulness of the metrics.

### 3.2.3 Motivation

The Solidity language grammar definition changes very often and, consequently, the tools to measure the software metrics needs to be updated very frequently. Every time a new version of the Solidity program language is released, the existing standalone applications need to be updated accordingly from both the authors' and the end users' perspectives. This extra work, from the end-user point of view, could be avoided by using a web-based tool, which requires to be updated just by refreshing the web page. Of course also the engine behind the computation needs to be updated according to the new Solidity release. The PASO tool, presented in this section, accomplishes to all these requirements. A practical option, which is implemented in PASO, is the idea to have all components on the client side with no need to have a server. This solution has the advantage that there is no need to have any server to maintain and manage. What is needed to run the program

[1] https://jshint.com/about/
[2] https://protofire.github.io/solhint/

is just a web browser, which is installed in every operating system. In addition, PASO has offline functionality, i.e. it can work completely offline once all the PASO components, coded in CSS, HTML and JavaScript, have been downloaded.

### 3.2.4 PASO Components

PASO is available and can be tested at this link. [3]

The main components needed to build and run PASO are:

- Solidity Grammar,
- PASO Parser,
- PASO Metrics,
- PASO GUI (Graphical User Interface).

The following sections give a general definition for each main component for a better understanding of the work made to realize the tool PASO.

#### Solidity Grammar

A programming language is a set of commands, strings of characters readable by programmers but easy to translate to machine code. It has grammar and semantic rules. The grammar is a set of rules that define how the commands have to be arranged to make sense and to be correctly translated to the machine code. Semantics is a set of meanings assigned to every command of the language and it is used to correctly translate the program to machine code (**?**).

Figure 18 shows a piece of Solidity grammar according to the ANTLR rules. Each ANTLR rule consists of a name, followed by a colon, followed by its definition, and terminated by a semicolon. The "sourceUnit" symbol is the entry node of the grammar. Nonterminal nodes in ANTLR have to be lowercase. Terminal nodes have capitalized names, like EOF. EOF is a special terminal node, defined by ANTLR, meaning the end of the input. In particular, it stands for the end of the file, even though the input may also come from a string or a network connection rather than just from a single file.

The symbol "|" represents the alternation operator, the symbol "*" is the repetition operator, and parentheses are used for grouping, in the same way we have been using for a natural language grammar reading. Optional parts can be marked with the symbol "?". In ANTLR, Terminal nodes can be defined using regular expressions, but fixed strings are not permitted. For example, here are some Terminal patterns used in the Solidity grammar written in ANTLR syntax and with ANTLR naming convention: COMMENT → "/*" ".*?" "*/".

The Solidity grammar definition is available at this web site. [4]

[3] https://aphd.github.io/paso/

[4] https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

```
1  grammar Solidity;
2
3  sourceUnit
4    : (pragmaDirective | importDirective | contractDefinition)*
       EOF ;
5
6    ...
7
8  functionDefinition
9    : natSpec? 'function' identifier? parameterList modifierList
       returnParameters? ( ';' | block ) ;
10
11 returnParameters
12   : 'returns' parameterList ;
13
14 modifierList
15   : ( modifierInvocation | stateMutability | ExternalKeyword
16     | PublicKeyword | InternalKeyword | PrivateKeyword )* ;
```

Listing 3.1: This code shows how the Solidity grammar looks like as an ANTLR source file

### PASO Parser

The PASO Parser is generated from a Parser Generator. Figure 3.1 shows the input and the output of a Parser Generator. A parser generator is an application which generates a parser: it takes the Solidity grammar as input and automatically generates a source code named Parser. The parser is a function that takes the sequence of characters of a smart contract as input, attempts to match the sequence with the grammar and produces a parse tree as output. Figure 3.2 shows the input and the output of the PASO Parser.

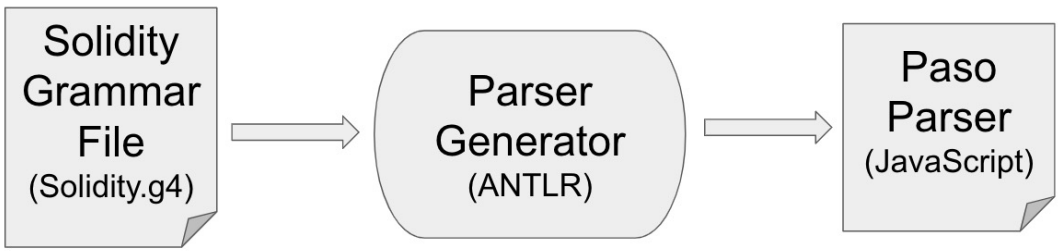



Figure 3.1: The Parser Generator takes a file containing the Solidity grammar rules. It produces a PASO Parser, i.e. a parser in JavaScript computer language that can be run in a client browser.

A parse tree or parsing tree is an ordered, rooted tree that represents the syntactic structure of the source code according to the grammar. The root of the parse tree is

the starting Nonterminal node of the grammar. In a parse tree, a Nonterminal node is a node of the parse tree which is either a root or a branch of the tree, whereas a Terminal node is a node of the parse tree which is a leaf.

There are different parser generator applications for various programming languages. To the aims of this work, it is necessary to use a Parser Generator that can generate a Parser in a client-side scripting language, like JavaScript. Among the different Parser Generators, we chose ANTLR4 (ANother Tool for Language Recognition), precisely because it can produce a Parser in JavaScript programming language that can run on the client part together with the GUI part.

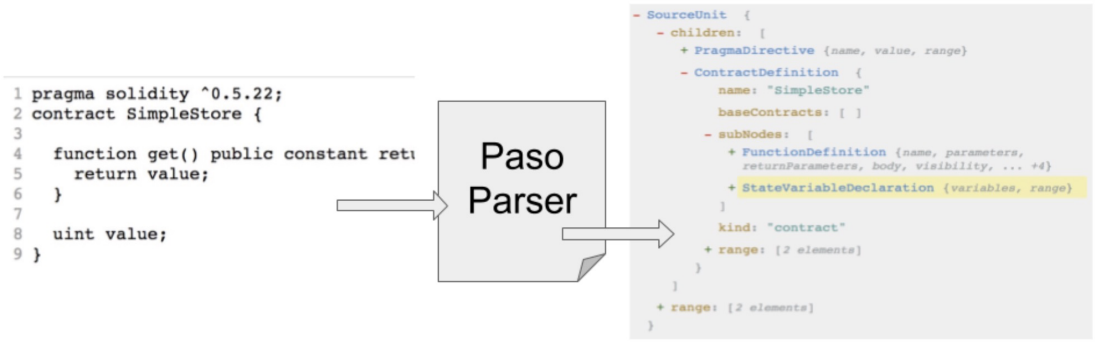



Figure 3.2: Example of input and output of the PASO Parser.

### PASO Metrics

In software engineering software metrics has been defined and used to measure software quality and, more in general, to qualify software under the principle "You Can't Manage What You Don't Measure". Code metrics can be used to detect any characteristic in the source code that possibly indicates a major problem of the code. They therefore act as a useful alert to detect a problem and improve the overall design of the code. An example of problem to detect is duplicated code, i.e. identical or very similar code which exists in one or more parts of the program (**??**). Code metrics can also be used to identify functions or a smart contracts formed by many lines of code (LOC), or to measure the Cyclomatic complexity, i.e. the existence of too many branches or loops. A high value of Cyclomatic complexity metric or/and the LOC metric may indeed indicate that a function needs to be broken into smaller functions, or that it can be simplified (**??**).

Many object-oriented metrics have been proposed over the last decade (**?**) and most of the metrics used in the tool are derived from "C&K" metrics. C&K metrics were proposed by Chidamber and Kemerer in 1991 for object-oriented software (**?**) and details of them can be found in (**?**). The metrics discussed in this work include C&K metrics and add further metrics. The overall set of metrics displayed in the PASO tool are therefore divided into two categories: 1) Object oriented metrics

used to measure properties of object oriented programming languages, such as java, smalltalk, C++ (**?**) 2) Solidity metrics, which are specific for Solidity programming language. PASO displays some of the metrics taken from the two categories 3.3

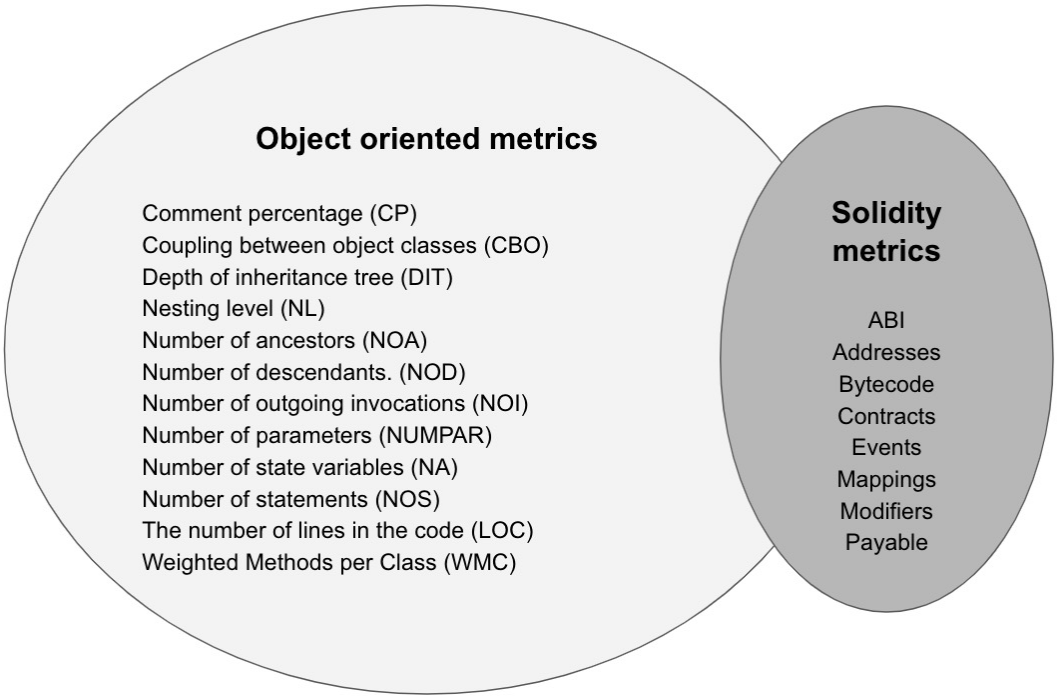



Figure 3.3: The two ovals respectively represent the set of Object oriented metrics (on the left) and the set of metrics that are specific to Solidity Language (on the right).

### Object Oriented Metrics

In the previous literature, static code analysis tools generate metrics for OOP (object oriented programming) languages. Some of the most used metrics are: 1) the number of code lines, 2) coupling, i.e. the number of connections a file or a class has to other files or classes, 3) the number of arguments for a function, etc. Table 3.1 lists the most used metrics in the scientific literature which are implemented in PASO tool.

### Solidity Metrics

Solidity programming language has some peculiarities that makes it unique when compared to other programming languages (**?**).

Table 3.2 lists the metrics that it is possible to obtain by parsing the smart contract code with the PASO tool. These metrics are defined only for Solidity programming languages.

Table 3.1: Some object oriented metrics

| **Metric name** | **Description** |
|---|---|
| CBO | Coupling between object classes. The coupling between object classes metric in the OO paradigm measures the number of classes that the actual class is connected to (by using the class as an attribute type, method parameter or return value, etc.). |
| CP | Comment percentage. |
| DIT | Depth of inheritance tree. The depth of inheritance metric measures how deep a class, library or interface is in the inheritance tree. |
| LOC | It indicates the number of lines in the code. A high number might indicate that a contract is trying to do too much work and should be split up. It might also indicate that the contract or method is hard to maintain. |
| Methods | Number of Methods. Contracts with too many methods may be trying to do too much, or in any case may be more difficult to maintain. |
| NA | Number of state variables. |
| NL | The nesting level metric denotes the sum of the deepest nesting level of the control structures within the functions of a class, library or interface. |
| NOS | Number of statements. The number of statements metric counts how many statements there are in a class, library or interface. |
| NOA | Number of ancestors. The number of ancestors metric counts all the different direct or transitive ancestors of a class, library or interface. |
| NOD | Number of descendants. The number of descendants metric measures how many different direct or transitive descendants a class, library or interface has. |
| NOI | Number of outgoing invocations (i.e. fan-out). The number of outgoing invocations metric measures how many different functions are called from a function in a class or library. |
| NUMPAR | Number of parameters. It counts how many parameters a function has. |
| WMC | Weighted Methods per Class (WMC) is an object-oriented metric to measure complexity in a class. |

Table 3.2: Some Solidity Metrics

| **Metric name** | **Description** |
|---|---|
| Payable | The number of Payable Functions. |
| Mappings | The number of Mapping types. Mappings, in Solidity programming language, can be seen as hash tables. |
| Modifiers | The number of Function Modifiers. |
| Addresses | The number of addresses. |
| Events | The number of Events. |
| Contracts | The number of Contracts. |
| ABI | The size of the ABI (Application Binary Interface). |
| Bytecode | The size of the Bytecode. |

#### PASO GUI

The PASO GUI is the PASO component that allows users to interact with the PASO Parser. The GUI (Fig. 3.11) is divided into two sections: 1) a textarea where the user can write or paste the smart contract or several smart contracts, 2) a section for results where the user can see the values of the metrics displayed. Figure 3.11 presents the two sections: the textarea is shown on the left (Fig. 3.4), while on the right some metrics value are shown (Fig. 3.5), corresponding to the smart contract written in the textarea.

### 3.2.5 Limitation

Studying the quality of a software might be a difficult, often subjective process. Having some metrics value that measure an application's source code provides a useful starting point to improve the existing code. Static code analysis tools such as PASO can spot many different kinds of mistakes, but cannot detect if the Solidity smart contract produces the correct system behavior. A solidity software developer should always combine tools like PASO with unit and functional tests as well as with code reviews. Indeed, PASO can check the correctness of the grammar written by the programmer and compute the associated metrics, but cannot understand whether the meanings assigned by the programmer are coherent, i.e. actually correspond to


Figure 3.4: PASO GUI Textarea.

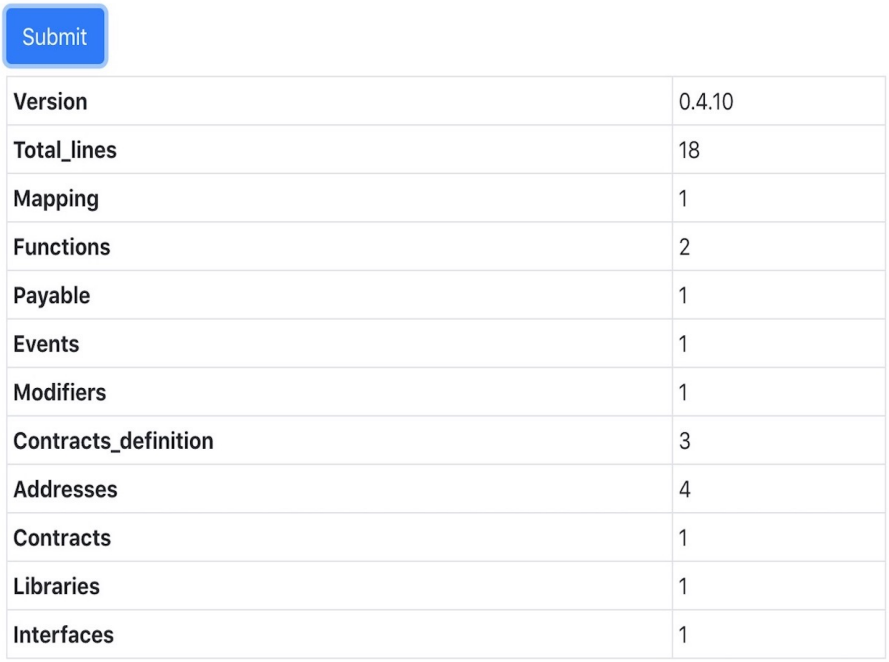



Figure 3.5: PASO GUI Metrics.

Figure 3.6: PASO GUI. Figure 3.4 shows the textarea where the user can write or paste the smart contract or several smart contracts. Figure 3.5 shows some metrics value corresponding to the smart contract written in the textarea.

what the programmer wanted to achieve.

The PASO tool is limited in the interface design. For instance, it displays the metrics value in numerical form. An improvement of the tool could be the visualization of metrics by using a treemap, as for example in the work by Balzer (**?**). PASO is also limited in the number of metrics it computes but is structured to be easily updated, also accordingly to the eventual future evolution of the Solidity EVM. Example of missing metrics in the PASO tool are: 1) the coupling metric that describes the number of connections a file or a contract has to other files or contracts, and 2) The Cyclomatic Complexity of a function. However, thanks to a modular design, it can be used as a basis for a richer implementation that gives more precise information to the developer and a more user-friendly graphical interface to the user.

### 3.2.6 Conclusion and Future Work

The work presents a fully web-based tool able to compute the smart contract metrics. The goal has been achieved by using the ANTLR (ANother Tool for Language Recognition) parser generator. We gave the Solidity grammar as an input to the ANTLR Parser Generator, which has been used to create a JavaScript Parser. Finally we wrote the Solidity Code in the PASO GUI textarea, thus giving it as an input to the JavaScript Parser, which calculated and displayed the metrics values on the screen. Before implementing PASO, there were only standalone applications, such as Pharo Solidity Parser and SolMet, which have allowed to parse and to generate metrics for a smart contract written in Solidity program language. The research assessed the hypothesis that it is possible to build a completely web-based tool, PASO, able to achieve at least the same results of the previous standalone applications. The main advantage of having such web-based tool - when compared to the previous ones - are: 1) users have no need to install a third-party software, like Java or Smalltalk Pharo, 2) PASO is able to cope in a more efficient way with the countless updates of Solidity programming language: by using PASO, there is no need to update the standalone application, it is only needed to update the web page.

The number of metrics discussed in the work are just a few: a complete list of metrics to be implemented in the PASO tool would indeed require more in-depth research that can be developed in future works. The aim of the present research was indeed limited to test the hypothesis that it is possible to build a fully and updatable web based tool to compute the metrics value of a smart contract written in Solidity without installing any tools on users' local computer. At the time of writing (December 2019), PASO is the only fully web-based tool that allows to parse and to generate metrics for a smart contract written in Solidity program language. It represents a starting point for a future richer implementation, able to give more relevant information to the developers and a more user-friendly graphical interface to the user.

## 3.3 Smart-Corpus

### 3.3.1 Introduction

With the advent of blockchain technology as a mainstream technological innovation, many researchers and software developers started investigating the new possibilities for software products relying on such an infrastructure. Second-generation blockchains offer the possibility to code so-called smart contracts in a Turing complete programming language on which all the main operations of traditional software systems can be carried out. The paradigmatic reference is the Ethereum blockchain, which offers the possibility to deploy and execute decentralized applications (dApps) which are mainly coded in Solidity, presently the most adopted programming language (**?**).

Coding smart contracts which run in a blockchain environment has its peculiarities and constraints and differs from coding in traditional out-of-chain contexts. One of the major differences is the immutability of deployed code: if bugs or bad smells are introduced into a smart contract, these cannot be fixed afterwards with patches. Another contract must be deployed in substitution of the former and users must be well advised not to use the wrong code. Another main issue is the interaction with the blockchain by means of transactions where information exchange can occur only between blockchain internal components. Furthermore, memory occupation on blockchain typically has a cost that developers want to reduce, and chaining all the blocks poses limitations to the reasonable space available for each smart contract imposing practical constraints to source code size.

This new programming paradigm poses major challenges also for expert developers, and famous failures are commonly found in blockchain software (**??**). The novelty of the paradigm largely contributes to these faults, since developers do not have historical records or examples to learn from previous code, as it happens in traditional software coding, where software reuse and coding by imitation are reference practices to help in coding better-quality software. Another issue is the lack of reference measures, such as quality, complexity or coupling metrics, which are extensively used in out-of-chain software production to keep software projects under control (**?**).

The situation is slowly changing for historical records (even if history is quite recent) of software code, since the Ethereum blockchain can now count on up to 1.5 million deployed smart contracts, which have been used and run in the last few years. Access to the source code of this body of smart contracts is still a challenge since transparency and open access granted by public blockchains regards only data registered in the blocks, where only contracts' byte codes are available.

To access the smart contracts' source codes, the developers must resort to other means or to code repositories, such as the classical GitHub or similar resources. Fortunately, in the last few years, EtherScan (`https://etherscan.io/`) and other web sites have started providing smart-contract checking as a service, so that Ethereum

developers can submit their source code to be analyzed and the source code is made available afterwards by the website. However, there is an odd side of the medal for many reasons: access to this body of knowledge is far from easy and far from fast; it is not structured and organized; and the smart-contract metrics are not available and must be computed separately. All these tasks can and need to be automatized to save developers time and work as well as computational resources. Indeed, in the last few years, a number of research papers have been published reporting findings based on smart contracts' source codes, mined from GitHub or some Ethereum block explorer such as EtherScan (**????**). However, when conducting this kind of empirical research on smart contracts with data from Ethereum blockchain, the abovementioned tasks need to be carried out by the developers themselves. The first task is downloading the smart contracts' source codes to be analyzed. One way to download smart contracts' source code data is to inspect open-source software (OSS) project repositories such as GitHub (**?**).

Another way to perform this task is to use an Ethereum block explorer. These web services allow users to find the desired information by directly accessing the Ethereum blocks, by using a unique identifier or by sequentially searching several blocks (**?**). Some of these Ethereum block explorers provide RESTful Web services, which allow the users to obtain a JSON format payload containing various data. These data may be related to a current or past state of the Ethereum blockchain: an example may be the list of transaction addresses included in a given block of the Ethereum blockchain. This activity might be tedious and time-consuming (**?**) when conducted by a single user/developer/researcher. Furthermore, the obtained smart contracts' data set can consist of duplicated smart contracts, i.e., smart contracts having different addresses but the same code.

In this work, we tackle these problems and propose an organized, easy to use, large and available software repository for Ethereum smart-contract source codes and metrics where users, researchers, blockchain startups and developers can take advantage of the body of knowledge collected during the last few years. This section thus proposes Smart Corpus, a repository that provides users with an interface which allows for searching for and downloading smart contracts' source codes. The user interface is available at the following online address: `https://aphd.github.io/smac-corpus/`. The main challenge of the implementation lies in the fact that the Ethereum blockchain stores a massive amount of heterogeneous data, smart contracts included, which grow enormously in time. For this reason, Smart Corpus was designed to be scalable by adopting the latest cutting-edge technology, such as document-oriented database, graph query language and serverless computing platform (**?**).

## Related Work

### Previous Literature on Software Corpus Analysis

Gabel and Su (**?**) built and studied a corpus of open-source software written in three of the most widely used languages: C, C++, and Java. The corpus contains six thousand software projects corresponding to 430 million lines of source code. The authors measured the degree to which each project of the corpus can be "assembled" solely from portions of the corpus, thus providing a precise measure of "uniqueness". Their primary contribution is to provide a quantitative answer to the following question: *how unique is software?* Our work also aims to answer this question because many smart contracts written in the Solidity language have code that is a replication of other smart contracts, although presenting different addresses, as we will see in Section 3.3.2. Our goal is therefore to answer the question: *how unique are smart contracts written in Solidity?* in order to provide a corpus that is composed of smart contracts which can be distinguished from each other.

Tempero and coauthors (**?**) presented the "Qualitas Corpus", a curated collection of open-source Java systems. The corpus reduced the time needed to find, collect and organize the necessary source code sets to the time needed to download the corpus. The metadata provided with the corpus explicitly indicate the metrics calculated to identify the main features of the source code: the number of code lines, the number of classes, etc. Our work also aims to present a curated collection of smart contracts equipped with a set of metadata with the aim of allowing experts in the blockchain field to perform static analysis.

### Static Analysis on Smart-Contract Code

There is a number of scientific publications with the objective of analysing smart contracts' source codes and testifying to the scientific community's interest in advancing the knowledge on the characteristics of smart contracts' code structure.

Hegedus (**?**) developed a metric calculator for Solidity code, inspired by the work by Tonelli and collaborators (**?**). The metric calculator uses a parser to generate an abstract syntax tree (AST), on which it computes various software metrics, such as the number of code lines for each smart contract, the cyclomatic complexity, the number of functions and the number of parameters for each function. This command-line tool is written in Java and is available on GitHub without license indication since February 2018 (`https://github.com/chicxurug/SolMet-Solidity-parser`). By using this tool, he calculated and published software metrics results for 10,206 Solidity smart-contract source code files written in Solidity languages. Our work also aims to calculate a set of metadata on the smart-contract corpus by using a similar software.

Pinna and colleagues (**?**) performed a static analysis on 10,174 smart contracts, deployed in the Ethereum blockchain. The authors showed that some metrics related to smart contracts, such as the number of transactions and the balances, follow

power-law distributions. Also, they reported that software code metrics in Solidity have (on average) lower values but higher variance than metrics values in other programming languages for standard softwares. Our work is inspired by their research as Smart Corpus is characterized by (some of) the metrics they defined, as we will explain in Section 4.3.4.

Pierro and Tonelli (**?**) pointed out that even the most experienced users, as software developers of smart contracts are, need to be helped to analyse smart contracts and to write a more reliable and secure code. For this reason, an open-source platform (`https://aphd.github.io/paso/`), called PASO, was proposed as an aid for experts in smart contracts' static analyses. Their work focused on Ethereum blockchain and smart contracts written in Solidity. The platform PASO facilitates debugging of smart contracts by providing software metrics commonly used to comply with coding guidelines.

### Related Projects

Other projects similar to Smart Corpus have been previously developed to access online smart contracts' codes deployed in the Ethereum blockchain platform. The projects present specific features and limitations, which are summarized in Table 3.3.

Table 3.3: Project list, main features and limitations.

| **Project's Name** | **Home Page** | **REST API URL** | **Limitations** |
|---|---|---|---|
| GitHub | https://github.com/ | https://developer.git... | Some repositories have restricted access. |
| Ethplorer | https://ethplorer.io/ | https://api.ethplorer... | Requests are limited to 3000/week. |
| EtherScan | https://etherscan.io/ | https://ethers... | Smart contracts' addresses are not immediately available. |
| EtherChain | https://www.etherch... | https://www.ethe... | Smart contracts' source codes are not available. |
| BlockScout | https://blocksco... | https://blockscout... | Smart contracts' source codes are not available. |

### GitHub

GitHub is the largest collaborative source code-hosting site built on top of the Git version control system (**?**). The availability of a comprehensive Application Programming Interface (API) has made GitHub a target for many software engineering and online collaboration research efforts (**?**). GitHub offers just open-source software to the community. In GitHub, there are many works regarding projects written in different programming languages, such as Java, Python and Solidity, which is by far the most commonly used language to write smart contracts.

The repository proposed in this work overcomes the following GitHub limitations:

- The smart-contract source codes collected in GitHub typically do not have a direct reference to smart contracts deployed on the blockchain through an Ethereum address; therefore, it is hard to find out whether it has been really tested or used on the blockchain.
- GitHub does not implement a search engine to filter smart contracts based on particular software metrics, such as the number of modifiers or payables. This is due to the fact that some metrics are specific to the type of language employed to write smart contracts, i.e., Solidity.
- In GitHub, there is no information on smart contracts' use in a real blockchain scenario, on the number of transactions invoking smart contracts or on the number of tokens associated with each smart contract.
- GitHub does not provide smart-contract ABIs or Opcodes.

It is highly probable that the users, especially if they are developers or researchers, want to access smart contracts' source codes, choosing the features implemented in Smart Corpus, on the basis of its specific software metrics and its real usage on the blockchain.

### Ethereum Block Explorers

Ethereum block explorers are platforms that allow the users to explore and search the Ethereum blockchain for transactions, addresses, tokens and other activities taking place on the Ethereum blockchain (**?**). Unlike GitHub, the Ethereum block explorers allow access to only Ethereum data used in the Ethereum blockchain and thus smart contracts' real use cases. To date, in the market, there are different Ethereum block explorers:

- **Ethplorer** (`https://ethplorer.io/`) provides an API to access many Ethereum data, such as the balances for a specified token and the description of a specific address, but it does not allow access to the smart contracts' code. The full documentation of the Ethpoler API is available

at the following address (`https://github.com/EverexIO/Ethplorer/wiki/Ethplorer-API`). The requests to API are limited to 5 per second, 50/min, 200/h, 2000/24 h and 3000/week.

- **EtherChain** (`https://etherchain.org/`) is an explorer for the Ethereum blockchain. Unlike Ethplorer, it claims to provide smart contract code, even though it actually displays the contract byte code and the constructor arguments for a specific smart contract's address. EtherChain provides the API just to access the Oracle gas price predictions (`https://www.etherchain.org/api/gasPriceOracle`), but not the Ethreum data. If the users want to gather Ethereum data from EtherChain, they need to parse the HTML code.

- **BlockScout** (`https://blockscout.com/poa/xdai/`) provides an API to access the Ethereum data. It claims to have an API to access only the source code of a few verified smart contracts. Anyway, the addresses list of the verified smart contracts is not available in BlockScout.

- **EtherScan** allows for exploration and searching of the Ethereum blockchain for smart contracts. However, when downloading the smart contracts' source code, the block explorer presents some limitation. First, smart contracts' data and number are huge (on the Giga scale, based on our estimation), but there is a limited API rate of 100 submissions per day per user to retrieve just a smart contract, making the complete download of data an impossible endeavour (`https://etherscan.io/apis#contracts`). Second, the EtherScan's API does not provide facilities to obtain a list of the smart contracts' addresses, as the existing API calls mainly allow navigation from one block to another. Third, a researcher cannot directly and easily explore the smart contract's source code but, rather, has to first inspect any block in Ethereum and then look for all the transactions that involve an address associated with the smart contract.

### 3.3.2 Research Methodology

Smart Corpus has been designed to provide the users with a reasoned repository, i.e., a repository which is not just a webspace where to collect them but also mainly a service to help the researchers filter and analyze the smart contracts' source codes. To this aim, Smart Corpus has been planned to perform four main automatic operations on smart contracts' source codes (data):

1. data retrieving,

2. data cleaning,

3. data modelling and

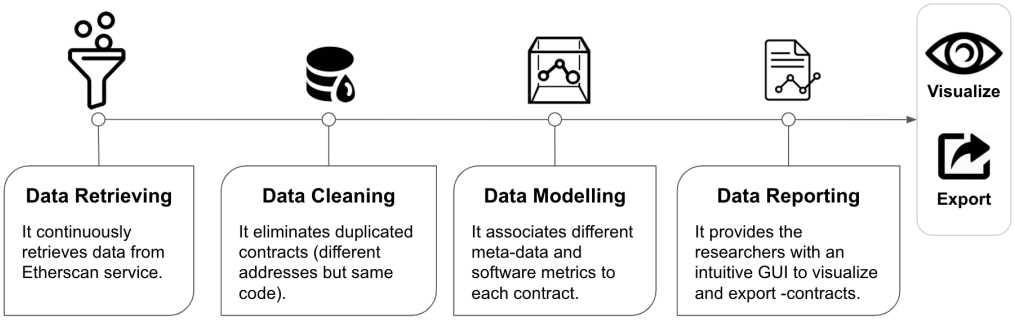



Figure 3.7: Smart-Corpus's pipeline model.

4. data querying.

Figure 3.7 shows the Smart Corpus's pipeline of operations.

### Retrieving Data

We collected smart contracts' source codes, smart contracts' application binary interfaces (ABIs) and smart contracts' byte codes through the EtherScan website, which makes available the source code of a subset of verified smart contracts deployed on the Etehreum blockchain, though in a labourious way. We instead made this task easier and automatic via a retrieving data script available at the following address (`https://github.com/aphd/solidity-metrics/tree/master/examples`). During this phase, the blockchain blocks are automatically inspected. Each block is formed by a list of transactions between two different blockchain addresses, which can refer to a wallet or to a smart contract. The script looks for addresses that refer to a smart contract, and when the source code is available, it downloads the smart contract's source code, the ABI and the byte code. The data coming from the source code are not immediately available as they are embedded in the HTML code of the webpage provided by EtherScan. Therefore, the script removes the HTML tags and stores the code cleaned up.

Figure 3.11a shows how Smart Corpus finds the smart contracts' list in a given block. Figure 3.11b shows the HTML page where the smart contract code is available. The HTML page containing the smart-contract code and the HTML tags is downloaded. Figure 3.11c shows the HTML code that will be processed to remove the HTML tags and to save just the Solidity source code of the smart-contract.

The smart contracts' codes are stored in the filesystem of the Smart Corpus server. Due to the quota limits on queries per second (the EtherScan website allows a few connections per second), Smart Corpus contains only a portion of all available smart contracts. However, the retrieving data phase is continuously collecting data, starting from 10 December 2019. To date, thirty thousand smart contracts (source code, ABI and byte code) have been downloaded and made available through Smart Corpus.

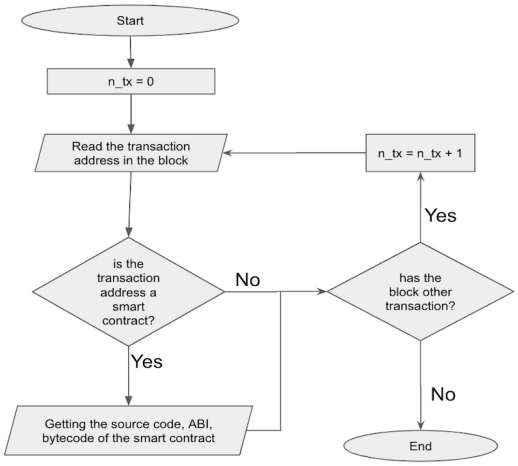



Figure 3.8


Figure 3.9

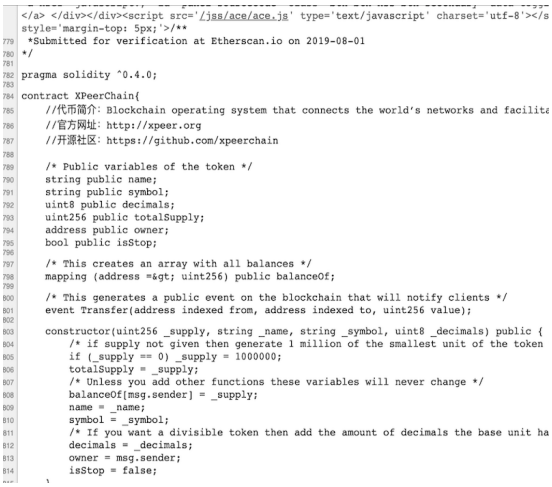



Figure 3.10

Figure 3.11: Data retrieving pipeline: Figure 3.11a–c shows three different phases to retrieve the smart contracts. (**a**) Transactions list in a block, (**b**) smart contract's webpage code and (**c**) smart contract's source code.

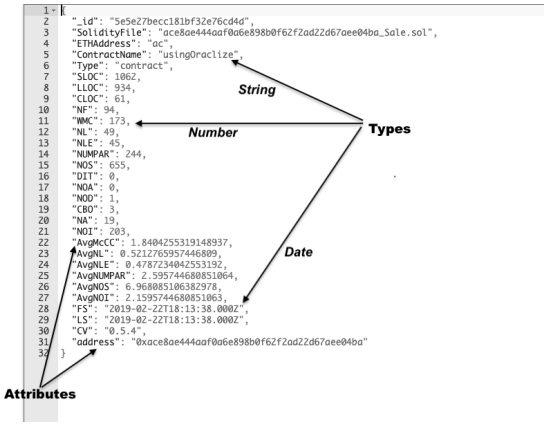



Figure 3.12: Smart Corpus's database schema.

### Cleaning Data

Each smart contract in the Ethereum blockchain is distinguished from any other smart contract as it is identified by a unique address, i.e., a hash of 160 bits, and its byte code is stored on the blockchain (**?**). Indeed, each time a smart contract is deployed in the network, either in the main or in the test network, a unique address is associated with the smart contract even in the case where the source code of two or more smart contracts is the same. However, this is a problem for the analysis of the software metrics because the smart contracts are distinguished only on the basis of their address and not on their content. Therefore, Smart Corpus eliminates double contracts in order to provide a clean smart contracts' corpus on which to perform the analysis. To this aim, duplicate smart contracts have been defined on the basis of their content, i.e., having the same code despite presenting different addresses.

### Modelling Data

Unlike the tools discussed in the related work of Section 4.4.3, Smart Corpus associates different metrics (intrinsic metrics and extrinsic metrics) to the smart contracts, aiming to facilitate the selection of a smart-contract set that meets precise requirements. The metrics associated with the smart contracts are then stored in a document-oriented database. Figure 3.12 shows the database schema of a smart contract.

### Smart Contracts' Intrinsic Metrics

The smart contracts' intrinsic metrics are smart contracts' software metrics which depend on internal properties of the smart contracts' code, such as the number of lines of code, modifiers, payable, etc. Table 3.4 shows the smart contracts' intrinsic software metrics.

Table 3.4: Smart contracts' intrinsic metrics.

| Name | Description |
|---|---|
| Pragma | "Pragma" indicates which version of Solidity compiler is used to prevent issues with future compiler versions. |
| SLOC | "SLOC" indicates the number of lines in a smart contracts' source code. |
| Modifiers | "Modifiers" indicates the number of function modifiers in a smart-contract. |
| Payable | "Payable" indicates the number of payable functions in a smart-contract. |
| Mapping | "Mapping" indicates the number of variables of mapping types in a smart-contract. |
| Address | "Address" indicates the number of variables of address types in a smart-contract. |

### Smart Contracts' Extrinsic Metric

The smart contracts' extrinsic metrics are properties depending on external factors rather than the code itself, such as the number of transactions executed to the smart contracts or the number of tokens associated with the smart contracts. Table 3.5 shows the smart contracts' extrinsic metrics.

### Filtered Data

The smart contracts' source code is stored in a file system and is organized in folders and subfolders to ease the navigation. Figure 3.13 shows the subdirectory structure. The first leaf corresponds to the first two letters of the smart-contract address, and then, each directory contains the file named using the full address of the smart contract and three different extensions, respectively .sol for the Solidity source code, .abi for the ABI and .bytecode for the byte code.

The metadata (both the intrinsic and extrinsic metrics) are stored in a document-oriented database: MongoDB (**?**). The choice to use a document-oriented database instead of a relational database such as Mysql is based on the following:

- Relational databases are prone to deterioration when data sets overcome a size threshold, while a document-oriented database such as MongoDB comes with an inbuilt load balancer, which makes it a better solution in applications with high data load (**?**). We update MongoDB each day to generate the data archive.

Table 3.5: Smart contracts' extrinsic metrics.

| **Name** | **Description** |
|---|---|
| Transactions | "Transactions" represent the total number of transactions generated by the smart contract (sent or received). |
| Balance | "Balance" is the amount of crypto coins associated with a smart-contract address. |
| EtherValue | "EtherValue" is the dollar value associated with a smart-contract address. |
| Token | "Token" is the value for each token associated with a smart-contract address. |
| Last_seen | "Last_seen" is the timestamp of the last time that the smart contract was used (sent or received). |
| First_seen | "First_seen" is the timestamp of the first time that the smart contract was used (sent or received). |

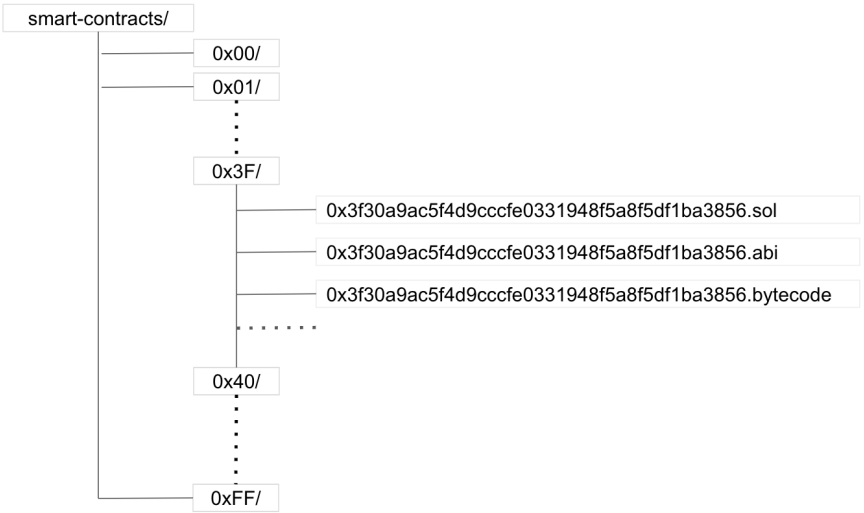
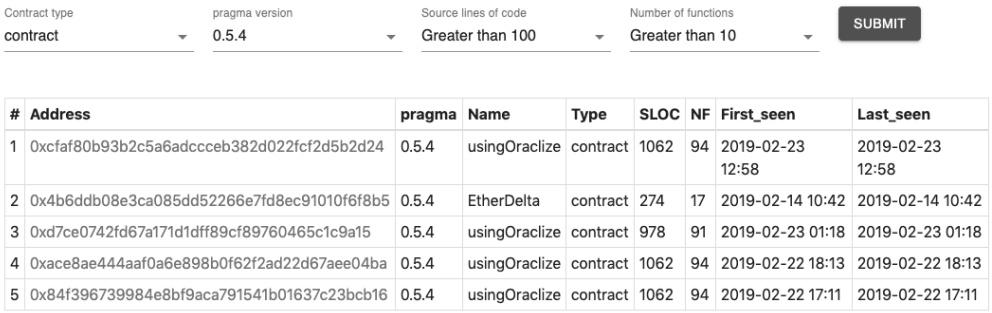



Figure 3.13: Smart contracts' directory structure.




| # | Address | pragma | Name | Type | SLOC | NF | First_seen | Last_seen |
|---|---|---|---|---|---|---|---|---|
| 1 | 0xcfaf80b93b2c5a6adccceb382d022fcf2d5b2d24 | 0.5.4 | usingOraclize | contract | 1062 | 94 | 2019-02-23 12:58 | 2019-02-23 12:58 |
| 2 | 0x4b6ddb08e3ca085dd52266e7fd8ec91010f6f8b5 | 0.5.4 | EtherDelta | contract | 274 | 17 | 2019-02-14 10:42 | 2019-02-14 10:42 |
| 3 | 0xd7ce0742fd67a171d1dff89cf89760465c1c9a15 | 0.5.4 | usingOraclize | contract | 978 | 91 | 2019-02-23 01:18 | 2019-02-23 01:18 |
| 4 | 0xace8ae444aaf0a6e898b0f62f2ad22d67aee04ba | 0.5.4 | usingOraclize | contract | 1062 | 94 | 2019-02-22 18:13 | 2019-02-22 18:13 |
| 5 | 0x84f396739984e8bf9aca791541b01637c23bcb16 | 0.5.4 | usingOraclize | contract | 1062 | 94 | 2019-02-22 17:11 | 2019-02-22 17:11 |

Figure 3.14: Smart Corpus's user interface.

- Unlike relational databases where data is stored in rows and columns, document-oriented databases store data in documents. The documents typically use a structure similar to JSON (JavaScript Object Notation); they indeed provide a natural way to model data that is closely aligned with object-oriented programming. Each document is considered an object in object-oriented programming; similarly, each document is a JSON in document-oriented database. The concept of a schema in document databases is dynamic: every document might contain a different number of fields. This is useful when modeling unstructured and polymorphic data. Also, document databases allow robust queries: any combination of fields in the document can be combined for querying data (**?**).

### User Interface

Smart Corpus' graphical user interface (GUI) allows users to access the smart contracts' repository. There are two ways to access the smart contracts' repository: through the "HTML user interface" and through a "GraphiQL application", both of them via a web browser.

### Smart Corpus HTML User Interface

The Smart Corpus HTML user interface is publicly available since January 2020 (`https://aphd.github.io/smac-corpus/`). Figure 3.14 shows the different components of the GUI.

- At the top, the user can find the form to filter the smart contracts. The form is made of a number of drop-down lists, each one corresponding to a different metric and a submit button to perform the research. The GUI form allows the user to inspect smart contracts based on some metadata, such as the "pragma version", and software metrics, such as the numbers of "modifiers" and/or the numbers of "payable".

- Below the form, the smart contracts filtered by the user are displayed. For readability, only a part of the smart-contract metrics are presented in the table layout format. Each column header in the table indicates the name of a metric associated to smart contracts. While the HTML GUI displays just some metrics, the user can access all the metrics and the smart contracts' source codes by selecting the checkbox displayed on the right of the smart-contract address and by clicking on the red button "download". The user can also access the original repository where the smart contract was retrieved, i.e., the EtherScan service.

### Smart Corpus GraphQL Application

Graph Query Language (GQL) is a full data query language to implement web-based services, centered on high-level abstractions, such as schemas, types, queries and mutations. GQL is a domain-specific language internally developed in Facebook from 2012 onward and publicly announced in 2015, with the release of a draft language specification. The language was conceived with the following goals:

- To reduce possible overload of data transfer relative to Representational State Transfer (REST)-like web service models in terms of both the amount of data unnecessarily transferred and the number of separate queries required to do it.

- To reduce the potential of errors caused by invalid queries on the part of the client. In particular, with the GQL application, the user can execute "type introspection", i.e., the user can examine the type or properties of an object at runtime. For example, thanks to introspection queries, the user can find out both the intrinsic and the extrinsic metrics associated with a specific smart-contract while typing the query.

Figure 3.15 shows an example query and its result.

Smart Corpus GQL application, unlike the Smart Corpus HTML user interface, is still in the development and testing phase. However, the Smart Corpus GQL application source code is publicly available and can be downloaded and deployed on any platform having the software requirements specified in its documentation (`https://github.com/aphd/smac-corpus-api`). Appendix 3.3.2 presents all the queries GraphQL application can perform.

### Appendix

Listing A1 displays a GQL query that returns smart contracts' addresses having more than 20 methods defined in a contract. Listing A2 displays the query results in the JSON format. The query output, in addition to the smart contract's addresses, contains various information (intrinsic metrics) such as the number of events, the


Figure 3.15: Example use of variables to filter a query result with GraphQL.

number of functions, the number of modifiers and the number of payables, as specified by the query A1,

Listing A1: A Graph Query Language (GQL) query for displaying intrinsic metrics.

```
{
    metrics(query:{functions_gt: 20}) {
      adress
      events
      functions
      modifiers
      payable
    }
  }
```

Listing A2: A GQL result displaying intrinsic metrics.

```
{
    "data": {
      "metrics": [
        {
          "contractAddress": "0
              xb7f4c286851cbf0cbf2fe8ebf40412b196c0e8ad",
          "events": 7,
          "functions": 27,
          "modifiers": 1,
          "payable": 1
        },
        {
          "contractAddress": "0
              x755cebe8cc53c7cb1e1bb641026a17d37d4aea91",
          "events": 4,
          "functions": 31,
          "modifiers": 1,
```

```
16                "payable": 4
17              },
18              {
19                "contractAddress": "0
                      xb92aa4a864daf0d6a509e73a9364feba44384965",
20                "events": 3,
21                "functions": 24,
22                "modifiers": 1,
23                "payable": 1
24              },
25
26              ...
27
28          }
29      }
```

Listing A3 displays a GQL query that returns some extrinsic metrics of a specific smart contract's address. Listing A4 displays the query results in the JSON format. The query output, in addition to the smart contract's address, contains information such as the total number of transactions generated by the smart contract and the amount of crypto coins associated with the smart contract's address specified by the query A3,

Listing A3: A GQL query for displaying exstrinsic metrics.

```
1       {
2           metrics(query:{address_eq: "0
                x536c7efeebff067a69393133b1c87a163a6b0598"})
3           {
4             adress
5             transactions
6             balance
7           }
8         }
```

Listing A4: A GQL result displaying exstrinsic metrics.

```
1       {
2           "data": {
3             "metrics": [
4               {
5                 "contractAddress": "0
                      x536c7efeebff067a69393133b1c87a163a6b0598",
6                 "transactions": 639 ,
7                 "balance": 0 Ether
8               }
9             ]
```

```
10              }
11          }
```

**Use Case**

A use case for Smart Corpus might concern a researcher interested in the static analysis of smart contracts. For example, the researcher might be interested in performing an analysis of smart contracts written with a particular version of the Solidity language, 6.0, and having at least a payable function in the smart contract. The research of smart contracts that meets these requirements would be very expensive in terms of time, work and computational resources using a service like EtherScan. Instead, thanks to Smart Corpus, the user needs to perform only a few steps, as described below:

- connect to the service through the link: `https://aphd.github.io/smac-corpus/`,
- select the option "version 6.0" from the drop-down menu entitled "pragma version",
- select the option "greater than zero" from the drop-down menu entitled "number of payables" and
- submit the form by clicking on the button "submit".

After few seconds, depending on the number of smart contracts that meet the requirements specified by the user, the smart contracts' addresses and the metrics values will be displayed in a table layout format ready to be downloaded.

### 3.3.3 Results

Smart Corpus has been in use for 10 months, since December 2019, and 100 K smart contracts have been downloaded via the user interface. Until the tool was developed (October 2020), Smart Corpus was a curated corpus of 30 K smart-contract source codes, ABI and byte codes with related metadata and software metrics. As time passes, Smart Corpus is continuously increasing at a rate of 100 smart contracts per day. Figure 3.16 shows the number of smart contracts' source codes, ABI and byte codes retrieved per day since Smart Corpus was deployed for the first time. For each smart contract, Smart Corpus computed extrinsic and intrinsic metrics, as described in Sections 3.3.2 and 3.3.2.

Summing up, Smart Corpus has two GUIs to access data: the HTML GUI and the GraphQL interface. The HTML GUI is described in Section 3.3.2, while the GraphQL interface is described in Section 3.3.2. The GraphQL interface gives blockchain researchers the ability to request for exactly what they need. The user can directly access the results via GraphQL interface, as shown in Figure 3.15.

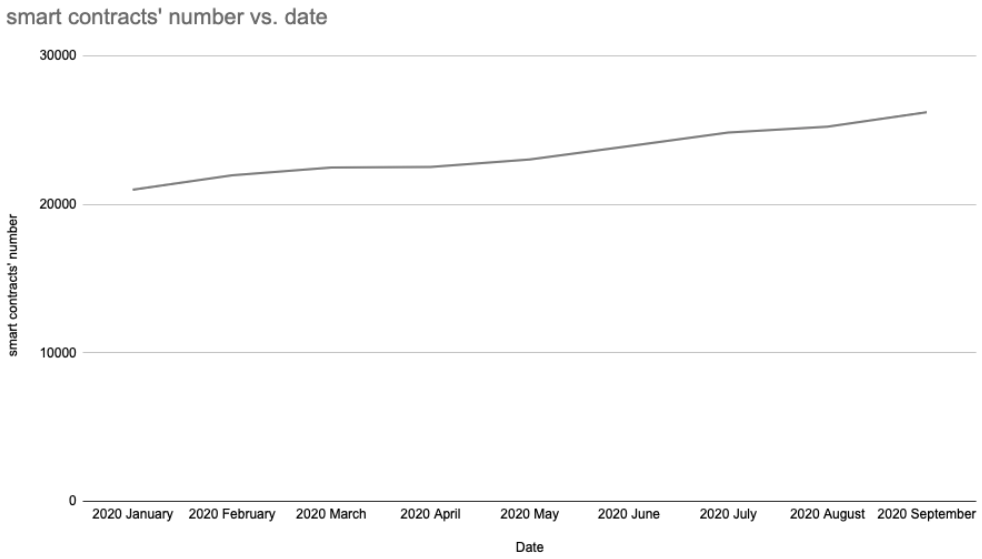



Figure 3.16: Number of smart contracts collected in Smart Corpus.

Unlike the existing repositories (see Section 3.3.1) which make available the source code in a laborious way, Smart Corpus instead made this task easier and faster. Indeed, one of the advantages of using Smart Corpus lies in the fact that it can reduce the costs in performing the smart-contract static analysis. For example, it can be used to easily analyze design and programming patterns for the smart-contract programming language.

Even though the Smart Corpus service has been working for a few months and has not been advertised yet, it has already collected 30K smart contracts, thus providing an interesting and helpful future venue for researchers and software developers interested in the blockchain. Moreover, Smart Corpus allows for analysis of how industry companies use the Solidity programming language to solve concrete problems in different application areas, such as healthcare, insurance, transportation, government, entertainment and energy.

### 3.3.4 Conclusions and Future Works

In this work, we described the Smart Corpus project, an effort to bring smart-contract data (source codes, ABIs and byte codes) to the hands of the research community, providing help to reproducible research and a less time-consuming way to gather data and to perform static analysis. The project has already stored several megabytes of data, which correspond to about thirty thousand smart contracts. This work corresponds to 10 months of data retrieving that are made available to the blockchain scientific community and blockchain developers in a few seconds. The Smart Corpus data set has strong potential to provide an interesting venue for research in many software engineering areas, including but not limited to the best

practices for Solidity software development, distributed collaboration, and code paternity and attribution. The Smart Corpus project is in its initial stage of development, but it can already provide useful insight for researchers on smart contracts' coding and everyday use in the blockchain. The corpus will continue to be expanded in content and in the provision of intrinsic and extrinsic metrics, thus becoming more and more representative of the Solidity code actually used in the blockchain community.

## 3.4 Code Clones in Solidity

### 3.4.1 Introduction

A bad programming habit could be the "code cloning". We define "code cloning" in smart contracts as the act of duplicating identical or near identical pieces of already written source code. According to previous literature (**?**), some problems related to code cloning are:

- the code cloning's tendency to create inconsistencies in the process of update, which hinder maintenance and contribute to the aging of the software.

- the increasing size of the source code due to code cloning.

In the Ethereum blockchain, the cost of Gas to deploy the smart contract is also related to the size of the source code (**?**). Usually, smart contracts with duplicate code can be refactored with a saving in terms of Gas (**??**), but this means that further work for developers is required (**?**).

A motivation for this work is precisely the fact that code clones make the smart contract source files very hard to consistently modify. For instance, if a smart contract has several functions created by code duplication with a slight modification, the software developer needs to carefully modify all the other functions in the smart contracts when a fault is found in one function.

The research addresses the following research questions:

- Q1: What is the percentage of duplicated code on smart contracts deployed in the Ethereum blockchain? Is it increasing or decreasing over the last 5 years?

- Q2: What might be some causes of smart contracts' source code cloning in the Ethereum blockchain?

The research aims to answer the questions, analyzing two corpora of smart contracts and also discussing some cases of clones refactoring.

### 3.4.2 Background

Manual source code copy and modification is often used by programmers as an easy means for the reuse of some functionality. Nevertheless, such a practice produces duplicated pieces of code or clones whose maintenance might be difficult. Duplicated codes are therefore good candidates for system redesign (**?**).

We can define two types of code clone:

- "Local code clone" indicates that the same piece of source code is present in different parts of the same smart contract.
- "Global code clone" indicates that the same piece of source code is present in different smart contracts deployed in the Ethereum blockchain.

Both types of code clones have been considered as a bad software development practice (**?**). Some of the reasons are:

- they can potentially cause maintainability problems, for example when a cloned code fragment needs to be changed, it might be necessary to align such a change across all clones.
- Code duplication increases the size of the code, extending compile time, expanding the size of the executable and thus increasing the costs in the Ethereum blockchain.
- Code duplication often indicates design problems, such as missing inheritance or procedural abstraction which hampers the addition of functionalities.

Previous research pointed out that source code clones are introduced for reasons such as:

- making a copy of a code fragment is simpler and faster than writing the code from scratch. (**?**),
- writing a code with time pressure leads to plenty of opportunities for code duplication, especially in industrial software development contexts, (**??**).

We proposed other hypotheses in the case of smart contracts, as the Ethereum blockchain has other specificities with respect to other programming ecosystems. Some smart contract developers may copy the code from the code of a smart contract that has already proved to be successful, deeming to be successful in their turn. Another reason may be due to the fact that the Ethereum blockchain does not have an official package manager to deploy smart contracts. A package manager is a programming language tool to create project environments which allow to easily import external dependencies (**?**).

The reasons why code clones appear in source code have been analyzed in other programming languages (**?**) and code clones' detection tools have been proposed

as well (**??**). Methods for clone resolution include refactoring (**?**) and meta-level techniques (**?**).

The aim of our research is to investigate the use of source code clone information as a basis for smart contracts source code refactoring. Indeed, sometimes removing clones could be so difficult, that it would be better to maintain the duplication, but sometimes clones could also be good candidates to redesign the system, as they represent duplicated code whose consistent maintenance might be difficult to achieve (**?**). They also form possible explicit connections among components that share the same piece of code and functionalities. Detection of source code clones in large software systems, such as JDK, FreeBSD, NetBSD, Linux, and many other systems has been investigated in the past by (**?**) while clone elimination or reduction in programming languages, such as Java, has been investigated by Balazinska (**?**).

### 3.4.3 Related Work

M. Kondo et al. (**?**) studied the phenomenon of smart contracts cloning in Ethereum Blockchain. They found that 79.2% of the smart contact studied are clones and that the percentage of clones among newly created smart contracts continues to increase over time. Moreover, they identified 26.3% of all 165,005 code blocks extracted from their corpus as identical to OpenZeppelin code blocks. Most of these code blocks belong to the ERC20 OpenZeppelin category. Our study confirmed their findings, by investigating another corpus of smart contracts source code. Also, their analysis regarded smart contracts deployed on the Ethereum blockchain until February 2018. We extended their research to the months until December 2020.

N. He et al. (**?**) proposed a classification of the code clones among the smart contracts deployed in the Ethereum blockchain. They analyzed a corpus of 10 million smart contracts, deployed from July 2015 to December 2018. Interestingly, they found that a large number of duplicated contracts suffered from the vulnerability issues inherited from the original contracts. Some of their results are confirmed by our research. We extended the analysis to smart contracts deployed in the last two years and, in addition, we also considered the code duplication inside the same smart contract.

M. Araoz et al. (**?**) present OpenZeppelin, one of the most popular packages to develop secure smart contracts. OpenZeppelin contains a collection of code blocks (subcontracts, libraries, and interfaces) that can be used as building blocks to develop blockchain-based applications. For instance, it includes implementations of the ERC20 standard, mathematical libraries (e.g., SafeMath), contract lifecycle management contracts (e.g., Pausable contract), and even cryptography utilities. As of December 28, 2020, the project has 2,218 commits, 243 contributors, and 8.9K stars in its GitHub repository. The development team at OpenZeppelin aims to produce high-quality code to be reused by smart contract developers. The team adheres to the following development principles: in-depth security, simple and modular code, clarity-driven naming conventions, comprehensive unit testing, pre-and-

post-condition sanity checks, code consistency, and regular audits. Ultimately, code blocks from OpenZeppelin can be interpreted as "certified" pieces of code that are developed by a community that strives for security and performance. In particular, these code blocks are meant to be reused without modification. Their work is very relevant for us, because a solution to code duplication can come from the libraries proposed by their repository. In our work, we precisely show how code cloning can be avoided by simply using their libraries.

### 3.4.4 Research Methodology

#### Research questions

The research has been lead by the following questions:

- Q1: What is the percentage of duplicated code on smart contracts deployed in the Ethereum blockchain? Is it increasing or decreasing over the last 5 years?
- Q2: What might be some causes of source code cloning in the Ethereum blockchain?

#### Data Collection

To collect smart contracts source code we used the "Smart Corpus" (**?**). "Smart Corpus" is a repository made of 30K smart contract data (source codes, ABIs and byte codes). Unlike the existing repositories which make available the source code in a laborious way, "Smart Corpus" instead makes this task easier and faster. Indeed, one of the main advantages of using Smart Corpus lies in the fact that it can reduce the costs in performing the smart-contract static analysis. For our analysis we have not considered all the smart contracts in the corpus but only a part. This choice was made for two reasons: 1) to have a homogeneous distribution of smart contracts with respect to the programming language version (the pragma) and with respect to the year in which the smart contracts were installed on the Ethrereum blockchain. 2) to reduce the amount of time needed to compare all smart contracts searching for code clones.

#### Data Cleaning

Before performing the algorithm to identify smart contracts code clones, we cleaned the data collected in the "Smart Corpus" based on some considerations of Ethereum blockchain's specific features. The users have no permission to change the smart contracts deployed in the Ethereum blockchain. Indeed, if the user wants to correct a bug in a smart contract, s/he is forced to redeploy and correct the same smart contract by using a new unique address. As a result, on the Ethereum blockchain there might be two or more almost identical smart contracts with different addresses. The fact that different addresses refer to the same smart contract let us suppose that

many smart contracts might simply be "trials" or smart contracts deployed in the blockchain to test and eventually modified them on the basis of the test results.

Fortunately, the smart corpus used to analyse the source code (**?**) in addition to the smart contract's address, contains the smart contract creator's address. The smart contract creator's address is the address of the smart contract owner, who has deployed the smart contract on the Ethereum blockchain. This piece of information allows us to test the hypothesis that many smart contracts are deployed from the same smart contract creator address with few differences. Indeed, for the purpose of this analysis, we excluded similar smart contracts having the same smart contract creator's address: 28% of smart contracts, 2134 of 7623 were excluded for this reason.

### Data Reporting

We distinguished two types of smart contract clones. The local smart contracts clones defined as source code duplication inside the same smart contract and the global smart contracts clones defined as code duplication among all smart contracts deployed in the Ethereum blockchain. We used a script based on simple string matching, to find code clones on the same smart contract. The script performs the following steps:

- the source code is slightly transformed using string manipulation operations that remove spaces, empty lines and comments;
- Then, all the lines of the source code are compared among them to find code clones.

We chose the source code line as the minimal unit on which to perform the algorithm. As an example, the line of a smart contract source code

```
if( a > b && a > c) { // if else statement
```

is condensed to

```
if(a>b&&a>c){
```

We used a different approach to find code clones among different smart contracts, as for instance many DL-based code clone detection methods (**?**). L. Jiang et al. (**?**) developed an algorithm named Deckard. Deckard algorithm is based on Abstract Syntax Tree (AST) of the source code of a program to find exact or close matches of subtrees of another AST source code. The algorithm code is available at the following address: https://github.com/skyhover/Deckard.

As an example, consider the following two smart contracts fragments:

```
for (uint i=0; i<arrayLength; i++) {
  totalValue += mappedUsers[addressIndices[i]];
}

for (uint j=0; j<customersLen; j++) {
  total += customers[addressIndeces[j]];
}
```

The parse trees for the two code fragments are identical, because the code differs only in the identifier names and literal values.

We used Deckard algorithm for the following reasons:

- It is language-independent and works in any programming language that has a context-free grammar (CFG), such as Solidity, the programming language used to write smart contracts.

- It has already been used to analyze clones in smart contracts, and we could thus compare the results of our study with previous studies that used the same algorithm (**?**).

For the Deckard algorithm configuration we set the variable "min_tokens" equal to 50 and the variable "similarity" equal to 0.79. The variable "similarity" is the threshold for tree similarity. Tree similarity is determined as a function of tree editing distance, which is the minimal sequence of edit operations (either relabel a node, insert a node, or delete a node) required to transform one parse tree into another. Following previous literature recommendations, we set the variable value at 0.79 (**?**).

### 3.4.5 Results and Discussion

Tables 3.6, 3.7 present the results of our analysis. The first table 3.6 presents the average percentage of both smart contracts' local and global code duplication, grouped per year in which the smart contracts were deployed in the Ethereum blockchain. The second table 3.7 presents the average percentage of both smart contracts' local and global duplication, grouped per pragma version of the smart contracts deployed in the Ethereum blockchain. Pragma is a directive that specifies what compiler version to use to compile the source code to obtain the runtime byte-code. The results of the trends of both local and global code clones, presented in the two tables, do not change. This can be explained by the fact that there is a direct correlation between the date and the different versions of the Solidity language.

As to the first research question (Q1), Table 3.6 shows how the percentage of both local and global source code cloned is very significant, though the percentage of smart contracts cloned in the Ethereum blockchain is based on a limited sample. Moreover, the results are compatible with previous studies (**?**), even when analyzing a different time window by including the years 2019-20 which had not previously been studied. Table 3.6 also shows that the percentage of global clones increases over time and pragma versions.

As to the second question (Q2), we need to separately discuss the two cases, global and local code clones.

An increase of "global code cloning" over time can be observed. We propose the following explanation for this phenomenon:

Table 3.6: Average Percentage of Duplication divided per year

| Year | 2016 | 2017 | 2018 | 2019 | 2020 |
|---|---|---|---|---|---|
| Local Clone (in the same smart contract) | 68% | 63% | 61% | 57% | 51% |
| Global Clone (among different smart contracts) | 37% | 49% | 68% | 76% | 82% |
| Number of contracts analysed | 1213 | 1984 | 1567 | 1342 | 983 |

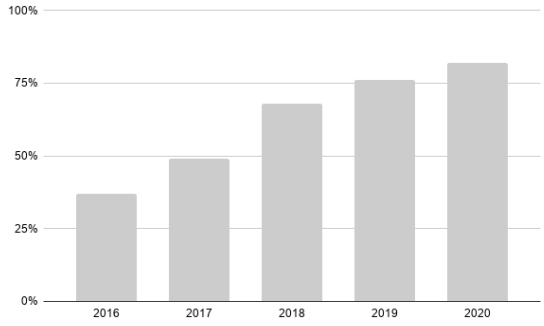



Figure 3.17: Evolution of the percentage of global clones among the smart contracts for every year.

Table 3.7: Average Percentage of Duplication divided per pragma version

| Pragma | 0.3.x | 0.4.x | 0.5.x | 0.6.x | 0.7.x |
|---|---|---|---|---|---|
| Local Clone (in the same smart contract) | 68% | 64% | 63% | 61% | 57% |
| Global Clone (among different smart contracts) | 37% | 49% | 68% | 76% | 82% |
| Number of contracts analysed | 767 | 1912 | 2043 | 898 | 657 |

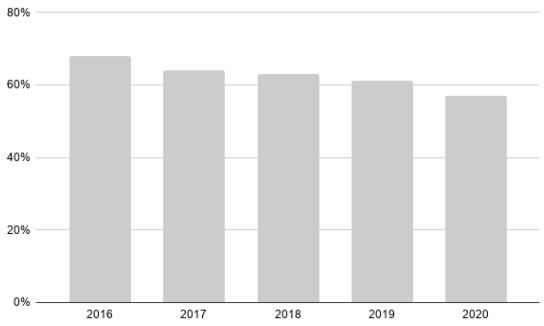



Figure 3.18: Evolution of the percentage of local clones among the smart contracts for every year.

Firstly, from a manual inspection we discovered that some of these cloned contracts refers to highly-active smart contracts. In the context of this research, we considered as highly-active smart contracts the smart contracts having an overall number of transactions (both in input and in output) involving the contract greater than 100 per day. An example of highly-active smart contract is the CryptoKitties smart contract (**?**). It is reasonable to assume that smart contract developers clone highly-active contracts with the hope to achieve the same commercial success (**?**). In detail, the booming success of the popular Ethereum game Decentralized Application (DApp) "Crypto Kitties" (**?**) (a game in which players collect and breed digital cats) in the late 2017 led to the development of a plethora of smart contract clone versions. Some of these Ethereum game DApps are "Crypto Dogs" (**?**) and "Crypto Alpaca" (**?**). However, neither of these smart contracts' clones ever achieved the same popularity of CryptoKitties. Secondly, differently from other more oldest programming languages in use today (C, C++, Java, PHP, Python, ECMAScript), Solidity does not have a package manager.

A package manager is a programming language tool to create project environments which allows software developers to import external dependencies. By using a package manager the developer does not need to reinvent the wheel or to copy and paste the code from other projects to implement new features. Software developers who use other programming languages, such as Java and ECMAScript, can employ this practice. For example, Java software developers use "Apache Maven"(https://maven.apache.org/), Script software developers use "node package manager" (https://www.npmjs.com/). Indeed, a package manager for Solidity would be very useful to add functionalities to smart contracts using code certified by the open source community. Moreover the package manager can perform a security review of the project's dependency tree. Audit reports contain information about security vulnerabilities in the dependencies and can help to fix a vulnerability by providing simple-to-run commands and recommendations for further troubleshooting. Recently, a non-official package manager to develop smart contracts in the Ethereum

blockchain has been proposed. The project is currently under development and the proposal is available at the following address: https://docs.ethpm.com/

An decrease of "local code cloning" over time can be observed, as shown by Figure 3.18. To better understand this decreasing trend, we made a manual inspection of the cloning fragments. From a manual inspection we discovered that some local cloning is easily removable by following the recommendations given by the Solidity programming language documentation available at the following address: https://docs.soliditylang.org/.

Listing A1 shows an example of a local clone (see lines 17 and 22).

Listing A1: Smart contract with local clone (see lines 17 and 22)

```
1
2   pragma solidity >=0.7.0 <0.8.0;
3
4   contract BasicAccessControl1 {
5
6       address payable admin;
7
8       constructor() {
9           admin = msg.sender;
10      }
11
12      function publicFunction1() external {
13          // ...
14      }
15
16      function privateFunction1() external {
17          require(msg.sender == admin, 'Only Admin');
18          // ...
19      }
20
21      function privateFunction2() external {
22          require(msg.sender == admin, 'Only Admin');
23          // ...
24      }
25
26  }
```

According to the official Solidity language documentation, the code duplication can be avoided by using a "function modifier". A function modifier is a Solidity construct which is used as a pattern to change the behavior of some functions, and in many cases, to restrict them. Listing A2 displays the improved version of Listing A1. In detail, it is possible to avoid the code repetition, by including the logic in modifiers (see lines 24-27 of Listing A2) and applying them to a function (see lines 16 and 20 of Listing A2).

Listing A2: Smart contract without local clone.

```
1
```

```
2   pragma solidity >=0.7.0 <0.8.0;
3
4   contract BasicAccessControl1 {
5
6     address public admin;
7
8     constructor() {
9         admin = msg.sender;
10    }
11
12    function publicFunction1() external {
13        // ...
14    }
15
16    function privateFunction1() external onlyAdmin() {
17        // ...
18    }
19
20    function privateFunction2() external onlyAdmin() {
21        // ...
22    }
23
24    modifier onlyAdmin() {
25        require(msg.sender == admin, 'Only Admin');
26        _;
27    }
28 }
```

We propose the following explanation for the decreasing trend of local code repetition over the years.

Firstly, it is reasonable to assume that over time there are more and more tools that help the smart contract developers to write code following the "coding best practices". Coding best practices are a set of informal rules that the software development community employs to improve the quality of softwares (**?**). Indeed, in recent years, several tools have been proposed and published in academic papers. Some of these tools are SmartCheck (**?**), SmartAnvil (**?**) and PASO (**?**). These tools were not available in early versions of Solidity programming language. They share the ability to help the user to detect bad coding practices, such as code repetition and/or possible vulnerabilities.

Another plausible reason to explain this trend lies in the fact that, in general, source code reuse in object-oriented programming languages is made possible through different mechanisms, such as inheritance, shared libraries, object composition, and so on. In the first version of Solidity, some of these mechanisms were not provided to the smart contracts' developers. For example, the "Interface Contract" was introduced only starting from Solidity version "0.4.11". "Interface Contracts", similarly to the interfaces used in object-oriented languages, allow decoupling the definition of a contract from its implementation, providing better extensibility. Indeed, when a Contract Interface is defined, the implementations of a new Contract

can be provided for any existing functions without modifying their declarations.

As a project grows, the need for additional functionality increases. Some of these functionalities (see lines 21–24 of Listing A2), that are cloned among different smart contracts (the global code clones), can be found in various libraries, such as OpenZeppelin. However, Solidity does not have a package manager. Instead the user who aims to reuse the code provided by OpenZeppelin, needs to look for the module in its code repository. This is not a major obstacle for programmers with long experience who know how to search the code in a repository such as OpenZeppelin, but it could be a problem for less experienced programmers, who may search in the web for already written code to implement additional functionalities in their smart contract without a package manager which helps to solve their problems. Some of the already written code may not be updated with the last version of Solidity and present some vulnerabilities.

Listing A3 displays the improved version of the listing A2 by removing the function modifier, which is one of the most copied code among the smart contracts deployed in the Ethereum blockchain, and by importing the "Ownable.sol" module from the OpenZeppelin project (see line 4 of Listing A3). The "Ownable.sol" module provided by the OpenZeppelin project makes the modifier "onlyOwner" available, which can be applied to private functions to restrict their use to the smart contract's owner.

Listing A3: Smart contract which imports the openzeppelin module.

```
1
2  pragma solidity >=0.7.0 <0.8.0;
3
4  import "@openzeppelin/contracts/ownership/Ownable.sol";
5
6  contract BasicAccessControl1 is Ownable{
7
8      address public admin;
9
10     constructor() Ownable{}
11
12     function publicFunction1() external {
13         // ...
14     }
15
16     function privateFunction1() external onlyOwner() {
17         // ...
18     }
19
20     function privateFunction2() external onlyOwner() {
21         // ...
22     }
23 }
```

### 3.4.6 Conclusion

The research showed that different types of code clones can be found in smart contracts. Out of 7500 smart contracts analysed, there are about 80% of smart contracts that contain code which is duplicated from other smart contracts deployed in the Ethereum blockchain. In the same smart contracts corpus, we found that the code clones within the same smart contract are about 40%. Based on previous literature, we know that maintaining these clones is an error-prone task and a potential threat to the system's overall security.

From the analysis done in the Smart Corpus (**?**) we have seen that the two kinds of "code clone" have opposite trends. While the local code repetition is decreasing over the years and it is inversely proportional to the pragma version number, the global code repetition is increasing over the years. Based on the data, we provided some explanations on the possible causes of code duplication for both types of code clones. The proliferation of clones may be caused by the desire to copy successful Ethereum DApps or by the lack of a package manager tool that allows smart contracts developers to easily import external dependencies without the need to copy and paste existing code. A qualitative study involving smart contracts developers (e.g., a survey or a series of interviews) could provide additional insights into the causes of code cloning in the Ethereum blockchain.

# Chapter 4

# Blockchain Models to Design Tools for Non-Expert Users

## 4.1 Introduction

The previous chapter 3, underpinned the idea that blockchain-based tools for expert users are still in an early stage and standards for developing blockchain-based applications have not been defined yet. This fact has hampered and slowed down the adoption of the blockchain (**?**). Many academic studies (**???**) suggest that productivity tools for expert users are crucial to improve the developers' work and to facilitate the adoption of blockchain technology. However, this might not be enough. A new technology needs tools for non-expert users that are easy to use and able to satisfy their needs (**??**), to be adopted on a larger scale.

Although much effort has been made by private companies and academic researchers, non-expert users' interaction with the blockchain technology is still difficult (**??**). Interacting with the blockchain requires technical knowledge (e.g. installing a wallet, being aware of the inability to recover the password of the blockchain account, deciding the minimum fee to pay to execute a transaction within a certain time). Moreover, there is still a lot of misinformation or even missing information on the blockchain among non-expert users, who still prefer traditional applications rather than applications based on the blockchain technology. Therefore, many non-expert users associate the blockchain technology with suspicious investment, because the cryptocurrency is very volatile or because there might be fraudulent smart contracts. To overcome both misinformation and mistrust on the blockchain, more user-centered tools could be provided to overcome the barriers that can limit the access to a wider public, thus facilitating the interaction and understanding of the possibilities that this technology can open.

This chapter is organized as follows. Section 4.2 presents the factors that influence the Ethereum transaction fees (**?**). The section sheds light on how different variables might interact and influence the Oracle Gas Price, i.e. softwares that are

supposed to suggest the best price in Gas units to be paid by the users to execute their transactions (**?**). Section 4.3 evaluates the validity of the prediction the Gas Oracles make on the Gas price to pay to have the transaction recorded in the blockchain (**?**). Section 4.4 presents a new model for the Gas price prediction to meet the user's real needs in terms of fees to pay and waiting time to execute their blockchain transactions. Section 4.5 analyses malicious smart contracts to provide non-expert users with a means to prevent transaction fraud. The malicious smart contracts are intended to steal money especially from non-expert users that have no technical skills to understand that such smart contracts advertise something other than what they actually do.

## 4.2 The Influence Factors on Ethereum Transaction Fees

.

Blockchains are made of *blocks*. A block is a container data structure and it is composed of a header and a list of transactions. Every transaction in Ethereum must pay a transaction fee in a special resource called *Gas* (**??**). Gas is "fuel" for computational instructions executed in the blockchain. The general idea is to make users pay for the computational costs (*e.g.,* energy, CPU) necessary to execute, create, and approve their transactions. Gas is bought by using Ether, a digital currency. A *Miner* approves a batch of transactions by adding them to the blockchain ledger and gets a reward as well as the transaction fees converted to Ether. Since Ether is the second most valued cryptocurrency in the world[1] (the most valued is Bitcoin (**?**)), the transaction fees have a non-trivial cost for the user.

For a Miner, these fees contribute to his/her profit. For example, the average revenue per block[2] (in September 2018) was 3.48 Ether ($785.58 USD[3]). From that amount, 0.48 Ether ($108.35 USD) was just in transaction fees. Every Miner can set his/her own minimum fees (measured in Gas price) and there is no consensus on what value should that be (**?**).

In the period considered in this research (from December 1, 2018 to December 15, 2018), the Ethereum transaction fees paid to have the transaction mined in approximately 30 seconds (i.e., two blocks in the Ethereum platform) noticeably changed from 60 GWei to 20 GWei[4]. Figure 4.1 shows the Gas prices estimated by the Etherchain's API[5] in 8 hours.

[1] `https://coinmarketcap.com/coins`, 2018-09-28.

[2] `https://bitinfocharts.com/ethereum/`, 2018-09-28.

[3] We are assuming an exchange rate of 1 Ether = $225.742791 USD, based on the values by `https://currencio.co/eth/usd/` at 2018-09-28.

[4] 1 GWei = $10^{-9}$ Ether

[5] `https://www.etherchain.org/api/gasPriceOracle`

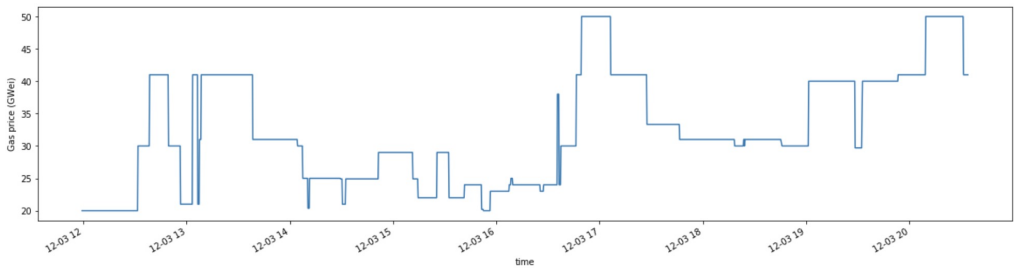



Figure 4.1: Ethereum transaction fees variation

It is essential for a user to get a good estimate of the price s/he should set for his/her transaction to be approved in a given time. We assume that users do not want to overpay for Gas. Moreover, if a user does not need that his/her transaction is approved quickly, s/he may wait until it is possible to pay less and thus save money.

The variation of Ethereum transaction fees poses some questions worth considering, such as: what are the factors that influence the Ethereum transaction fees? What is the relationship between the Ethereum transaction fees and other economic indicators? Is there any connection between the Ethereum transaction fees and the number of pending transactions in Ethereum? Is there any connection between the Ethereum transaction fees and the Miners' policy, such as the minimum Gas price and the maximum Gas limit for mining a transaction?

The aim of the research is to investigate the factors that influence the Ethereum transaction fees and therefore the possible decision making behaviour of blockchain users, miners included. The results are relevant not only from a computer science perspective but also from an economic perspective, because blockchain is a technology adopted by an increasing number of institutions, because it can coordinate strategic activities across different sectors: global agrifood chains, healthcare system (**?**), banks, insurance and entertainment companies, etc. Section 4.2.1 presents previous literature on the same research topic. Section 4.2.2 presents the main research question of the work. Section 4.2.3 describes some aspects of the Ethereum protocol and Granger causality, needed to understand the relationship among variables that might influence the Ethereum transaction fees. Section **??** presents the methodology used to gather and analyze the data from Ethereum, to evaluate the variables that affect the Ethereum Gas fees. Section 4.2.5 presents and discusses the results. Section 4.2.6 presents the conclusions of the research.

### 4.2.1 Related Work

Several studies examined factors that influence cryptocurrencies prices and fees (**?????**). Sovbetov (**?**) examines factors that influence the five cryptocurrencies Bitcoin, Ethereum, Dash, Litecoin, and Monero, over 2010-2018 using weekly data.

Giudici and Abu-Hashish (**?**) propose a new model that explains the dynamics of bitcoin prices and models the interconnections among different crypto and classic asset prices. Houy (**?**) analyses the economics of Bitcoin's transaction fees and shows that a fixed and imposed transaction fee can keep Bitcoin blockchain secure enough when the transaction fee is high enough. Möser and Böhme (**?**) analyse the transaction fees paid within 45.7 million transactions recorded in the public Bitcoin blockchain from the inception of Bitcoin until the end of August 2014. They interpret the heterogeneity and instability of transaction fees as an indication that the protocol's market mechanism fails to set a fair price for transactions. Easley et al.(**?**) develops a game-theoretic model to explain the factors leading to the emergence of transactions fees, as well as the strategic behaviour of miners and users. He highlights the role played by mining rewards and by transactions volume.

### 4.2.2 Research question

Previous studies focus on the factors that influence cryptocurrencies prices and fees in a daily, weekly or monthly time frame, while the present study considers a narrower time frame, in seconds. Moreover, previous literature especially considers the Bitcoin blockchain, while the present study investigates the Ethereum blockchain. Figure 4.1 shows that the Gas prices estimated by the Etherchain's API changes many times in just 8 hours ranging from 20 to 50 Gwei. Therefore, we decided to analyse the variables in a 15 seconds time frame. We chose this interval of time, because it is the average time to mine a block in the ethereum network, as shown by the row block_time in Table 4.3. We thought this is the best way to answer our main research question: what are the factors influencing the transaction fee price in the time frame to mine a block?

### 4.2.3 Background

#### Ethereum protocol

The Ethereum protocol (**?**) defines how the Ethereum network works, how Miners should generally operate, and rules everyone must follow to be a valid part of the network. The protocol is written in general terms such that anyone could implement his/her own version of the protocol into a custom Ethereum client. The most used Ethereum clients are Go-Ethereum and Parity-Ethereum. Go-Ethereum, named Geth, is written in GO, Parity-Ethereum is written in Rust. Figure 4.2 is a pie chart showing the most used Ethereum clients[6].

Through the Ethereum clients the Miners can set the conditions that the transactions must satisfy to be accepted and transmitted in the network. For example the Miners can set the minimum Gas price to mine a transaction and the amount of Gas per block to target when mining a new block. Each Ethereum mining client

[6] `https://www.ethernodes.org/network/1`

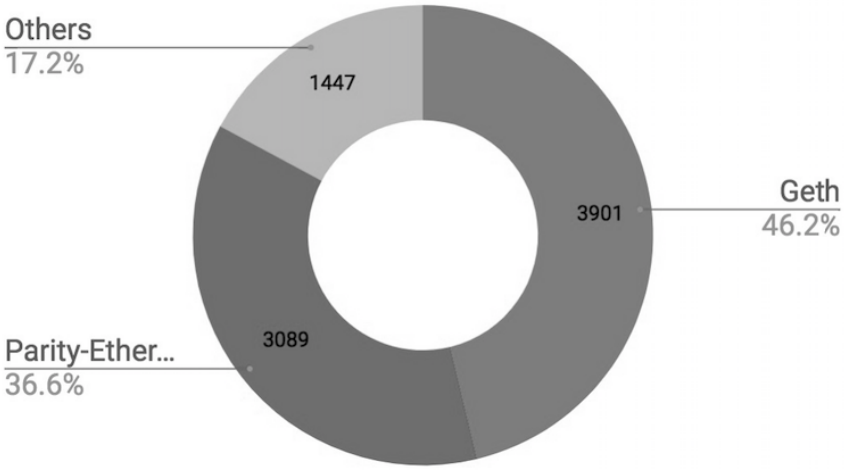



Figure 4.2: Ethereum clients

has different default values to mine transactions and blocks. For example, the Geth Ethereum client has 2 GWei as the minimum Gas price and 4,712,388 as the maximum amount of Gas per block. The Miners can change the settings according to their needs. For example, to gain more money, a Miner can set a higher minimum Gas price or, to prevent malicious denial of service (DoS), a Miner can lower the block Gas limit to 2 million units of Gas.

### Life cycle of an Ethereum transaction

Figure 4.3 presents the life cycle of Ethereum transactions. These are the main stages of the transactions workflow:

1. A user logs into his/her Ethereum account and sends his/her transactions to the Ethereum network, a set of interconnected nodes.
2. Some nodes receive the transactions and each one can pass them to nearby nodes.
3. The nodes that can mine the blocks, i.e. the Miners, select the transactions going to the mempool, according to its settings.
4. A miner picks the transactions up from the mempool, puts them in a block and tries to find the nonce value representing a correct solution to a cryptographic problem.
5. The first miner that finds a solution for its block, broadcasts the solution to all the other nodes.
6. The nodes that received the solution, verify whether it corresponds to the problem of the senders' block. If the solution is correct, the other nodes can confirm that the block can be added to the blockchain.

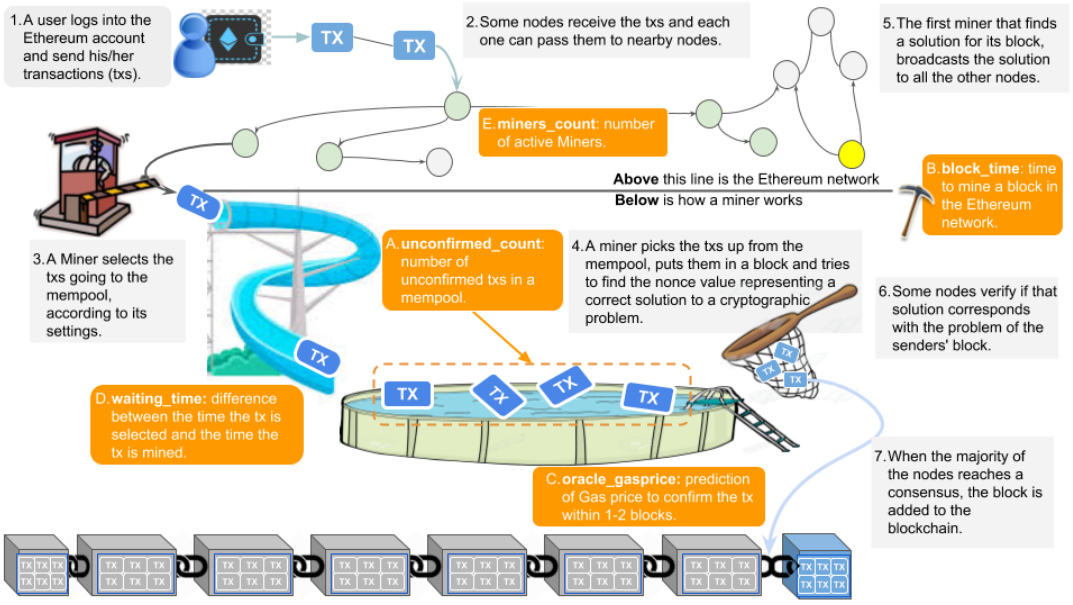



Figure 4.3: Life cycle of an Ethereum transaction (tx). Orange boxes represent the variables possibly influencing the Ethereum txs fee. Grey numbered boxes represent the stages of the txs workflow.

7. When the majority of the nodes reaches a consensus, the block is added to the blockchain.

### Gas Price Oracle

The Etherchain Gas Price Oracle[7] is a tool that provides a prediction on the fairest Gas price to pay to get a transaction confirmed within a certain number of blocks. It uses the method developed by EthGasStation[8] to estimate the prices. We decided to use Etherchain instead of EthGasStation mainly because Etherchain provides its oracle data as a REST service, which is easier to acquire automatically. EthGasStation does not provide such interface requiring a manual or text-mining interaction to acquire its prices.

The Etherchain tool provides four recommended Gas prices based on the desired transaction speed and cost: "safe low", "average", "fast", and "fastest".

- "Safe low" is the Gas price intended to be both cheap and successful. It may take a bit longer to get a transaction confirmed with this price, but anyway less than 30 minutes.

- "Average" is the price accepted by the top Miners who account for at least 50% of the blocks. It takes around five minutes to get a transaction confirmed

[7] `https://www.etherchain.org/api/gasPriceOracle`
[8] `https://ethgasstation.info/`

with this price.

- "Fast" is the price accepted by the top Miners which takes approximately one minute to get a transaction approved using this price.
- "Fastest" is the lowest Gas price that is accepted by all top Miners (estimated over the last two days). It takes at most 30 seconds to get a transaction confirmed with this price. Paying more than this price, it is unlikely to decrease transaction confirmation time.

The research will consider the "fastest" prices, as a variable named "oracle_gasprice" from now on. Figure 4.4 shows the "oracle_gasprice" variation history during the day. The "oracle_gasprice" recommendations are based on the lowest Gas price accepted by the Miners in the last 200 blocks. The goal of the research is to understand whether other variables can affect the "oracle_gasprice" values, *i.e.,* the alledged fairest Ethereum transaction fees to pay.

### Granger causality

To understand whether the data series on one variable affects the data series on the other variable, a specific relationship among the series needs to be observed. A time series variable is called causal to another if the ability to predict the second variable is improved by incorporating information about the first one. The notion of causality was first proposed by Wiener (**?**). Granger causality is a technique to determine whether one time series is useful in forecasting another. Granger (**??**) defined causality as follows: A variable Y is causal for another variable X if knowledge of the past history of Y is useful to predict the future state of X in addition to the knowledge of the past history of X itself. So if the prediction of X is improved by including Y as a predictor, then Y is said to be Granger causal for X.

Granger causality between two variables can be unidirectional, bidirectional (or feedback) and neither unidirectional nor bidirectional, *i.e.,* independent or without Granger-causality in any direction. As to what concerns the results of this research, the Granger causality test is calculated for different lags from 0 to 24 lags. Each lag corresponds to a time interval of 15 seconds. It means that, if the variable x Granger causes the variable y within one lag, it will need 15 seconds before the variable x affects the variable y.

### Augmented Dickey-Fuller test

Granger causality test can be applied only to statistically stationary time series. A stationary time series is a series whose statistical properties, such as mean, variance, etc., are all constant over time. Most statistical forecasting methods, the Granger causality test included, are based on the assumption that the time series can be made approximately stationary (i.e., "stationarized") via mathematical transformations.

Table 4.1: The variables studied to investigate whether they were predictive of oracle_gasprice

| Variable name | Description |
|---|---|
| **oracle_gasprice** | Gas paid to have the transaction confirmed within 1 to 2 blocks time |
| **unconfirmed_count** | Number of unconfirmed transactions in a particular memory pool |
| **block_time** | Time spent to mine a block in the Ethereum network |
| **miners_count** | Number of active Miners |
| **hashrate (Hash/s)** | Speed at which a miner solves the Ethereum code |
| **difficulty** | Number expressing 'how difficult' it is to find a new block |
| **eth_btc** | Value of the BTC/Ether ratio |
| **eth_usd** | Value of the USD/Ether ratio |

If the time series are non-stationary, then the time series model should be applied to temporally differenced data rather than to the original data. Augmented Dickey-Fuller test (ADF) shows whether time series have some upward or downward trend or seasonal effects, i.e. whether mean or variance are not constant over time.

### 4.2.4 Methodology

The selection of variables taken into account in this study is based on different works related to factors that influence cryptocurrencies prices, such as Bitcoin, presented in(**?????**) and listed in Table 4.1. Data were analysed along the following variables: the Gas price oracle (oracle_gasprice), the number of unconfirmed transactions (unconfirmed_count), the block time (block_time), the number of active Miners (miners_count), the current hashrate of the network, a unit measured in hashes per second or H/s (hashrate), the current difficulty of the network (difficulty), the value of the USD/Ether ratio (eth_usd), the value of the BTC/Ether ratio (eth_btc).

- The variable oracle_gasprice, measured in Wei, is the Gas paid to have the transaction confirmed within 1 to 2 blocks time (around 15/30 seconds). In our case it is the Ethereum transaction fees.
- The variable hashrate it is the speed at which a miner solves the Ethereum code. In December 2018, the hash rate of the network was approximately 300 billion H/s or 300 GH/s.
- The variable unconfirmed_count refers to the number of unconfirmed transactions in a particular memory pool. The number of waiting transactions of a particular memory pool differ from the total number of waiting transactions in the Ethereum network. Moreover, the memory pool of each node might

Table 4.2: RESTful Services list

| Variable name | Service Name | URI |
|---|---|---|
| **oracle_gasprice** | Etherchain's API | https://www.etherchain.org |
| **unconfirmed_count** | BlockCypher's Ethereum API | https://api.blockcypher.com |
| **block_time** | Ethpool's API | https://api.ethpool.org |
| **miners_count** | | https://api.ethpool.org |
| **hashrate (H/s)** | | https://api.ethpool.org |
| **difficulty** | | https://api.ethpool.org |
| **eth_btc** | Etherscan's API | https://api.etherscan.io |
| **eth_usd** | | https://api.etherscan.io |

differ from the memory pool of other nodes: while there is a consensus on the mined transactions, there is no enforced consensus on what is stored in the memory pool of each node. However, for the aims of this research, it is reasonable to assume that the waiting transactions trend in the memory pool is representative of the general waiting transactions trend in the global network.

- The variable difficulty indicates how difficult it is to find the hash of a new block. The difficulty is adjusted periodically as a function of how much hashing power has been deployed by the network of miners.

- The variable block_time refers to the time spent to mine a block in the Ethereum network.

The data are sampled each 15 seconds and were sourced from the different RESTful services listed in Table 4.12.

### Dataset

For each variable of the dataset there are 92,160 observations, collected from December 1, 2018 to December 15, 2018 (Table 4.12). The dataset is publicly available at Github.[9]

Table 4.3 shows statistics on our dataset. For each variable we measured the mean, the standard deviation (SD), minimum (min), the 25th, 50th, and 75th percentiles and maximum (max).

The data were analyzed to determine the Granger causality between a specific variable, *i.e.,* oracle_gasprice (the Gas price to have the transaction mined in 2 blocks max), and all the other variables of Table 4.3. A test was previously conducted to

[9]https://github.com/apierr/gas-price

Table 4.3: Statistical description of sample data

| | **Mean** | **SD** | **min** | **25%** | **50%** | **75%** | **max** |
|---|---|---|---|---|---|---|---|
| **oracle_gasprice** | 33.59 | 8.83 | 20 | 26 | 31 | 41 | 60 |
| **unconfirmed_count** | 91,768 | 6,312 | 70,772 | 88,429 | 92,915 | 95,582 | 111,252 |
| **block_time** | 14.46 | 0.91 | 11.70 | 13.8 | 14.4 | 15.1 | 17.5 |
| **miners_count** | 785 | 58.64 | 699 | 747 | 770 | 803 | 1,031 |
| **hashrate (TH/s)** | 173 | 4.15 | 167 | 169 | 175 | 177 | 181 |
| **difficulty ($\div 10^{15}$)** | 2.37 | 0.0718 | 2.22 | 2.32 | 2.37 | 2.44 | 2.55 |
| **eth_btc** | 0.0268 | 0.0008 | 0.0246 | 0.0263 | 0.0267 | 0.0277 | 0.0283 |
| **eth_usd** | 98.41 | 8.64 | 82.56 | 91.66 | 96.64 | 107.33 | 113.78 |

ensure that the data were stationary in terms of the variables used. In the case of non-stationary series, the first difference of the series was performed just once to make the series stationary. A pair Granger causality test was then performed for all the series.

### 4.2.5 Results and Discussion

When we analyzed the data, some series were stationary (ex. data of oracle_gasprice variable), whilst other series revealed to be non-stationary (ex. data of eth_usd variable). Figure 4.4 is a graphical representation of the stationarity vs. non stationarity of raw data concerning a sample of variables, *i.e.,*hashrate, difficulty, eth_usd, eth_btc, unconfirmed_count, miners_count, block_time, and oracle_gasprice.

In Figure 4.4, the first five (hashrate, difficulty, eth_usd, eth_btc, and unconfirmed_count) are non-stationary series, whilst the remaining represents stationary series (miners_count, block_time, and oracle_gasprice).

#### ADF test

To confirm whether the series were stationary or not, we perform a Augmented Dickey-Fuller (ADF) test. ADF test results show that the data series on variables hashrate, difficulty, eth_usd, eth_btc, and unconfirmed_count are all non-stationary. However they are all stationary after the first differentiation, therefore it is possible to state that they are integrated at the first order. The ADF test results are summarized in Table 4.4.

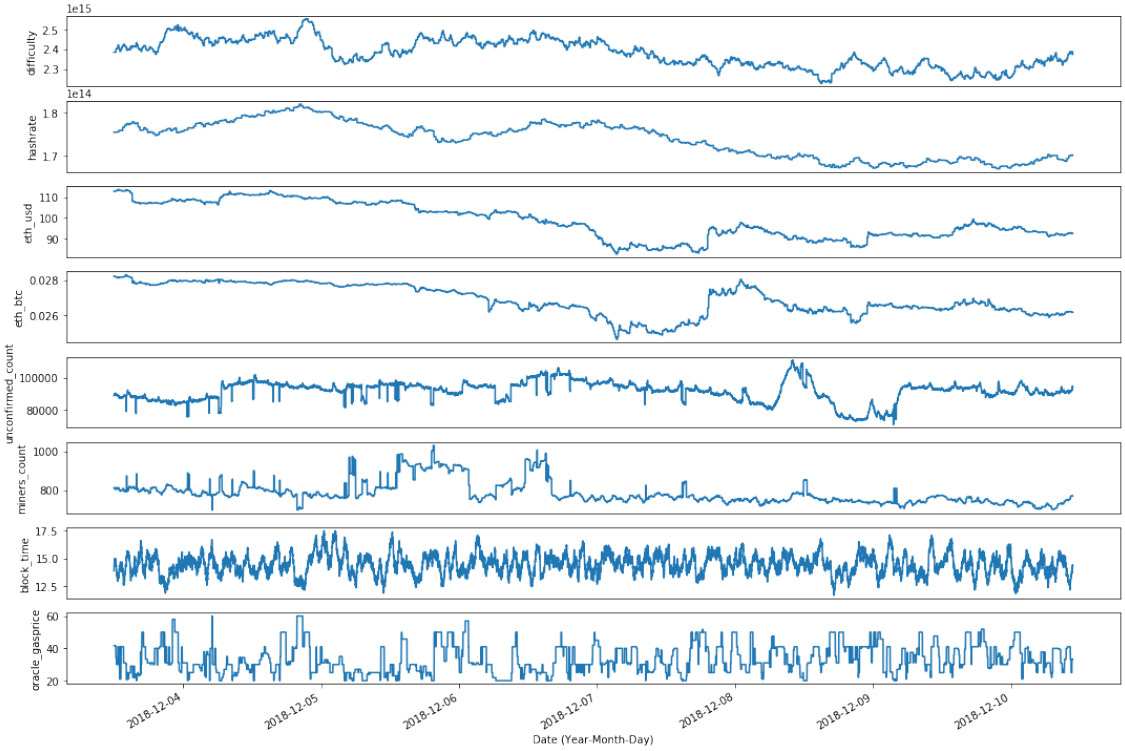



Figure 4.4: Time series datasets

Table 4.4: ADF test results

| Variables | **ADF Statistic** | **p-value** | **Conclusion** |
|---|---|---|---|
| **oracle_gasprice** | -10.00 | **0.000*** | stationarity |
| **unconfirmed_count** | -2.958 | 0.058 | non-stationarity |
| **block_time** | -10.92 | **0.000*** | stationarity |
| **miners_count** | -4.427 | **0.001*** | stationarity |
| **hashrate** | -0.672 | 0.853 | non-stationarity |
| **difficulty** | -0.105 | 0.948 | non-stationarity |
| **eth_btc** | -1.928 | 0.318 | non-stationarity |
| **eth_usd** | -1.686 | 0.437 | non-stationarity |

*$p < 0.05$ means that the null hypothesis is rejected, indicating that the data are stationary

### Granger causality test

The pair-wise Granger causality test was thereafter performed for oracle_gasprice variable versus the variables listed in Table 4.5.

Table 4.5: Granger causality test results

| Null Hypothesis: | F-stat. | Prob. | Decision |
|---|---|---|---|
| unconfirmed_count does not Granger cause oracle_gasprice | 2.6274 | 0.0723 | Accepted |
| **oracle_gasprice does not Granger cause unconfirmed_count** | 5.2830 | **0.0215*** | **Reject** |
| eth_usd does not Granger cause oracle_gasprice | 0.6094 | 0.5437 | Accepted |
| oracle_gasprice does not Granger cause eth_usd | 0.1961 | 0.6579 | Accepted |
| eth_btc does not Granger cause oracle_gasprice | 0.5923 | 0.4415 | Accepted |
| oracle_gasprice does not Granger cause eth_btc | 1.5087 | 0.2193 | Accepted |
| **miners_count does not Granger cause oracle_gasprice** | 1.0331 | **0.0309*** | **Reject** |
| oracle_gasprice does not Granger cause miners_count | 0.0035 | 0.9527 | Accepted |
| difficulty does not Granger cause oracle_gasprice | 1.2373 | 0.2499 | Accepted |
| oracle_gasprice does not Granger cause difficulty | -32186 | 1.0000 | Accepted |
| block_time does not Granger cause oracle_gasprice | 1.8749 | 0.0323 | Accepted |
| oracle_gasprice does not Granger cause block_time | 0.3624 | 0.5472 | Accepted |
| hashrate does not Granger cause oracle_gasprice | 1.9968 | 0.1576 | Accepted |
| oracle_gasprice does not Granger cause hashrate | -7.2403 | 1.0000 | Accepted |

*$p < 0.05$ means that the null hypothesis is rejected, indicating that the effect of the lagged values (value coming from an earlier point in time) of the other variable is statistically significant.

### Pair-wise comparison: oracle_gasprice vs. hashrate, difficulty, block_time, eth_btc, and eth_usd

The results in Table 4.5 show that the data series on the variable hashrate does *not* Granger cause the data series on the variable oracle_gasprice, because the p-value (0.1576) is not significant. They also show that the data series on the variable oracle_gasprice does *not* Granger cause the data series on the variable difficulty, because the p-value (1.000) is also not significant. To sum up, there is no Granger causality between the series, running from oracle_gasprice to difficulty and the other way, because difficulty does not affect oracle_gasprice and the converse is also true. Hence, the Granger causality is independent or non-directional between the two series.

Similar results were obtained for the hashrate variable. Indeed, when the hashrate increases, as a result, also the difficulty of validating newly added blocks increases; when the hashrate decreases, also the difficulty decreases, to speed up the time needed to mine a block. Based on the total hashrate, the difficulty is adjusted

by the nodes and the time to mine a block is kept constant (15 seconds). This is the reason why these variables do not influence the transactions fees.

Finally, we obtained similar results for the relationship between oracle_gasprice and the following variables: eth_btc, eth_usd, and block_time.

#### Pair-wise comparison: oracle_gasprice vs. unconf_count

The results in Table 4.5 also show that the data series on the variable oracle_gasprice *does* Granger cause the data series on the variable unconfirmed_count, because the p-value (0.0215) is significant. They also show that the data series on the variable unconfirmed_count does not Granger cause the data series on the variable oracle_gasprice, because the p-value (0.0723) is not significant. To sum up, there is Granger causality between the series, running from oracle_gasprice to unconfirmed_count, but not the other way, because unconfirmed_count does not affect oracle_gasprice. Hence, the Granger causality is unidirectional between the two series.

#### Pearson correlation: unconf_count vs. oracle_gasprice

To better understand the unidirectional relationship of Granger causality between oracle_gasprice and unconfirmed_count variables, a Pearson correlation test was performed. The Pearson correlation coefficient is equal to -0.6. Therefore the Pearson correlation test result suggests that there is an inverse relationship between oracle_gasprice and unconfirmed_count. A plausible explanation for this result is that, when the oracle suggests a high price, users are not encouraged to submit the transactions. It is indeed reasonable to assume that people that "can" wait, will do so if the Oracle price is too high.

#### Pair-wise comparison: oracle_gasprice vs. miners_count

Furthermore, the results in Table 4.5 show that the data series on the variable miners_count *does* Granger cause the data series on the variable oracle_gasprice, because the p-value (0.0309) is significant. They also show that the data series on the variable oracle_gasprice does *not* Granger cause the data series on the variable unconfirmed_count, because the p-value (0.9527) is not significant. To sum up, there is Granger causality between the series, running from miners_count to oracle_gasprice and not the other way, because oracle_gasprice does not affect miners_count and the converse is not true. Hence, the Granger causality is unidirectional between the two series.

#### Pearson correlation: miner_count vs. oracle_gasprice

To better understand the unidirectional relationship of Granger causality between miner_count and oracle_gasprice variables, a Pearson correlation test was performed.

The Pearson correlation coefficient is equal to -0.41. Therefore the Pearson correlation test result suggests that there is an inverse relationship between oracle_gasprice and miner_count: the more the number of miners, the lower the price predicted by the Oracle. A plausible exaplanation might come from the fact that an increase of the number of miners entails an increase of the mining competition to get the transactions. As a consequence of the supply/demand balance, the increase in competition entails a decrease of the prices.

### 4.2.6 Summary and Conclusions

There is much work (?????) investigating the factors influencing cryptocurrencies prices and fees, focusing on the most common cryptocurrencies, such as Bitcoin. Previous studies analyzed the factors on weekly or daily data. On the other hand, this research analyzes instead Ethereum and adopts a finer time frame, to investigate the factors that influence the Ethereum transaction fees in the average time to mine one block (i.e., approximately 15 seconds).

The main objective of the research is to analyze the Granger causality relationship between the data series on the variable oracle_gasprice and other variables, such as unconfirmed_count, miners_count and eth_usd. A ADF test and a pair-wise Granger causality test were performed to establish whether there is a Granger causality between the data series on the variable oracle_gasprice and the data series on other variables.

The results of the ADF test showed that the data series on eth_usd and unconfirmed_count did not present any stationarity. After taking the first difference of the series, the results of the ADF test showed a stationarity.

In light of the results of the pair-wise Granger causality test (see Table 4.5), a non-directional causality relationship was observed between the data series on the oracle_gasprice variable and the data series on the block_time variable and between the data series on the oracle_gasprice variable and the data series on the hashrate variable. This means that the past history of both the oracle_gasprice and the block_time variables cannot help in respectively predicting their future values.

In a similar vein, it is possible to conclude that ethtbc, eth_usd, hashrate and difficulty variables cannot be used to forecast the values of the oracle_gasprice variable, and also the converse is true (see Table 4.5).

Interestingly, a unidirectional causality was observed from the data series on the oracle_gasprice variable to the data series on the unconfirmed_count variable (see Table 4.5). This result shows that the past history of the oracle_gasprice variable is useful to forecast the number of waiting transactions, even though the converse is not true. The results of the Pearson correlation test showed that they are inversely correlated: when the oracle price increases, the number of waiting transactions in the Ethereum network decreases. It stands to reason that when the oracle suggests a high price to pay, the users wait to submit a transaction, thus decreasing the overall number of pending transactions in their memory pools. As to what con-

cerns the results of this research, the oracle_gasprice variable Granger causes the unconfirmed_count variable when the number of lags is greater than 6. This means that the oracle_gasprice variable does not immediately affect the unconfirmed_count variable. This result is compatible with the fact that the user cannot be immediately aware of the variation of the oracle_gasprice. Moreover we need to consider the time taken by the user to submit the smart contract to the blockchain and the time required by the transactions to propagate in the network. When choosing a time frame greater than 15 seconds, it is not possible to appreciate the time needed for the system to equilibrate the variation of a variable.

Finally, a unidirectional causality was also found from the data series on the miners_count variable to the data series on the oracle_gasprice variable (see Table 4.5). The result of the Pearson correlation test showed that the number of Miners and the oracle_gasprice variables are inversely correlated: when the number of Miners increases, the oracle price decreases, as per the supply/demand balance.

Overall, the results of the research are useful to improve the predictions on the Ethereum transaction fees at a given time, because they shed a light on how different variables might interact and influence the Oracle Gas Price. Knowing that Oracle predictions are biased by some variables might be useful for blockchain users, to reach a more mindful and efficient use of the platform. Such results are relevant not only from a computer science perspective but also from an economic perspective, because they show the financial mechanisms of blockchain which is adopted by both public and private institutions. Further research is anyway needed to build a model that precisely provides the users with an estimate of the best Gas Price to pay to have the transaction executed in a given time lapse, taking into account the variables that influence the overall Ethereum transaction fees.

## 4.3 Are the Gas Prices Oracle Reliable?

### 4.3.1 Introduction

In Ethereum (**?**), users need to pay a fee in a special resource called Gas when creating transactions (**???**). Gas is like fuel for computational instructions executed in the blockchain. There are mainly three reasons for the Gas fees concept: (i) to make the users pay for the computation costs (e.g., energy, CPU) required to create and approve their transactions; (ii) to limit blockchain resource use; and (iii) to avoid issues of intentional or non-intentional network abuse (e.g., DoS attacks, infinite loops).

Transactions occurring in the network are verified by special nodes named "miners". In Ethereum, verifying a transaction means checking the sender and the content of the transaction. Miners generate a new block of transactions and then add such a block to the network. Currently, miners need to solve a mathematical puzzle (called "Proof of Work") to create a new Ethereum block. Miners receive the Gas

transaction fees converted into cryptocurrency as a reward for adding a new block to the blockchain (**???**).

The Gas price value is set by the user who chooses how much to pay to execute the transaction. If the value set by the user is too low for the Ethereum miners, the transaction risks to never be included in the blockchain. On the other hand, if the transaction price is very high, the blockchain miners will be prone to include it in the Ethereum blockchain, but the user will allegedly waste money.

In this work, we analyze the data of one popular oracle to predict the Gas price, along with the Ethereum transactions' and blocks' data. More specifically, we use in this study the *EthGasStation* oracle, as its API is public and can be used by any other oracle or user. The Ethereum transactions' variables considered in the study are:

- the interval of time elapsed between the time when the transaction was first seen in the Transaction Pool and the time in which the transaction was added to the Ethereum Blockchain;

- the Gas price, i.e., the amount of Ether the user is willing to pay for every unit of Gas, which is measured in "GWei".

Oracle data is useful to predict the Gas price a user should pay to influence miners to verify a transaction (and consequently, add such transaction to a block). To help users decide on the price to pay to submit a transaction, Gas Oracle proposes the following four price categories: safe low, standard, fast, and fastest. These categories define the Gas price required to have a transaction confirmed within the next 100, 20, 5, and 2 blocks, respectively.

First, the results show that EthGasStation gives the Gas price prediction with a higher margin of error compared to what it claims (2%). The margin of error ranges from a minimum of 4% for the "fastest" category to a maximum of 28% for the "fast" category. Second, we argue that by performing the Poisson regression more frequently, the margin of error can, in theory, be decreased to the declared mark of 2% for the "fastest" category. Finally, the results suggest that two of the Gas Oracle categories are not frequently used in practice: fast and average categories. It is indeed reasonable to expect that, to save money, single users or companies could set different requirements in terms of interval time to add a transaction, that is not provided by all the default categories.

This study is therefore relevant from a users' perspective for at least two reasons:

- It shows the EthGasStation oracle is less reliable than advertised. Therefore, users or companies employing Ethereum blockchain technology should be more careful to trust oracles' recommendations.

- It suggests that two of the four categories proposed by the Gas Oracle may not meet the needs of Ethereum users. Therefore other categories could be

created better suit the users' requirements. For example, a company might be interested to record a transaction with a maximum delay of 24 hours. Such a category, not considered by the Oracle, might help the company to save money in transaction fees.

The rest of the work is organized as follows. Section 4.4.2 described the transaction pool and the Gas Oracle investigated in the research. Section 4.3.3 presents the experimental design and methodology used to collect and analyze the data. Section 4.3.5 discusses the results of the case study using EthGasStation Oracle. Section 4.3.5 describes the threats for the validity of the study. Section 4.4.3 presents the related work. Finally, Section 4.4.9 presents the conclusions and outlines future work ideas.

### Background

In this section, we provide background information needed to better understand our study.

### Block

The blockchain is an ordered list of blocks, where each block is identified by its cryptographic hash and a progressive number named "height" (**?**). Each block refers to the block preceding it, resulting in a chain of blocks. Each block consists of a set of transactions. Once a block is created and attached to the blockchain, the transactions in the block cannot be changed or reverted. This is to ensure the integrity of the transactions and to prevent the double-spending problem (**?**).

The "block time" is defined as the interval of time the blockchain takes to mine a block. The time interval is not constant and changes every time a block is added to the blockchain. In Ethereum blockchain the block time is expected to be between 10 to 19 seconds (with an average of 15 seconds). A standard unit of measurement for time in the blockchain is not the second but the block number.

### Transaction Pool

Each node in the Ethereum network has a virtual place named "Transaction Pool", where transactions enter when they are received from the network or submitted locally. The Transaction Pool contains all currently known pending/unconfirmed transactions. They exit the Transaction Pool when they are included in the Ethereum blockchain. The miners separate processable transactions, which can be added to a block, and future transactions, which can be wait to be added. Transactions move between those two states over time as they are received and processed.

Each node maintains its own Transaction Pool. When a node receives a new valid block, it removes all the transactions contained in the block from its Transaction Pool as well as the transactions which attempt to double spend the same output. A

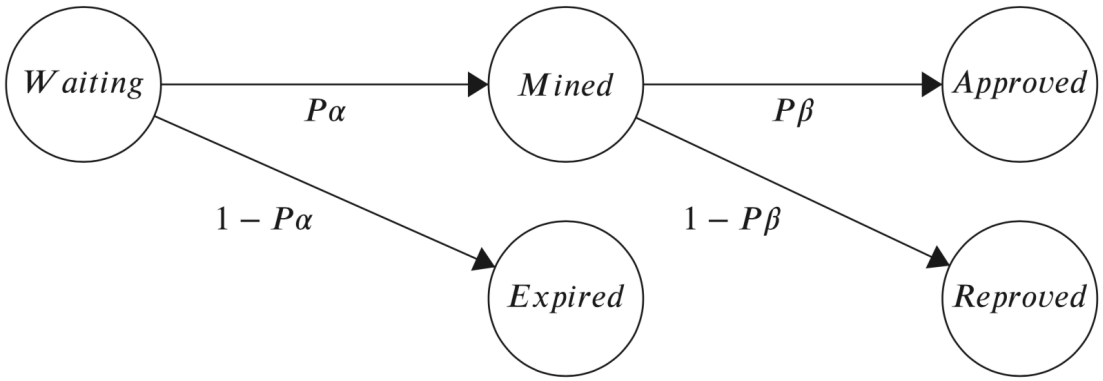



Figure 4.5: Life cycle of a transaction. $P_\alpha$ is the probability of a transaction in the memory-pool to be mined, and $P_\beta$ is the probability of a mined transaction to be approved.

double spend is a potential flaw in a digital cash scheme in which the same single digital token can be spent more than once. The node can decide different policies: for example, if the Transaction Pool size gets too close to the RAM capacity, the node can set up a minimal fee threshold. Transactions with Gas price lower than the threshold are immediately removed from the Transaction Pool and only new transactions with a Gas price high enough are allowed to enter the transaction pool.

### Life Cycle of a Transaction

To understand the concept of the data collection performed in the research, it is necessary first to present the possible states of a transaction. Figure 4.5 illustrates the life cycle of a transaction. A transaction life cycle starts when it is first observed in a vantage point, a memory-pool, in the Ethereum network. A user offers a reward to a miner to process its transaction and waits for the final results.

Figure 2.1 illustrates the compression property of hash functions.

The transaction approval depends on the amount of Gas offered by the user to be sufficient to act on the network. When the user offers a sufficient amount of Gas, miners can process this transaction, and as a consequence, the user will have its transaction approved. Similarly to a car, a sufficient amount of Gas allows for the transaction to reach its final state successfully. Otherwise, if a miner spends the amount of Gas and is not able to process the transaction, the transaction is discharged and stated as reproved. Some transactions were not mined during the time window of our observation. In fact, we observed some transactions in the memory-pool, and even a long time after this first observation, we were not able to decide if this transaction is considered either approved or reproved. When such a situation occurs, we consider the transaction as expired.

### 4.3.2 Gas Oracle

An Oracle is a software that finds and analyses data concerning real-world facts. Based on the data, it computes an estimate, extracting relevant information to predict future data trends. Examples of real-world facts are commodities and goods prices, flight or train delays. In the Ethereum blockchain, the information provided by an oracle can be used by smart contracts the participants have agreed on, to execute the transactions.

In the context of this research, Gas Oracle is an oracle that analyses blockchain data to predict the best Gas price to pay for a transaction to be approved within a certain number of blocks. We analyze data from one specific Gas Oracle, ETH-GasStation.

This oracle claims that all predicted values are estimations based on the current network condition and should be used as an indication. More specifically, ETH-GasStation employs a Poisson Regression (**?**) on Ethereum data to estimate the Gas prices accepted by the miners. However, it only computes and updates its predictions every 100 blocks (approximately 1,500 seconds or 25 minutes). Therefore, the estimation made by the oracles every 100 blocks may not reflect the most current status of the network. During this interval of time, some data regarding the blockchain network such as the number of miners, the number of transactions and the Gas price attached to the transactions might indeed suddenly change, thus having an impact on the value of the minimum transaction fee accepted by the miners to include a transaction in a block. All data regarding the blockchain network are publicly available, also in the form of a timeline chart confirming our hypothesis.[10]

### 4.3.3 Experimental Design

We planned to test the reliability of a Gas Oracle by looking over real data from the Ethereum blockchain. The research method consists of four phases: (a) Retrieving Data, (b) Cleaning Data, (c) Modelling Data, and (d) Analyzing Data.

#### Retrieving Data

In this phase, we collect the data by making requests to various HTTP RESTful API services at different times. Figure 4.6 describes the periodic polling used to get new information from the HTTP RESTful API services. The flow collects data as follows:

- a request is sent to the server every 15 seconds;
- if the request is successful, the server responds to the client request sending a payload in JSON format;

[10] `https://etherscan.io/charts`

Table 4.6: RESTful Services list

| Resource name | RESTful API service URI |
|---|---|
| **EthGasStation** | https://ethgasstation.info/json/ethgasAPI.json |
| **Block** | https://api.blockcypher.com/v1/eth/main/blocks/0 |
| **Unconfirmed Transactions** | https://api.blockcypher.com/v1/eth/main/txs |

- if the request is not successful, the client will not record any data for that time frame. During the data retrieving operation, an average of 1 request out of 20,160 requests was unsuccessful, i.e. 0.0049% of the requests failed.

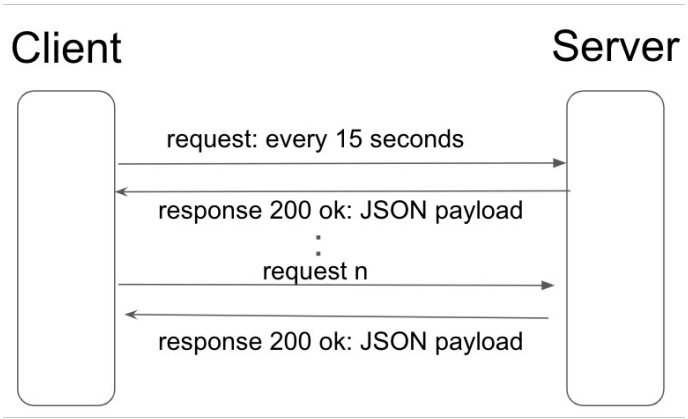



Figure 4.6: Regular Polling every 15 seconds

Table 4.12 shows the URI of the RESTful API services used to fetch the Gas Oracle data, the blocks data and the Unconfirmed Transactions data, i.e., the latest transactions that have not been included in any block.

The data have been stored as files in JSON format in the file system of the server where the analyses are performed. Figure 4.7 shows an example of the Gas Oracles payload formatted in JSON format.

The key value pairs shown in code 4.7 represent respectively the Gas to pay to have the transactions confirmed within: 2 blocks (fastest), 5 blocks (fast), 20 blocks (average), and 100 blocks (safe low).

```
{
    "fastest":116.0,
    "fast":100.0,
    "safeLow":17.0,
    "average":60.0,
    "block_time":13.24,
    "blockNum":8937688
 }
```


Figure 4.7: JSON payload extracted from EthGasStation RESTFul API Services

### Cleaning Data

In this phase, we perform a control of the data quality. The data retrieved have been checked in compliance to the API documentation; Example of data not in compliance to the API documentation are:

- string value where a numeric value was instead expected;
- numeric value where a string value was instead expected;
- numeric value which is not in the expected range;
- date value which is not in the expected time frame;
- missing value;
- missing key/value pairs;
- numeric value with different units of measurement.

We rejected the data, falling in one of the categories listed above, except for the latest category where the values have been recalculated according to the expected measurement unit.

As an example, a numeric value which is not in the expected range, is a negative block's height value, a date value in the future, or a negative value of the waiting time for transactions to be added to the Ethereum blockchain. According to Kanda and Shudo (**?**), a negative value of the waiting time variable may suggest a transaction propagation delay among different nodes. This means that different nodes in the blockchain could see the transaction at different instants of time.

During the cleaning phase, 0.83% of data has been rejected, distributed as follows: 770K out of 11M transactions (0.75%), 182 out of 345K blocks (0.05%) and 112 out of 345K Oracle's predictions of the Gas price (0.03%).

### 4.3.4 Modelling Data

#### Validation Condition

In this step, we define the condition to assess the correctness of the Oracles' Gas price prediction. Oracles usually make the prediction based on the history of the mined blocks data, such as the lowest Gas price accepted by the miner to add the transaction to the block.

Suppose that during the time interval when the i-th block, $B_i$, is mined:

- *op* (Oracle Price) is the price predicted by the Gas Oracle to have the transaction included at the most within j blocks;
- $B = \{B_{i+1}, ..., B_{i+j}\}$ is the set of j blocks mined in the blockchain following the i-th block;
- $T = \{tx_1, tx_2, ..., tx_n\}$ is the set of unconfirmed transactions, i.e. that have not been included in any blocks, with Gas price respectively of $tp_1, tp_2, , tp_n$, that there were in a Transaction Pool when the i-th block was mined.

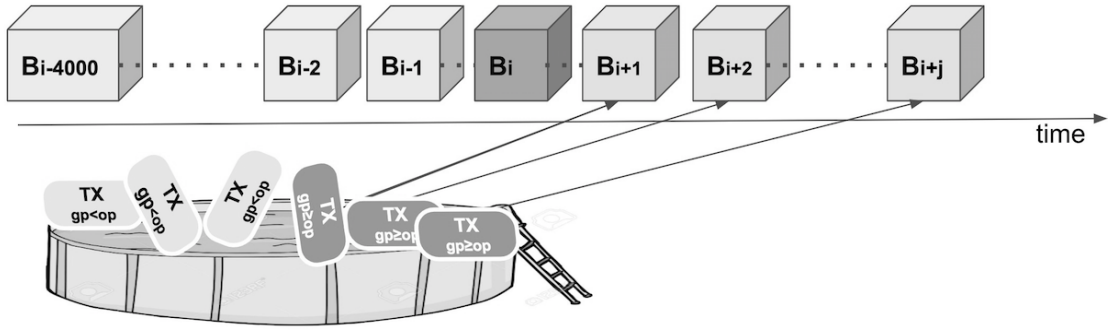



Figure 4.8: Gas Oracle prediction based on block history. The transaction having a Gas price higher or equal to *op* (the yellow circle) are displayed in white text on a darker background. The transaction having a Gas price lower than *op* are displayed in black text on a lighter background.

Figure 4.14 depicts the Ethereum blockchain in the past (grey color $B_{i-4000}...B_{i-1}$), with the addition of the block mined in the present (red color $B_i$) and with future blocks (green color $B_{i+1}...B_{i+j}$). The Gas Oracle prediction is based on the block history of the last 4,000 blocks represented in grey color. The swimming pool represents the status of a Transaction Pool at the time when the Gas Oracle makes the prediction (red block $B_i$). All the transactions belonging to the set $T$ that have a Gas price higher or equal to *op* (the yellow circle) are displayed in green color to the right of the $B_i$ block. If the prediction of the Oracle were correct, all the transactions belonging to the set $T$ having a Gas price higher or equals to *op*, would be mined in one of the following j blocks $\{B_{i+1}, ..., B_{i+j}\}$.

The condition is expressed by the following equation:

$$\forall tx_i \in T \wedge tx_i \geq op : tx_i \in \{B_{i+1}, ..., B_{i+j}\} \tag{4.1}$$

Equation 4.1 is used to verify the prediction of the EthGasStation Oracle Gas price.

### Data Modelling

In this step, data were collected and stored in a relational database, where each table represents the following items: blocks, transactions, and Oracles. Even though, for this study we only considered data from one Oracle, we are planning to expand this research to include other Oracles in the future.

Figure 4.15 shows the data contained in the database, the relationships between table fields and their types (ex. string, integer, boolean, enumerate). It is noteworthy that the wait time for a transaction is not stored in the database but calculated by the difference between confirmed (time) and received (time).

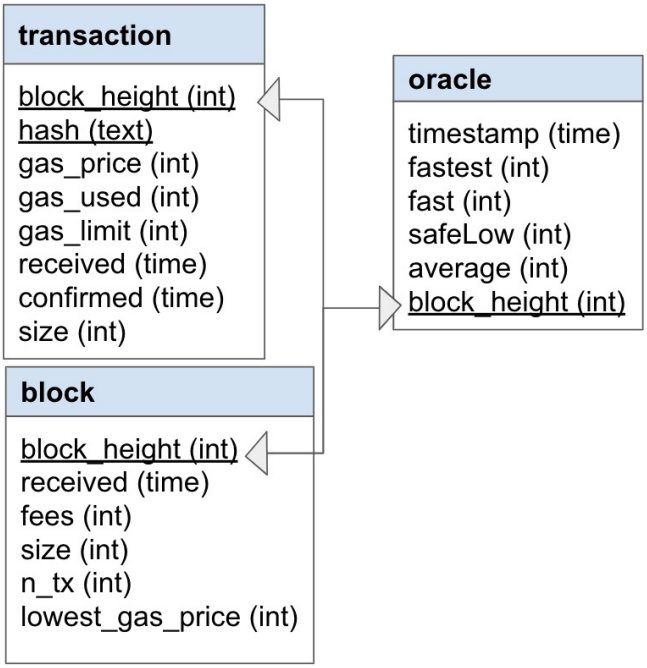



Figure 4.9: Database schema.

## 4.3.5 Analyzing Data

In this phase, we analyse the data to:

Table 4.7: Statistical description of Transactions data

.

| | **Mean** | **St.D** | **Mode** | **Min** | **25%** | **50%** | **75%** | **Max** |
|---|---|---|---|---|---|---|---|---|
| waiting_time (s) | 44 | 82 | 25 | 0 | 25 | 29 | 38 | 1,499 |
| gas_price (GWei) | 32 | 443 | 50 | 0 | 10 | 20 | 50 | 313,734 |
| gas_used | 70,124 | 320,908 | 21,000 | 0 | 21,000 | 21,969 | 49,993 | 8,000,000 |
| gas_limit | 303,967 | 947,926 | 21,000 | 21,000 | 42,000 | 70,000 | 150,000 | 8,000,030 |
| size (Byte) | 191 | 499 | – | 83 | 112 | 114 | 174 | 31,791 |

1. view some aggregated statistical metrics such as percentile, mean, standard deviation, mode of the numerical data series;

2. find distribution of different variables such as: gas_prices and time a transaction needs to wait before being recorded in the blockchain;

The goal is to make a descriptive analysis of the data-sets concerning different items, as modeled in the Section 4.3.4. All the data cover a period of time ranging from March 29, 2019 to May 28, 2019.

### Case Study: EthGasStation

In this section, we present our case study by analyzing oracle data from the EthGasStation. The data-set analysed in this work is publicly available at our open-access repository.[11] The dataset is an SQLite database having three tables. The first table, named "transaction", contains more than 11 millions rows. The second table, named "block", contains around 345 thousand blocks. The last table, named "oracle", contains 345 thousand rows of Oracles' predictions for the Gas price of each category (fast, fastest, average and safe low). The dataset refers to a period of time of two months, ranging from March 29, 2019 to May 28, 2019.

### Transactions Data Analysis

Figure 4.16a shows the box plot of the waiting time in seconds before a transaction is added to the Ethereum Blockchain. The violin plot shows the presence of a time peak along the vertical axis at the value of 20 seconds, which means that, according to the data-set 4.13 analysed in this research, most transactions wait from one to two blocks before being added to the Ethereum Blockchain.

Figure 4.11 shows two violin plots of the waiting time of the transactions for different Gas prices. Interestingly, the violin plots suggest that the Gas price attached to the transaction influences the interval of time the transaction needs to wait before being added to the Ethereum blockchain. The violin plots also present

[11] `http://doi.org/10.5281/zenodo.3584242`

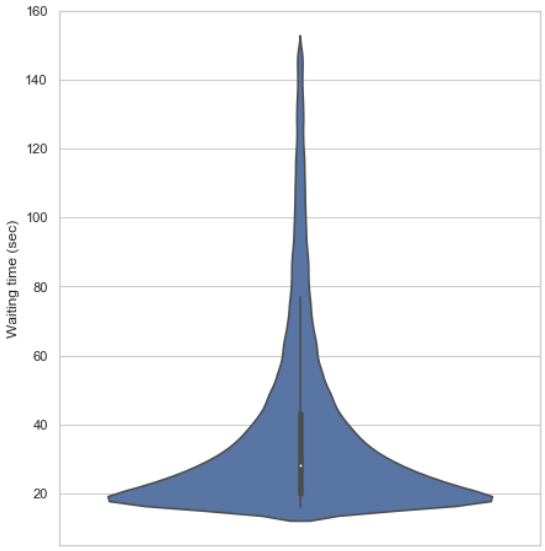



Figure 4.10: Violin Plot (median, first and third percentiles, range) of the waiting time in seconds before a transaction is added to the Ethereum Blockchain.

the same peak at the value of 20 seconds regardless of the Gas price. The difference is the shape of the violin plot, which becomes larger at the decreasing of the Gas price. In addition, having the Gas price higher than 10 Gwei does not guarantee that the transaction is added to the blockchain within 1-2 blocks, but the probability is anyway higher when compared to the transactions having the Gas price lower than 10 GWei.

### Gas Oracle Data Analysis

We analyzed the data of the Oracle EthGasStation. This Oracle can diversely predict the Gas price values to attach to transactions to have the transaction included at the most within $n$ blocks.

Table 4.8 reports the mean, the standard deviation (SD), the minimum (min), the first quartile (25%), the median (50%), the third quartile (75%) and maximum (max) of the Gas price recommendation for the transaction to be included at most in two blocks according to EthGasStation.

Figure 4.12 shows the violin plots of the Gas price prediction according to the EthGasStation Oracle for each Gas price category: 1) fastest (in blue, leftmost plot), 2) fast (in orange, second plot), 3) average (in green, third plot), and 4) safe low (in red, rightmost plot). The violin plot shows that the most frequent value of Gas price for each category is as follows: 20 GWei for the fastest category, 5 GWei for both the fast and average category, and 3 GWei for the safeLow category.

Table 4.15 reports the mean, the standard deviation (SD), the mode, the minimum (min), the first quartile (25%), the median (50%), the third quartile (75%)

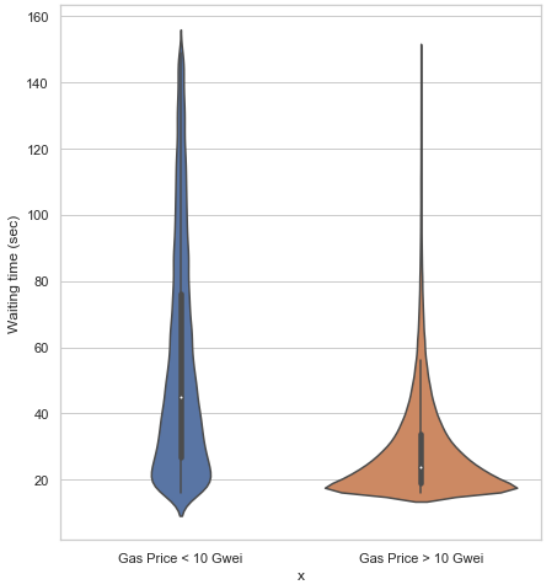



Figure 4.11: Violin Plot of the waiting time in seconds before a transaction is added to the Ethereum Blockchain. The blue plot to the left refers to the transactions having a Gas price lower than 10 GWei. The orange plot to the right refers to the transactions having a Gas price higher than or equal to 10 GWei.

Table 4.8: Statistical description of EthGasStation prediction on the Gas price recommendation (in GWei) for the transaction to be included at most in two blocks.

| **Mean** | **St.D** | **Min** | **25%** | **50%** | **75%** | **Max** |
|---|---|---|---|---|---|---|
| 15.28 | 6.60 | 3.0 | 10.0 | 20.0 | 20.0 | 61.0 |

and maximum (max) of the Gas price recommendation for the transaction for each Gas price category.

Table 4.9: Statistical description of EthGasStation Gas price categories (in GWei)
.

| | **Mean** | **St.D** | **Mode** | **Min** | **25%** | **50%** | **75%** | **Max** |
|---|---|---|---|---|---|---|---|---|
| Fastest | 15.33 | 6.60 | 20 | 3 | 10 | 20.0 | 20.0 | 61.0 |
| Fast | 4.58 | 2.42 | 3 | 3 | 3 | 3.6 | 5.0 | 60.0 |
| Average | 2.83 | 0.80 | 3 | 1 | 3 | 3.0 | 3.0 | 14.5 |
| Safe Low | 1.34 | 0.68 | 1 | 1 | 1 | 1.0 | 1.1 | 14.5 |

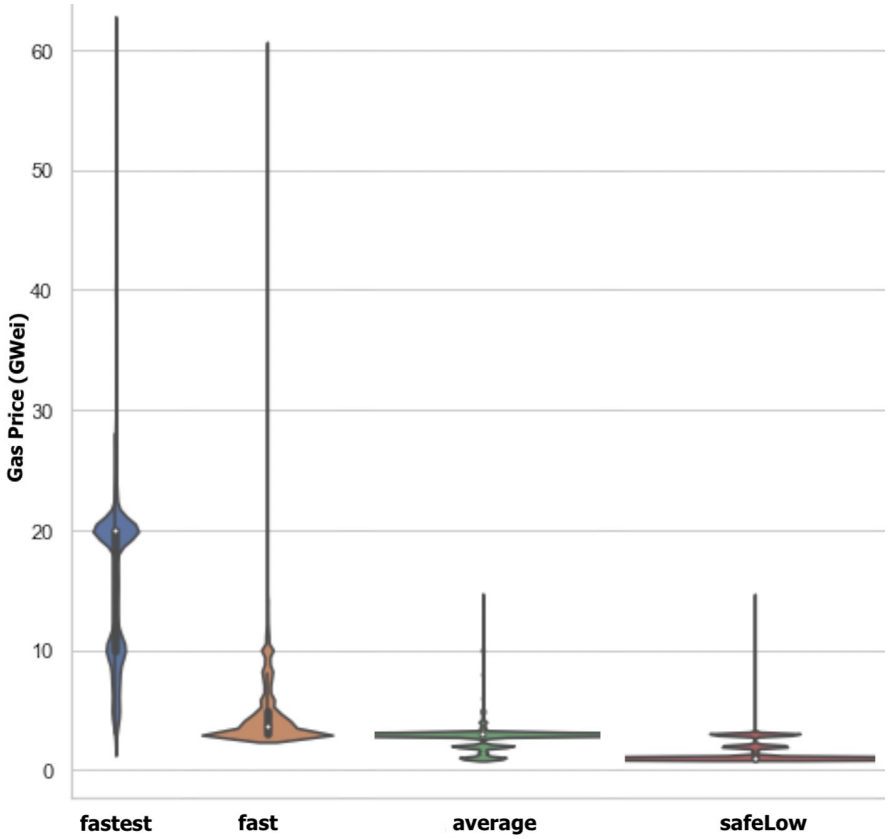



Figure 4.12: Violin plot of the EthGasStation Oracle's Gas price categories

The mode for the "fast" and "average" categories are the same. This indicates that most times the Gas price is the same for both categories, in spite of being different categories in terms of execution time. Therefore, our analysis suggests that these two categories should be merged.

It is not possible to be sure that the users who submit the transactions are following the Gas Oracles' recommendation. However, it is reasonable to assume that the users who set the Gas price equal to the Gas price suggested by the Oracle are indeed following the Oracle's recommendation. Even if they were not following the Oracle's recommendation, it is likely that the user agrees with the Gas price attached to the transaction and the waiting time. If the user disagrees with the waiting time, s/he would change the Gas price to rely on the expected waiting time.

The analysis of the Gas price of the transactions in the Transaction Pool shows that 16% of the transactions have a price equal to the price suggested by the Gas Oracle. The percentage of transactions having the Gas price equal to the Gas price suggested by the Oracle is distributed among the four categories as follows: 1) 7% safe low, 2) 1% fast and average, and 3) 8% fastest.

Figure 4.19b presents the percentage of Gas price categories used in Ethereum Blockchain. The data analysis of the transactions waiting in the Transaction Pool to be added to the Ethereum blockchain suggests that the categories "fast" and "average" are not followed by most users probably because these categories do not respond to their need. On the other hand, the categories "fastest" and "safe low" are much more used in practice.

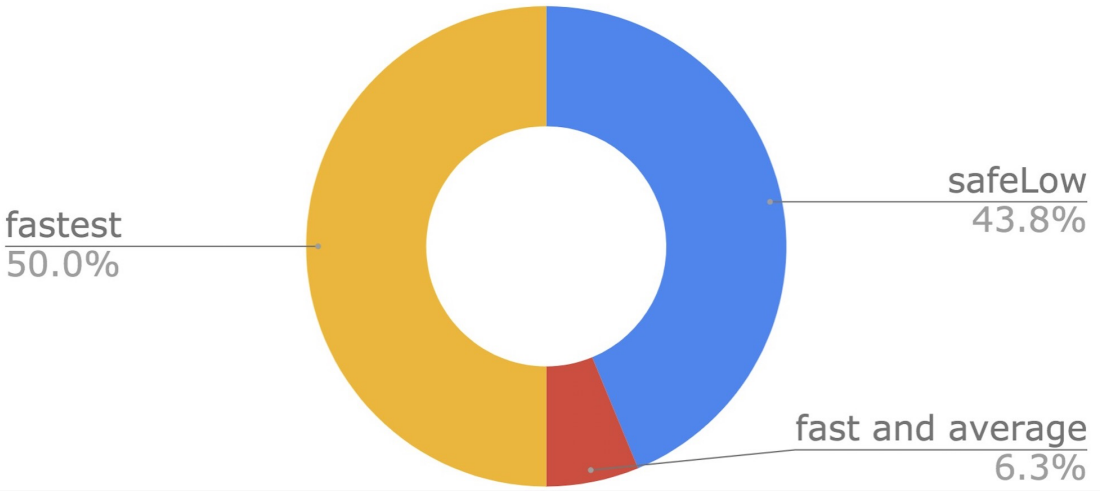



Figure 4.13: Usage of Gas Oracles Categories.

We used Equation 4.1 to calculate the margin of error of the EthGasStation predictions. EthGasStation claims to have a 2% margin of error. Table 4.10 shows the percentage of error among the Gas price categories recommended by the Oracle. As we can see in Table 4.10, the margin of error for every category is greater than

Table 4.10: Percentage of error among the Gas prices recommended by EtherGasStation according to Equation 4.1.

| | **Margin of error** | | | |
|---|---|---|---|---|
| | Fastest | Fast | Average | Safe Low |
| EthGasStation | 4% | 28% | 7% | 5% |

2%, and the "fast" category shows the greatest margin with 28% margin. Therefore, EthGasStation predictions may be less reliable than advertised.

**Discussion: How to Improve the Margin of Error?**

One of the reasons for the EthGasStation Oracle to have such a margin of error is because it performs its calculations (a Poisson Regression) to update the Gas predictions every 100 blocks (approximately 25 minutes). We argue that 100 blocks is not an appropriate interval of time for an Oracle to update its recommendations. Especially in the Ethereum blockchain, where the Gas prices can vary a lot within minutes.

In future related work, we aim to show that, by performing the Poisson Regression at more frequent intervals, it is possible to improve the accuracy of the prediction of the minimum Gas price to pay to have the transaction executed in a given time lapse.

**Threats to Validity**

**External Validity.** In this research, we analyzed data Gas predictions from the EthGasStation Oracle. Since Gas is a concept unique to the Ehtereum blockchain platform, this study cannot be generalized to other blockchain platforms. We did not address this threat for two reasons: (i) this research is an exploratory study on one specific Oracle; and (ii) analyzing other types of transaction fees for different blockchains is outside the scope of this research.

**Construct Validity.** We could not find how the oracle originally calculates its margin of error. Therefore, the oracle may indeed have the margin of error it claims under its method. However, since such method for error calculation is not openly available, we need a properly defined one for comparison. Therefore, we defined the equation 4.1 to measure the margin of error of the Oracle's predictions.

### 4.3.6 Related Work

Singh and Hafid (**?**) proposes a more fine-grained classification model that splits the confirmation time of transactions into eight classes: within 15 seconds, within

30 seconds, within 1 minute, within 2 minutes, within 5 minutes, within 10 minutes, within 15 minutes and within 30 minutes or longer. We know that on average, a transaction has to wait for two block confirmations (∼30 seconds) before being added. However, in the cases where the model would predict that the transaction belongs to the "within 5 minutes" class, there is no way for the user to know if it would take 3 minutes, rather than 4 minutes or more. Hence, while the paper presents a model with good prediction accuracy, it considers confirmation time prediction as a simple classification problem. It can only provide a user with an approximation of time it would take for their transaction to be confirmed, which may or may not always be ideal. In addition, Singh and Hafid (**?**) compare the performance of two machine learning regression models (Multi-Layer Perceptron and Random Forest) and the more classical, statistical model (Poisson Regression) on the task of predicting the confirmation time for a transaction in Ethereum Blockchain. The authors suggest that machine learning regression models perform well and better than the already used statistical approach. However, due to the need for the model to be periodically retrained and the time taken by the model to learn new data, the two machine learning regression models are not the most viable solution for the confirmation of the time prediction, as the users may need to know the Oracles' response in a much shorter time interval.

Pierro and Rocha (**?**) investigated the factors that influence the Ethereum transaction fees and the possible resulting decision-making behaviour of Ethereum Blockchain users, miners included. They observed that the past history of the Oracle Gas price prediction is useful to predict the number of waiting transactions, even though the converse is not true. The results of the Pearson correlation test showed that they are instead inversely correlated: when the Oracle price increases, the number of waiting transactions in the Ethereum network decreases. It stands to reason that when the oracle suggests a high price to pay, the users wait to submit a transaction, thus decreasing the overall number of pending transactions in their memory pools.

Chen et al. (**?**) identified seven gas costly patterns that are not optimized by the Solidity compiler. The authors analyze 4,240 contracts on three gas costly patterns. Their results show that over 80% of the contracts suffer from those costly patterns. The authors' work differs from ours because they focus on detecting possible waste of gas units, while in this study, we focus on the gas price (and the Oracles predictions for such price).

Ducasse et al. (**?**) proposed an open-source platform for blockchain analysis called SmartAnvil. Although SmartAnvil is intended to be independent of a specific blockchain platform, their work focus on Ethereum blockchain and contracts written in the Solidity language. For that reason, the authors have plans to include Gas optimization and estimation on SmartAnvil in the future, and they argue the importance of Gas estimation for contract development and analysis.

### 4.3.7 Conclusion

The present study evaluated the validity of the prediction the EthGasStation oracle makes on the Gas price to pay to have the transaction recorded in the blockchain.

We analyzed the oracle's predictions and have assessed that it carries a higher margin of error than originally claimed. EthGasStation claims to have a 2% margin of error, while our analysis shows that margin to be at least twice as much. For instance, the "Fastest" category showed a 4% margin of error, while the "Fast" category showed 28%. We argued the reason for that higher margin of error is because it cannot take into account changes in the Ethereum Network in real-time. We also argued that by updating the Gas price recommendations at every new block (instead of every 25 minutes), the margin of error can be lowered considerably.

The data analysis also indicates that the categories "average" and "fast" are not very used in practice, with less than 1% of the transactions set the Gas price suggested by the Oracle in those categories.

As future work, we plan to analyze data on other Gas Oracles besides the EthGasStation, to see if our findings also occur in other oracles. We are also going to implement our oracle to assess the feasibility and reliability to update the Gas recommendations more frequently.

## 4.4 A User-Oriented Model for Oracles' Gas Price Prediction

### 4.4.1 Introduction

Ethereum blockchain is a distributed ledger where transactions are recorded into a sequence of ordered blocks. The Gas is a unit of measurement unique to the Ethereum blockchain that measures the computational work required to run transactions within the Ethereum Virtual Machine (EVM). The transactors, i.e. the users or the smart contracts that submit transactions to the blockchain network (henceforth "users"), also propose a fee in terms of Gas price to validate, include, and compute the transactions effect, when an executable code, the so called "smart contract", is called (**?**). The users especially pay the fee in Ethers, the Ethereum cryptocurrency, for the effort required to compute the proof-of-work (PoW). The PoW keeps the network resilient, though requiring a big investment of the miner, i.e. the node that solves the PoW challenge [12]. The PoW challenge indeed consists in a cryptographic puzzle requiring large computational resources (**?**). As there is a (weighted) distribution of minimum acceptable Gas prices, the users will have a trade-off to decide between lowering the Gas price and maximising the chance that their transaction will be timely committed to the blockchain (**?**).

[12]Ethereum is currently migrating to a Proof of Stake consensus algorithm. However, the study targets current transactions on the main network where the PoW is still in use.

To send a transaction on the Ethereum blockchain, the user needs to specify a Gas limit, which is the maximum amount of Gas that can be consumed by the transaction, and a *Gas price* which is the cost in Ether the user is willing to pay per unit of Gas consumed. If the transaction spends less Gas than the Gas limit, the remaining Gas will be refunded to the user and the miner will earn less than the maximum Gas Limit. Indeed, unlike the Bitcoin blockchain, where the users do not need to set a Gas limit, but just the transaction fee which will be paid to miners, in the Ethereum blockchain there is always the possibility for the miners to receive a minor reward compared to the Gas limit set by the users (**?**).

There are some main reasons to have a *Gas price* in the Ethereum blockchain: 1) the users must pay for computational costs and resources used (*e.g.,* energy, CPU) to generate and include their transactions into blockchain blocks upon approval; 2) a Gas price regulates and limits the use of blockchain resources; 3) a Gas price incentivizes miners to actually include transactions in the blocks without just mining empty blocks; 4) a Gas price allows the users to express (and pay for) priority; 5) a Gas limit avoids network abuse or misuses, intentional or unintentional (*e.g.,* DoS attacks, infinite loops) (**?**).

The users, sometimes via an intermediary, send a transaction to an Ethereum node. From there, the transaction is broadcast to other nodes and distributed across the network. When the transaction reaches a miner's node, the miner can add it to the pool of pending transactions (also called "memory pool") and then include it in a new block which may be appended to the last one in the chain (**?**).

The computational cost of a transaction in Gas units depends only on the computations occurred to process such transaction. The Ethereum documentation provides the different costs of each elementary operation. The users are free to specify any *Gas price* that they wish, however the miners are free to ignore transactions as they choose. Some miners, especially the miners with high computational resources, may seek to make the highest profit and change the source code to evaluate the transactions based on the Gas parameters, i.e. the Gas limit and the Gas price (**?**). For instance, "Go Ethereum", a software installed in some nodes of the Ethereum network, might be used to set the parameter expressed by the variables "–txpool.pricelimit" and "–txpool.lifetime". In particular, the "txpool.pricelimit" variable defines a baseline transaction price under which the node will simply not accept transactions (not even to forward it to other nodes) (**?**). Consequently, on the one hand, if the value set by the user is too low, miners will probably ignore such transactions which risk to be never included in the blockchain. On the other hand, if the transaction fee is too high, miners will be prone to include it in the Ethereum blockchain, but the user will allegedly waste money. To suggest the best trade off for Gas price, the Gas Oracles assign the *Gas price* to categories, which are actually based on four quantiles (50th, 75th, 95th, 99th) determined from past Gas price observations. Section 4.4.2 explains how the Gas Oracles model the Gas price, based on data from past blocks (**?**).

In this study, we extend preliminary results (**?**) obtained for a single Oracle case,

the *EthGasStation Oracle*, to another case, the *Ethchain Oracle*, in a wider timeframe, by analyzing the data of the Oracles that predict the Gas price, along with the Ethereum transactions' and blocks' data. The Ethereum transactions' variables considered in the study are:

- the waiting time calculated as the time elapsing between the time the transaction was seen by the miner we are considering in this research and the time the transaction has been included into the block (**?**).
- the Gas price, *i.e.,* the amount of Ethers the user is willing to pay for every unit of Gas, which is measured in "GWei" (**?**).

Oracles' data are useful to predict the Gas price a user should pay to make it convenient for a miner to include the transaction into a block. To help the users in deciding the price to pay for the cost of the PoW calculation, Gas Oracles propose the following four price categories: *'safeLow'*, *'average'*, *'fast'*, and *'fastest'*. These categories define the Gas price required to have a transaction included within the next 100, 20, 5, and 2 blocks, respectively. The study aims to answer the following research questions:

- RQ#1: Are the Oracles' predictions reliable as much as declared?
- RQ#2: Do the Gas price categories provided by the Oracles correspond to the *Gas price* categories the users actually set?
- RQ#3: How could the Oracles provide the users with more reliable predictions?

To answer our research questions, we hypothesized that 1) the predictions made by the Gas Oracles have a margin of error greater than the margin of error declared by them (2%); 2) the categorizations of the Gas price made by two Oracles do not correspond the Gas price the users and/or companies actually set; 3) it is possible to reduce the Gas Oracles' error margin by calculating the *'recommended Gas price'* when each block is added instead of every 100 added blocks as the existing Gas Oracles actually do (**?**).

We collected data in three-months time from two Gas Oracles (Etherchain and EtherGasStation) which predict the *Gas price* every time that 100 blocks are added to the Ethereum blockchain. During the same period, we also collected over 10 million transactions from a transaction pool. We then cross-checked the data collected by the transaction pool and the Oracles, to understand whether the Oracles' estimates fail.

First, the results of this research show that both Gas Oracles (Etherchain and EtherGasStation) give the *Gas price* prediction with a higher margin of error compared to what they declare (2%). The margin of error ranges from a minimum of 4% for the *'fastest'* category to a maximum of 28% for the *'fast'* category. Second, the

results show that the margin of error could be lowered to 2% for all the categories, by performing the Poisson regression at smaller intervals of time. Finally, the results suggest that two of the Gas Oracles categories are not frequently used in practice: *'fast'* and *'average'* categories. It is indeed reasonable to expect that single users or companies aim to save money and thus set some requirements, which are different in terms of waiting time and are not provided by the default categories.

The rest of the study is organized as follows. Section 4.4.2 presents the concepts needed to better understand our research, such as the transaction pool, the Gas Oracles, and the Gas price categories investigated in the study. Section 4.4.3 presents the related work the study uses as a starting point for a user-oriented model for the Gas Oracles' Gas price prediction. Section 4.4.4 presents the experimental hypotheses guiding the study. Section 4.4.4 describes the methodology used to test the hypotheses, to collect and perform the regression model on the data of the study. Section 4.5.4 presents the results of the study. Section 4.4.8 discusses the results in the light of the user-oriented model. Finally, Section 4.4.9 draws some conclusions and outlines some ideas for future work.

### 4.4.2 Background

This section provides the readers with a brief introduction on the blockchain technology and in particular on the Gas price mechanism sets on the Ethereum blockchain to ensure a balanced use of resources.

#### Blocks

The blockchain is an ordered sequence of blocks containing the records of valid transactions as approved by a consensus algorithms shared between a set of computational nodes in a peer-to-peer network. It is a shared ledger where, to keep unchangeable the block sequence and the temporal order of recorder transactions, each block includes a cryptographic hash depending on the information recorded on the previous block. Each block is also identified by progressive number named "height" (**?**). Once a block is created and added to the blockchain, the transactions in the block cannot be changed or deleted. This is to ensure the integrity of the transactions and to prevent the double-spending problem (**?**).

Block time is the time the network takes to generate one extra block In Ethereum the *'average'* block-time is expected to be between 10 to 19 seconds (with an average of 13 seconds) and depends on how long the miners take to find the correct hash to validate a block by brute force computation. In the blockchain there are two units of time measurements: (i) seconds, and (ii) blocks' number or height (**?**).

### Transaction Pool

The software running in each miner node collects the transaction into a virtual storage named "transaction pool". The miners distinguish processable transactions, which can be included into a block, from future transactions, which can wait to be included. Therefore the transactions move between these two states over time as they are received and processed (**?**). When a miner solves the PoW challenge to mine a block, the miner informs the adjacent nodes about that. As the adjacent nodes receive this piece of information about the newfound block, they will validate the received block and propagate the block data to peer nodes. In the case of mining nodes, they will remove all the transactions contained in the newfound block from their own transaction pool, checking also for transactions attempting to double-spend the same output. A double spend is a potential flaw in a digital cash scheme, where the same single digital token can be spent more than once (**?**). The miners have full control over their transaction pool and may adopt different policies to manage it. For instance, a miner could set up a minimum fee threshold, thus transactions with a *Gas price* lower than the threshold are immediately discarded from the transaction pool and only the new transactions with a price higher than the threshold are allowed to enter the transaction pool (**?**).

### Gas Oracle

In the blockchain terminology, Oracle may have different meanings. An Oracle can be a program which provides the smart contracts with reliable data collected from outside the blockchain. Oracles are also software systems which analyse some data and make some prediction on that basis (**?**).

In this study, the term Gas Oracle assumes a specific meaning related to the activity of forecasting Gas prices. The Ethereum wiki [13] reports the following definition: "a Gas Oracle is a helper function of the Geth client that tries to find an appropriate default *Gas price* when sending transactions and it can be parametrized". Thus an Gas Oracle analyses blockchain data to predict the best *Gas price* to pay for a transaction to be approved within a certain number of blocks. The Oracle's forecasts may be important for companies using the Ethereum blockchain because the time and the costs of performing transactions can affect their economical resources and clients' satisfaction (**?**). It is thus crucial for them that Oracles forecasts are as reliable as possible. However, based on the analysis performed in this study, it is not the case.

We indeed analyzed the predictions of two Gas Oracles: EtherGasStation and Etherchain. Both Gas Oracles claim that all predicted values are estimations based on the current network conditions and should be used as a suggestion. However, the Gas Oracles only compute and update their predictions every 100 blocks (approximately 1,500 seconds or 25 minutes) (**?**). Therefore, their estimations might not

[13] https://eth.wiki/

Table 4.11: *Gas price* categories with the relative waiting time

| Gas price category | Maximum waiting time to include the transaction into a block |
|---|---|
| ***'fastest'*** | 1-2 blocks |
| ***'fast'*** | at most in 2 minutes |
| ***'average'*** | at most in 5 minutes |
| ***'safeLow'*** | at most in 30 minutes |

mirror the current status of the network.

### Gas Price Categories

Gas Oracles, EtherGasStation and Etherchain, estimate the time interval required for a transaction to be included into the next blocks based on the Gas price attached to the transaction (**?**). To estimate the waiting-time a transaction needs to be included into a block, many variables need to be considered, such as the number of transactions submitted by the users in a given period of time, the number of miners and their policy (**?**).

Gas Oracles have defined four categories based on the quantiles of the *Gas price* offered to the miners by the users which are accessible from the transactions data. The four percentile are the 50th, the 75th, the 95th and the 99th percentile (**?**). The 50th percentile corresponds to the *'safeLow'* category, the 75th percentile to the *'average'* category, the 95th percentile corresponds to the *'fast'* category and finally the 99th percentile to the *'fastest'* category.

To make information more accessible to users, the Gas Oracles states that each category corresponds to a waiting time. In reality, it would be more effective for the users to know that the *Gas price* is related to the number of blocks to wait and not to the time because the average value to mine a block is 13 seconds but there can be strong oscillations ranging from a few seconds to over half an hour to mine a single block (see Section 4.5.4). Table 4.11 presents the categories defined by the Gas Oracles (Etherchain [14] and Etherscan [15]) and their waiting-times. The code which estimates the *Gas price* to pay to the Gas Oracles is publicly available under doi: 10.5281/zenodo.3758103.

[14]https://etherchain.org/tools/gasPriceOracle
[15]https://docs.ethgasstation.info/gas-price

### 4.4.3 Related Work

The blockchain can be disadvantageous from a user's perspective because of different kind of wasted resources. The study focuses on the waste of Gas price or waste of time the users might experience to add a transaction to a block. Previous work highlighted other kinds of waste from a user-oriented perspective. Chen et al. (**?**) identified seven Gas costly patterns, *i.e.,* programming solutions that are not optimized by the Solidity compiler. A Gas costly pattern required more computational resources, thus reducing the number of transactions that can be included into a block. Therefore the users need to wait or pay more to have their transaction executed. The authors analyzed 4,240 smart contracts on three Gas costly patterns. They found that over 80% of the contracts suffer from this kind of costly patterns. The authors' work is therefore interesting because it highlights an existing waste of Gas units in the blockchain, which disadvantages the users' interests.

In a previous study, the researchers (**?**) measured the time for transactions to be committed in both Ethereum and Bitcoin blockchain. The authors performed a detailed analysis of issues that could negatively impact commit times in permissionless PoW blockchains such as Ethereum. Their study is very interesting for the purpose of this reseatch because it identifies the *Gas price* as a cause of delay in the commitment of transactions.

A research study (**?**) investigated the reliability of seven Oracles on different platforms such as Augur, Ms Bletchley, TownCrier and Corda. They discovered that the common causes of failure are the data sources used by the Oracle to make various kinds of predictions such as the weather forecast. These failures can have a serious impact on the economy because many smart contracts perform operations on the basis of these predictions. To meet the users needs, they provided a framework that can be used to assess the Oracles' reliability. The authors' work supports the interesting idea that, providing this framework together with the Gas Oracle, it is possible to help the users' decision making. Differently from their work, in this research, we study the behavior of the Oracles that predict the Gas price in the Ethereum blockchain.

Ducasse et al. (**?**) pointed out that even more experienced users, as software developers of smart contracts, need to be helped to write smart contracts that are more effective by using fewer resources. This is the main reason why the authors proposed an open-source platform for blockchain analysis called SmartAnvil. Although SmartAnvil is independent from a specific blockchain platform and thus may be used to investigate any blockchain, their work focused on Ethereum blockchain and contracts written in Solidity. The authors provided a tool that can facilitate the identification of resources waste to solve a problem within the smart contracts. The authors' work is therefore interesting because it supports the idea that, providing this tool together with a Gas Oracle, it is possible to help users and companies to waste less time and money.

In a previous study (**?**), Singh and Hafid proposed a more fine-grained classifi-

cation model when compared to the existing Gas Oracles' classification. The model split the confirmation time of transactions into eight classes: respectively within 15 seconds, 30 seconds, 1 minute, 2 minutes, 5 minutes, 10 minutes, 15 minutes, and 30 minutes or longer. Interestingly, the authors proposed a classification that considers different possibilities for the users to set a Gas price that might meet their needs, while existing classification do not pay attention to the users' point of view. In this study, I limit my research to the existing classification of the Gas Oracles, but I accept the idea that there may be some categories that better represent the users' needs and interests.

Generally, the users or the companies may have the following interests and needs: 1) they may sometimes be willing to pay a lot to have the transaction executed as soon as possible, 2) they may sometimes be willing to save money and wait a lot, as long as their transactions are eventually added to the blockchain. For instance, the users or companies may be willing to pay a lot during an initial coin offer (ICO), when only a limited supply of tokens is available and thus "the first to arrive is the first to be served". On the contrary, when the time is not constrained, they may wait to save money. For instance, when the smart contracts need to refund users having an assurance in case of delay of arrival, the users might want to wait a few hours before receiving the reimbursement (**?**).

Singh and Hafid (**?**) also compared the performance of two machine learning regression models (Multi-Layer Perceptron and Random Forest) and the more classical, statistical model (Poisson Regression) on the task of predicting the confirmation time for a transaction in the Ethereum blockchain. Interestingly, the authors suggested that machine learning regression models perform well and better over the existing method performed by the Gas Oracles, than the already used statistical approach. However, the blockchain networks can change for many reasons (**?**). Previous research shows that the number of transactions moving through the Ethereum network can increase or decrease based on specific users-related events, as for instance when a company looks to raise money to create a new coin, app, or service. In such real scenarios, after a certain number of blocks we need to retrain a model like the one proposed by Singh and Hafid. In conclusion, the two machine learning regression models are not the most viable solution for the confirmation of the time prediction, as the users may need to know the Oracles' response in a much shorter time interval.

Another study (**?**) investigated other factors that might influence the Ethereum transaction fees and the possible resulting decision-making behaviour of Ethereum blockchain users, miners included. They observed that the past history of the Oracle Gas price prediction is useful to predict the number of waiting transactions, even though the converse is not true. The results of the Pearson correlation test showed that they are instead inversely correlated: when the Oracle price increases, the number of waiting transactions in the Ethereum network decreases. It stands to reason that when the Oracle suggests a high price to pay, the users that can wait, wait to submit a transaction, thus decreasing the overall number of pending transactions in

their memory pools. This result pushes us to target our research towards a model oriented on the users and not on the mere data or miners.

In a previous study (**?**), a quantitative study was conducted to determine whether the *Gas price* prediction of the Oracle EtherGasStation is reliable. The study aimed to evaluate the correctness of the Gas price prediction the EtherGasStation made to have the transaction recorded in the blockchain. The study investigated the EtherGasStation's predictions and found that it brings about a higher margin of error than originally declared. EtherGasStation indeed claims to have a 2% margin of error, while the analysis of the predictions showed that the margin is at least twice as much. For instance, the *'fastest'* category showed a 4% margin of error, while the *'fast'* category showed a 28% margin of error.

Moreover, the study argued that such a higher margin of error is due to the fact that EtherGasStation does not take into account changes in the Ethereum Network occurring in real-time. The study was anyway limited in various ways, as it considered just one Gas price Oracle in a short time framework, so that the results cannot be generalized and used to understand wider and general trends in Gas Oracles' prediction.

This study therefore provides a more comprehensive analysis of Gas price predictions, by performing: *a*) a quantitative analysis of the predictions of another Gas Oracle, Etherchain, to check whether they are reliable or whether also in this case they fail in suggesting the right Gas price as in the case of the EtherGasStation Oracle; *b*) a test for the hypothesis that the Gas price's margin error is reduced for each category, when reducing the time interval required for the estimation of the Gas price.

### 4.4.4 Research Methodology

The research methodology of the study includes the following phases: (a) the experimental hypotheses, (b) the Data Collection, (c) the Data Cleaning, (d) the Data Modelling, and (e) the Regression Analysis. The following sub-sections describe each phase.

#### Experimental Hypotheses

The study was designed to address the following Research Questions:

- RQ#1: Are the Oracles' predictions reliable as much as declared?
- RQ#2: Do the Gas price categories provided by the Oracles correspond to the Gas price categories the users actually set?
- RQ#3: How could the Oracles provide the users with more reliable predictions?

To answer the questions, we advanced the following hypotheses:

- H1: The Gas Oracles' predictions are not reliable. The Gas Oracles cannot indeed take into account all the changes in the Ethereum Network in real-time, especially because they compute the prediction every 30 minutes on average.
- H2: The Gas price categories proposed by the Oracles do not match the categories actually set by the users and/or companies. Single users or companies may indeed set different requirements in terms of waiting time that is not provided by the default categories.
- H3: A reduction of the margin of error in the Gas price prediction can be achieved by calculating the *'recommended Gas price'* at smaller interval of time, thus considering the current changes of the network in real time.

### Data Collection

In this research, we covered a 3-month analysis period and we made code publicly available to replicate the data collections of the transactions used in this research. The source code is available at the following online address `https://github.com/aphd/eset/tree/master/src`. The same code can be used to analyze the transactions' data in other time frames. We collected data by making requests to various REST API services at different times. The flow to collect data is:

- a request is sent to the server every 15 seconds;
- if the request is successful, the server responds to the client request sending a payload in JSON format;
- if the request is not successful, the client does not record any data for that time frame. During the data retrieving operation, an average of 1 request out of 20,160 requests was unsuccessful, *i.e.,* only the 0.0049% of requests have failed.

Table 4.12 shows the URI of the REST API services used to fetch the Gas Oracle data, the blocks data, and the unconfirmed transactions data, *i.e.,* the latest transactions that have not been included in any block. We choose to collect data from these Gas Oracles because they are very popular among the Ethereum community. Data were stored as files in JSON format in the file system of the server where the analyses were performed.

### Data Cleaning

A control over the data quality was performed. The data retrieved were accepted when in compliance to the API documentation, or otherwise rejected. The 0.8% of data was rejected, distributed as follows: 770K out of 11M transactions (0.75%),

Table 4.12: RESTful Services list

| Resource name | REST API service URI |
|---|---|
| **EtherGasStation** | https://ethgasstation.info/json/ethgasAPI.json |
| **Etherchain** | https://www.etherchain.org/api/gasPriceOracle |
| **Block** | https://api.blockcypher.com/v1/eth/main/blocks/0 |
| **Unconfirmed Transactions** | https://api.blockcypher.com/v1/eth/main/txs |

182 out of 345K blocks (0.05%) and 112 out of 345K Oracle's predictions of the Gas price (0.03%). Example of data, which were not in compliance to the API documentation, are:

- string value where a numeric value was instead expected;
- numeric value where a string value was instead expected;
- numeric value which is not in the expected range;
- date value which is not in the expected time frame;
- missing value;
- missing key/value pairs;
- numeric value with different units of measurement.

The data falling in one of the categories listed above were rejected, except for the latest category where the values were recalculated conforming with the expected measurement units. Just to give an example, a numeric value which is not in the expected range can be a negative block's height value, a date value in the future, or a negative value of the waiting time for transactions to be included into a block.

According to Kanda and Shudo (**?**), a negative value of the waiting time variable may suggest a transaction propagation delay among different nodes. This means that different nodes in the blockchain could see the transaction at different instants of times. The transactions data-set contains indeed transactions with a negative waiting time, around the 1.16% of all transactions. This might have at least two reasons: 1) every node can have a different clock time, and 2) there is a propagation delay defined as the difference between the time when a node announced the discovery of a new block or a transaction and the time when this announcement was received by other nodes (**?**). In the study, the transactions data were collected through an API that gives the transaction pool data of a single node. The API is

available at the following online address `https://www.blockcypher.com/dev/`. As the Ethereum network is distributed, not all miners receive the same transactions at the same time, therefore some nodes might store more transactions than others at some time (**?**). Furthermore, every node can be a miner with different hardware and software features and miners might have different RAM capacity to store unconfirmed transactions. As a result, each miner has its own representation of the pending transactions. The existence of such delay - which is not negligible - justifies the negative times, because blocks can be discovered while communication and validation is still in process. Decker and Wattenhofer (**?**), for the BitCoin blockchain, observed that the median time until a node receives a block was 6.5 seconds, the mean was 12.6 seconds and the 95th percentile of the distribution was around 40 seconds. Moreover, they showed that an exponential distribution provides a reasonable fit to the propagation delay distribution. It is reasonable to think that there is a similar effect in the Ethereum blockchain.

### Modelling Data

In this subsection, we define the condition to assess the correctness of the Oracles' *Gas price* prediction. Both EtherGasStation and Etherchain make the prediction based on the history of the mined blocks data, such as the lowest *Gas price* accepted by the miner to add the transaction to the block and the Gas offered by the users.

Suppose that during the time interval when the i-th block, $B_i$, is mined:

- *op* (Oracle Price) is the price predicted by the Gas Oracle to have the transaction included at most within j blocks;
- $B = \{B_{i+1}, \ldots, B_{i+j}\}$ is the set of j blocks mined in the blockchain following the i-th block;
- $T = \{tx_1, tx_2, \ldots, tx_n\}$ is the set of unconfirmed transactions, *i.e.,* that have not been included in any blocks, with *Gas price* respectively of $tp_1, tp_2, \ldots, tp_n$, that there were in a transaction pool when the i-th block was mined.

In an ideal scenario where the Gas Oracles never fail, the transactors that set the Gas price equal or greater to the one suggested by the Gas oracle (*op*) should have the transaction committed in the blockchain after n blocks where n depends on the category proposed by the Gas Oracle and chosen by the user. Figure 4.14 shows the ideal scenario where the users that set the Gas price following the Gas Oracle's suggestions have their transactions committed in the blockchain in the following n blocks. The condition is expressed by the following equation:

$$\forall tx_i \in T \wedge tx_i \geq op : tx_i \in \{B_{i+1}, ..., B_{i+j}\} \tag{4.2}$$

The equation 4.2 is used to verify the prediction of the Oracles' *Gas price*: Etherchain and EtherGasStation.

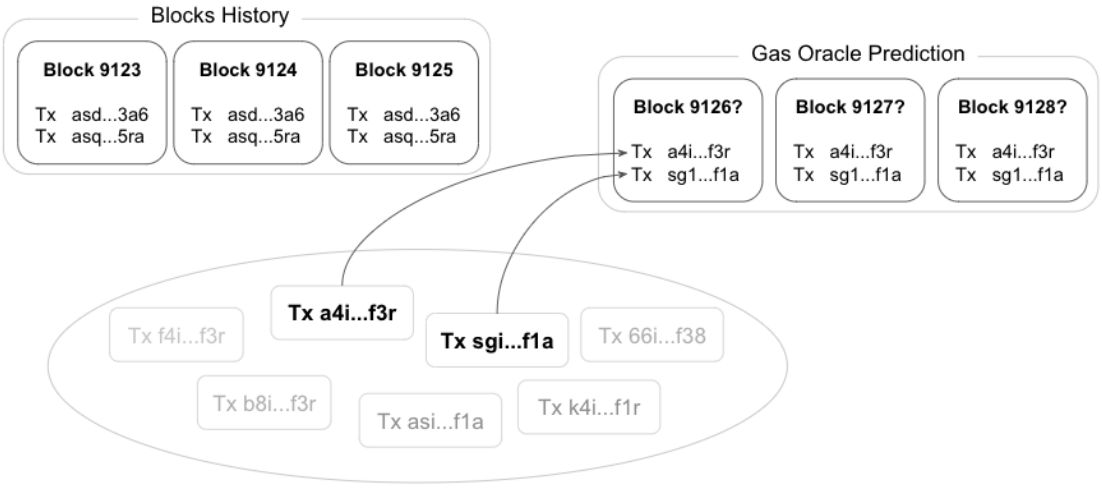



Figure 4.14: The transactions having a Gas price higher or equal to the one proposed by the Gas Oracle (*op*) are displayed in bold.

The existing Gas Oracles make the prediction every 100 blocks, performing a Poisson regression. Based on the *Gas price* distribution of the transactions mined in the last *200* blocks, the Gas Oracles estimate the *Gas price* to pay in relation to the number of blocks the users should wait to have their transaction added. The data are collected by querying the REST API Services mentioned in Table 4.12. To test the hypothesis that the prediction can be improved, we performed the same algorithm used by the Gas Oracles (**?**) at shorter time intervals (every 4 block for the *'fastest'*) instead of every 100 blocks as the Gas Oracles actually do. The results are collected in a table called realTimeOracle 4.15.

In this phase the data were collected and stored in a relational database, where each table represents the following items: blocks, transactions, Oracles, and OtherPrediction. Figure 4.15 shows the data contained in the database, the relationships between table fields and their types (*e.g.,* string, integer, boolean, enumerate). The table named *transaction* stores all the transaction information such as the received time detected in the transaction pool that was monitored for this research. It is noteworthy that the waiting time for a transaction is not stored in the database but calculated by the difference between confirmed (time) and received (time). The table named *block* stores all the block information such as the current block number in the blockchain (block_height), the number of transactions stored in a block (n_tx) and the lowest *Gas price* among all the transactions added in that block (lowest_gas_price). The tables *Etherchain* and *EtherGasStation* store the *'recommended Gas price'* to have the transaction included in the block for each category. Finally, table *realTimeOracle* stores all *'recommended Gas price'* to have the transaction included in the block for the *'fastest'* and *'safeLow'* category. The other two categories considered by the Gas Oracles have been excluded since this work, as well as

previous works (**?**), shows how the *'fast'* and *'average'* categories do not reflect the requirements of companies and users.

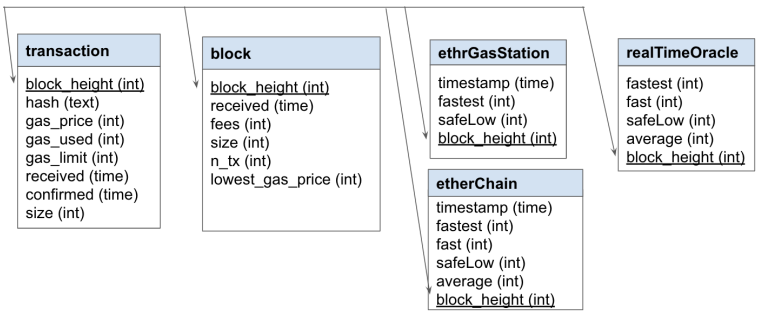



Figure 4.15: Database schema.

### Poisson Regression Model

The Oracles adopt a Poisson regression model, as per source code available via Zenodo (**?**). As anticipated in section 4.4.3, different models, such as the machine learning regression models, have been applied to improve the Gas Oracles prediction. Although some of these models can give better results compared with the Poisson Regression model, they have the drawback to be very expensive in terms of computing resources. The time required to make the prediction (around 20 minutes) is too long, as it is greater than the time taken by the blockchain network to mine a block. This is the main reason why we investigated how to improve the Poisson Regression model already used by the Gas Oracles.

In probability theory and statistics, the Poisson distribution is a discrete probability distribution of a given number of events occurring in a fixed interval of time or space (**?**). We use the Poisson distribution to estimate the number of transactions added to the blockchain per block period, which is on average 13 seconds. Let X represents the set of x transactions added to the blockchain in a one block period. For the sake of simplicity, let assume that all blockchain transactions are offering the same Gas price to the miners. Equation 4.3 shows the Probability Mass Function (PMF) of having x transactions added to the blockchain in one block period time.

$$P(X = x) = \frac{\lambda^x * e^{-\lambda}}{x!}, x \in [0, \infty) \tag{4.3}$$

In the Equation 4.3, $\lambda$ is the mean number of transactions added to the blockchain in one block period of time and $e$ is Euler's number.

To know the probability of having x transactions added in the n-th block, in the formula 4.3 the $\lambda$ value is to be multiplied for the number of blocks. Equation 4.4 shows the probability of having x transaction added to the n-th block.

$$P(X = x) = \frac{n * \lambda^x * e^{-(n*\lambda)}}{x!}, x \in [0, \infty) \tag{4.4}$$

The equations 4.3, 4.4 are valid when the events are observed under certain conditions.

Conditions (C) for Poisson Distribution are:

- C1: An event can occur any number of times during a time period. In our case, the event is the transaction added to the blockchain and the time period is the block period.
- C2: Events occur independently from each others. In our case, if a transaction is included into a block, it should not affect the probability of another transaction to be included in the same block, i.e. in the same interval of time.
- C3: The average rate of events occurrences, i.e. the number of transactions added to the blockchain per block, should be constant, i.e. the rate should not change based on the block number added to the blockchain.

Of course, the Poisson distribution conditions are highly theoretical and do not fit the blockchain real situations (RS) for many reasons:

- RS1: As to the condition C1, our results 4.14 show that the number of transactions added to a block could be any integer number greater or equal to zero. Thus, the condition C1 is satisfied.
- RS2: As to the condition C2, a transaction included into a block does affect the probability to have another transaction added to the same block, simply because the number of transactions is finite. However, in most cases, the number of transactions is so high that they can be considered independent with good approximation.
- RS3: As to the condition C3, there are cases where it may not be satisfied. For example, if the policy of the miners suddenly changes and they decide to mine empty blocks, the average rate of occurrence will drastically change. Although the data in table 4.14 confirm that this can happen from time to time, this is not the normal situation, because it goes against the interests of the miners themselves. The network would indeed lose its usefulness and the value of the Ethereum cryptocurrency (Ether) would decrease when compared to other currencies (USD, EUR, etc.), and as a consequence, also the reward of the miners, who are paid in Ether. Moreover, a changing number of transactions submitted by the users to the blockchain network could change the average transactions number per block, based on the Gas offered to the miners. This can happen, for instance, during an ICO.

The points discussed above might explain why Gas Oracles' margin error is larger than expected. We suppose that, by recalculating the *lambda* factor in equation 4.4 at lower time block intervals compared to the Gas Oracles which recalculate the *lambda* factor every 100 blocks, the margin of error in the probability computation of having a transaction added to the blockchain in a certain number of blocks, can be lowered. This does not mean that the Poisson model, in which the lambda is recalculated every time a block is added to the blockchain, is the best way to model the blockchain. Of course, other models might be tested taking into account the time limit of 15 seconds, but up to now our model gives better results compared to the current Gas Oracles' model, as will be shown in section 4.4.7.

**Regression Analysis**

The purpose of this phase is to estimate the *Gas price* by running a Poisson regression analysis on the data stored in the block table as Gas Oracles currently do. The results of the Poisson Regression fill the table named "realTimeOracle". Unlike Gas Oracles, we perform the Poisson regression more frequently based on the four category *'fastest'*, *'fast'*, *'average'* and *'safeLow'*. This choice is justified by:

- the categories are very different from each other based on waiting time requirements, expressed by different constraints. For example, the category *'fastest'* has the constraint of having to guarantee 98% of transactions to be included into a maximum of two blocks. This means that the error on the lambda determination must be very small compared to the block interval. The category *'safeLow'* instead requires to have the transactions included into a larger time frame (120 blocks) and so the error on the $\lambda$ can be greater compared to the category *'fastest'*.

- the time interval of 100 blocks to recalculate the $\lambda$ could be very long compared to changes in the network. According to our observed data 100 blocks correspond to about 30 minutes with a sigma equal to 20 minutes. We suppose that during this interval of time the network condition can change and this change can affect the value of $\lambda$.

### 4.4.5 Results

This section presents the data-sets (blocks data-set, transactions data-set, Oracles' predictions data-set, user-oriented predictions data-set), as modeled in Section 4.4.4. The data-sets are stored in an SQLite database with five tables, one table for each data-set. The total size of the database is of 1.1 Giga-Byte and is publicly available via Zenodo (**?**). The first table, named "transaction", contains more than 11 millions rows. The second table, named "block", contains around 345 thousand blocks. The blocks data-set consists of 103596 records starting from height 7590409 to height

Table 4.13: Statistical description of transactions data

| | mean | std | mode | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| waiting_time (s) | 44.02 | 82.65 | 25 | 0 | 25 | 29 | 38 | 1499 |
| gas_price (GWei) | 32.19 | 443.29 | 50 | 0 | 10 | 20 | 50 | 313734 |
| gas_used | 70124.2 | 320908 | 21K | 0 | 21K | 21969 | 49993 | 8e+06 |
| gas_limit | 303967 | 947926 | 21K | 21K | 42K | 70000 | 150000 | 8e+06 |
| size (Byte) | 191.11 | 499.98 | - | 83 | 112 | 114 | 174 | 31791 |

.

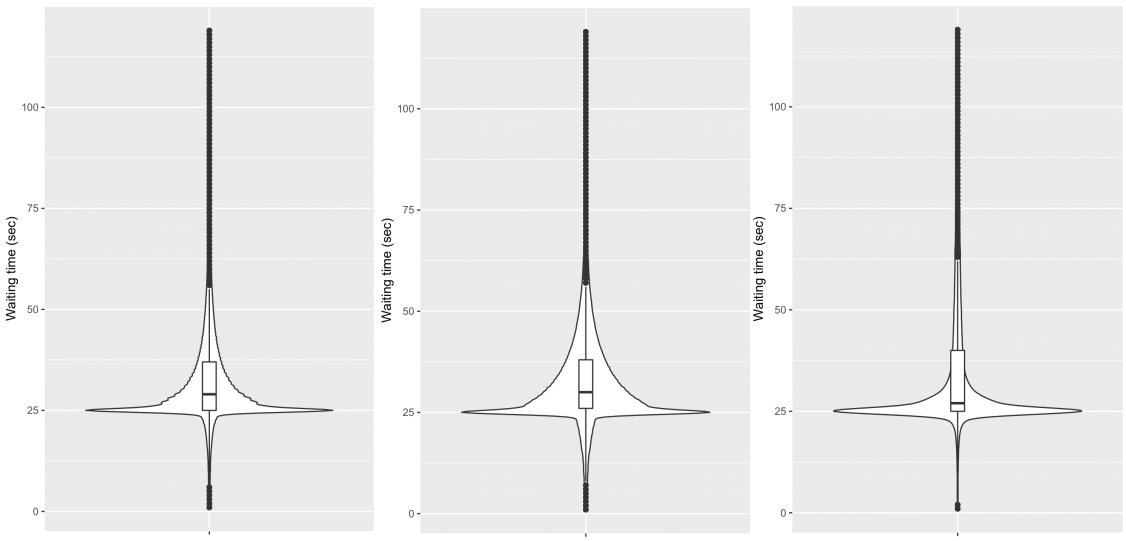
7694005. At the date of the research the last block is 7764216, meaning that we analyzed the $103596/7764216 * 100 = 1\%$ of the Ethereum blockchain. The two tables, named respectively "EtherGasStation" and "Etherchain", contains 345 thousand rows of Oracles' predictions for the Gas price of each category: (*'fast'*, *'fastest'*, *'average'*, and *'safeLow'*). The Oracles' predictions data-set covers a period of three months starting from 15 March 2020 with 15 seconds temporal resolution. Finally the table "realTimeOracle" contains the data of the user-orinted model for Oracles' *Gas price* predictions. The data-sets refer to a three-months period of time, ranging from March 1, 2020 to May 28, 2020.

The following sections present the results of the study as some aggregated statistical metrics such as percentile, mean, standard deviation, mode of the numerical data series for transactions, blocks, Gas Oracles' predictions and User-oriented predictions. The sections 4.4.5 also show the distribution of different variables, such as Gas_prices and time a transaction needs to wait before being recorded in the blockchain. The error margins are summarized in Table 4.17, comparing the results of the user-orinted model to the existing data-centered model of the Gas Oracles.

### Transactions Data Analysis

Table 4.13 shows the statistics of the transactions data-set. The mean, the standard deviation (SD), minimum (min), the 25th, 50th, and 75th percentiles and maximum (max) are calculated for each variable shown in the table. The main variable is the "gas_price" value for each transaction included in a specific block.

Figure 4.16a shows the violin plot of the waiting time in seconds before a transaction is included into a block. The plot shows the presence of a peak at the value of 20 seconds with a tail that tends towards infinity. Figures 4.16b, 4.16c show the violin plots of the waiting time of the transactions for different Gas prices. Interestingly, the violin plots show that the Gas price attached to the transaction influences the interval of time the transaction needs to wait before being included into a block. The violin plots also present the same peek at the value of 20 seconds regardless of the Gas price.




(a) (b) (c)

Figure 4.16: Violin Plot of the waiting time in seconds before a transaction is included into a block. (a) All transactions. (b) Transactions having a Gas price lower than 10 GWei. (c) Transactions having a Gas price higher than or equal to 10 GWei.

### Block Data Analysis

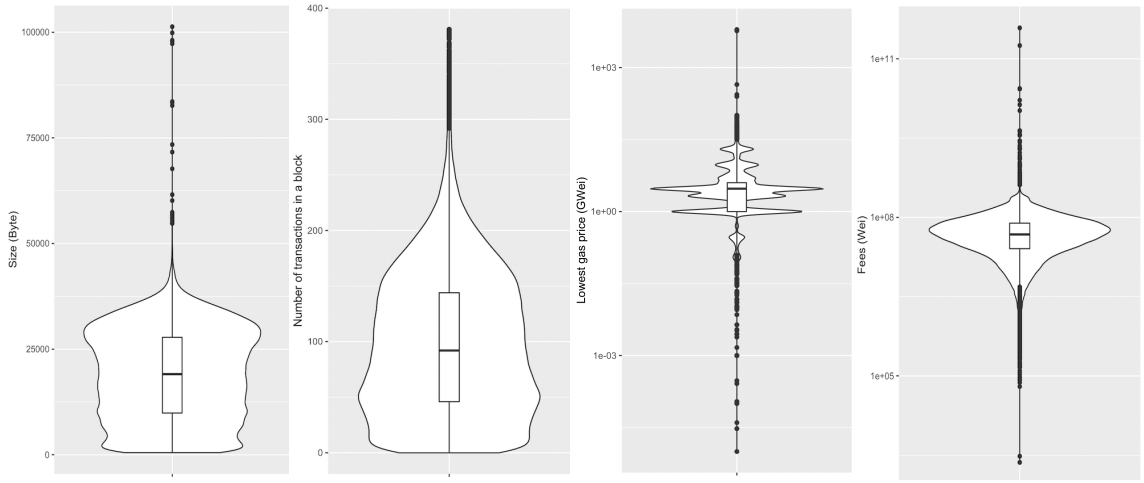
The blocks data-set gives information about each block, based on its height, *i.e.,* the index number that denotes its position in the blockchain. The data included in the blocks data-set are:

- the total number of fees in GWei, collected by miners in each block (fees);
- the size of the block (including the header and all the transactions) in Bytes (size);
- the number of transactions in each block (n_tx);
- the lowest Gas price attached to a transaction included in each block (lowest_gas_price).

Table 4.14 shows the statistics of the blocks data-set. The mean, the standard deviation (SD), minimum (min), the 25th, 50th, and 75th percentiles and maximum (max) are reported for each variable. Figures 4.17a, 4.17b, 4.17c, 4.17d show the probability density of each block variable at different values.

Table 4.14: Statistical description of the Ethereum blocks data (from the 6 871 349 th block to the 7 694 005 th block)

| | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|
| fees (Wei) | 6.4e+07 | 1.3e+09 | 0 | 2.45e+07 | 4.6e+07 | 7.7e+07 | 3.8e+11 |
| size (Bytes) | 18843.2 | 10580.7 | 524 | 9877 | 19045.5 | 27784.8 | 101294 |
| n_tx | 97.859 | 63.6377 | 0 | 45 | 92 | 144 | 381 |
| lowest_gas_price (GWei) | 4.46269 | 27.61 | 0 | 1 | 3 | 4 | 6215.03 |
| block_time (s) | 13.9453 | 12.9755 | 0 | 5 | 10 | 19 | 153 |




(a) (b) (c) (d)

Figure 4.17: (a) Block size (including header and all transactions) in bytes. (b) Violin plot of the number of transactions included in each block. (c) Lowest Gas price attached to each transaction and present in each block. (d) Violin Plot of the total fees (in Wei) collected by the miners in each block.

Table 4.15: Statistical description of Oracles categories

| | mean | std | mode | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| *'fastest'* (GWei) | 15.33 | 6.60 | 20 | 3 | 10 | 20 | 20 | 61 |
| *'fast'* (GWei) | 4.58 | 2.42 | 3 | 3 | 3 | 3.6 | 5 | 60 |
| *'average'* (GWei) | 2.83 | 0.80 | 3 | 1 | 3 | 3 | 3 | 14.5 |
| *'safeLow'* (GWei) | 1.34 | 0.68 | 1 | 1 | 1 | 1 | 1.1 | 14.5 |

**Oracles Data Analysis**

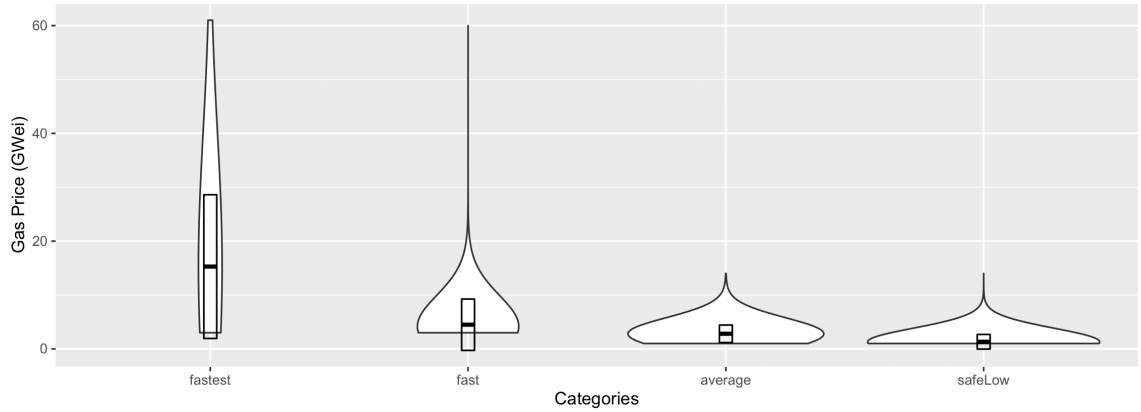
We analyzed the predictions data of the Oracles, Etherchain and EtherGasStation. The Gas Oracles can diversely predict the *Gas price* values to attach to transactions to have the transaction included at the most within n blocks.

Figure 4.18a shows the violin plots of the EtherGasStation Oracle's *Gas price* predictions for each *Gas price* category: 1) *'fastest'*, 2) *'fast'*, 3) *'average'*, and 4) *'safeLow'*. The *'recommended Gas price'* range is highly variable and the variability depends on each category. For example, for category *'fast'*, the *'recommended Gas price'* ranges from a maximum of 61 GWei to a minimum of 1 GWei and the most frequent value is 20 GWei. On the other side, the category *'safeLow'* has a *'recommended Gas price'* which ranges from a maximum of 15 GWei to a minimum of 1 GWei and the most frequent value is 1 GWei.

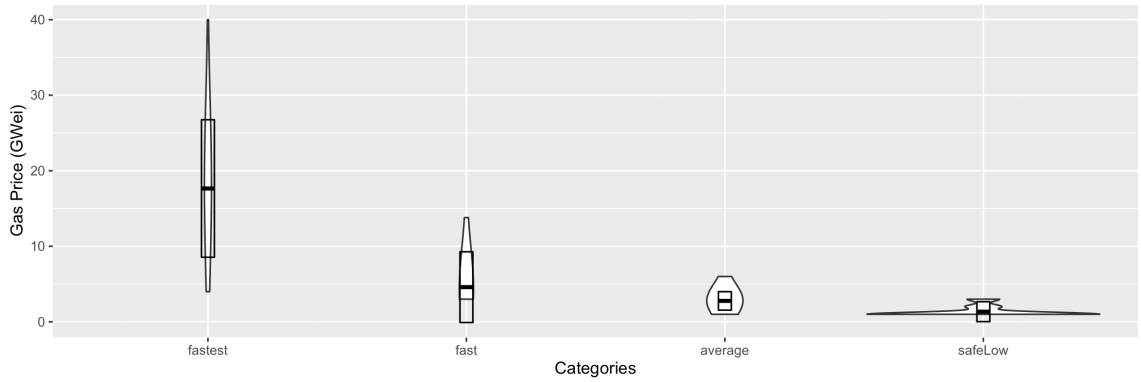
Figure 4.18b shows the violin plots of the Etherchain Oracle's *Gas price* predictions for each *Gas price* category: 1) *'fastest'*, 2) *'fast'*, 3) standard, 4) *'safeLow'*. Likewise the EtherGasStation, the *'recommended Gas price'* range of the Etherchain is highly variable and the variability depends on each category. For instance, for the category *'fast'*, the *'recommended Gas price'* ranges from a maximum of 61 GWei to a minimum of 1 GWei and the most frequent value is 20 GWei, while the category *'safeLow'* has a *'recommended Gas price'* which ranges from a maximum of 15 GWei to a minimum of 1 GWei and the most frequent value is 1 GWei.

Table 4.15 reports the mean, the standard deviation (SD), the mode, the minimum (min), the first quartile (25%), the median (50%), the third quartile (75%) and maximum (max) of the Gas price recommendation for the transactions for each Gas price category.

Figure 4.19a shows the violin plot of the *Gas price* prediction according to the Etherchain Oracle (in blue) and EtherGasStation Oracle (in orange). The values refer to the Gas price to pay to have the transactions confirmed within 1-2 blocks. Figure 4.19b presents the percentage of Gas price categories corresponding to the *Gas price* actually selected by the users in the Ethereum blockchain.

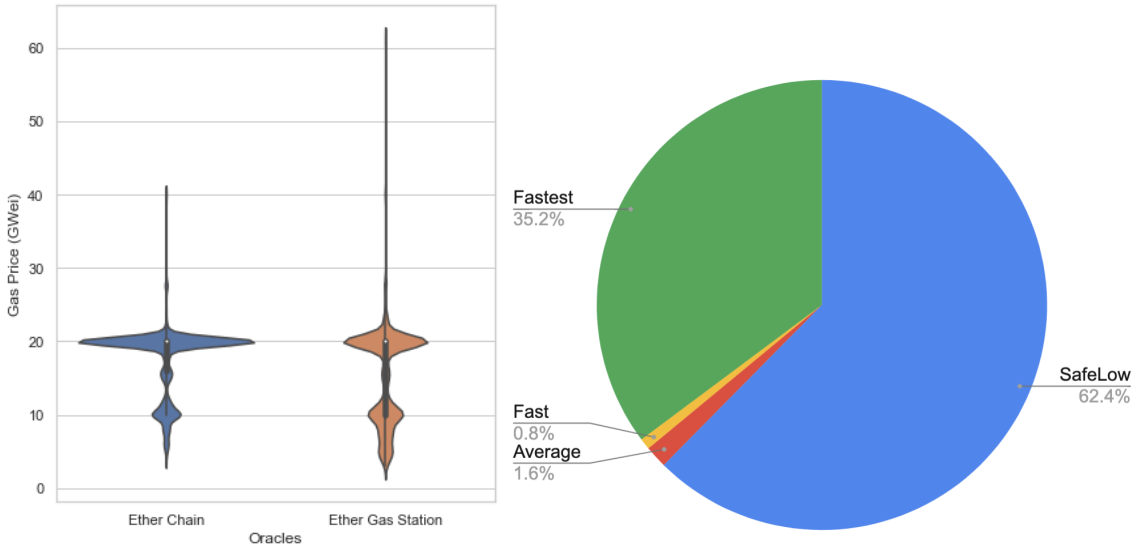



(a)




(b)

Figure 4.18: (a) Violin plot of the EtherGasStation Oracle's Gas price categories. (b) Violin plot of the Etherchain Oracle's Gas price categories.




(a) (b)

Figure 4.19: (a) Violin plot of the Oracles' Gas price prediction for the *'fastest'* category (b) Gas Oracles Categories corresponding to the *Gas price* actually set by the users.

### Evaluation of the Poisson Model

The Oracles assume that the observed data are distributed in accordance to the Poisson Model. Before checking whether the observed data actually follow the Poisson distribution, we checked the null hypothesis ($H_0$) that the observed data are homogeneously distributed over time among all blocks. In other words, we tested the hypothesis that the observed data are distributed in accordance with the Equiprobable Model, which predicts that the transactions have the same probability of ending in any of the next n blocks. Equation 4.5 expresses the expected probability to have the transaction included in any block as predicted by the Equiprobable Model:

$$H_0 : p_1 = p_2 = \ldots = p_{200} = 0.005 \tag{4.5}$$

where $p_i$ is the probability to have a transaction added to the $i^{th}$ block and it goes from 1 to 200. Table 4.16 (1st section) presents the results of the comparison between the expected frequency and the observed frequency based on the Equiprobable Model.

Therefore, we tested the alternative hypothesis that the observed data follow a Poisson distribution with parameter $\lambda > 0$. Equation 4.6 expresses the expected probability to have the transaction included into a block i based on the Poisson Model.

$$H_0 : \forall i \in [0, 200), P(X = x) = \frac{i * \lambda^x * e^{-(i*\lambda)}}{x!}, x \in [0, \infty) \tag{4.6}$$

Table 4.16: Null Hypothesis: the distribution of the transactions included in the blocks follows the Equiprobable vs Poisson Model

| Model | Category | lambda | X-squared | df | p-value | Decision |
|---|---|---|---|---|---|---|
| Equiprobable | *'fastest'* | not applicable | 864.1 | 199 | $< .00001$ | Rejected |
| | *'fast'* | | 910.9 | 199 | $< .00001$ | Rejected |
| | *'average'* | | 898.5 | 199 | $< .00001$ | Rejected |
| | *'safeLow'* | | 963.8 | 199 | $< .00001$ | Rejected |
| Poisson | *'fastest'* | 1.1295 | 190.86 | 199 | .648094 | Accepted |
| | *'fast'* | 1.3435 | 182.11 | 199 | .799031 | Accepted |
| | *'average'* | 1.3437 | 175.82 | 199 | .880322 | Accepted |
| | *'safeLow'* | 1.4973 | 182.12 | 199 | .798881 | Accepted |

*$p < 0.05$ means that the hypothesis is rejected, as there is a statistically significant difference between the expected frequency and the observed frequency.

Listing A1 shows the R code used to calculate the expected frequency of transactions per block based on the Poisson Model.

Listing A1: R code to compute the expected counts of transactions

```
1  blocks = 1:200 #list of blocks
2  total = sum(observed)
3  expected =
4      total * ((lambda^blocks)*exp(-lambda)) /
5          factorial(blocks)
```

Table 4.16 (2nd section) presents the results of the comparison between the expected frequency and the observed frequency based on the Poisson Model. The results are divided into four categories, as per Oracles' definition.

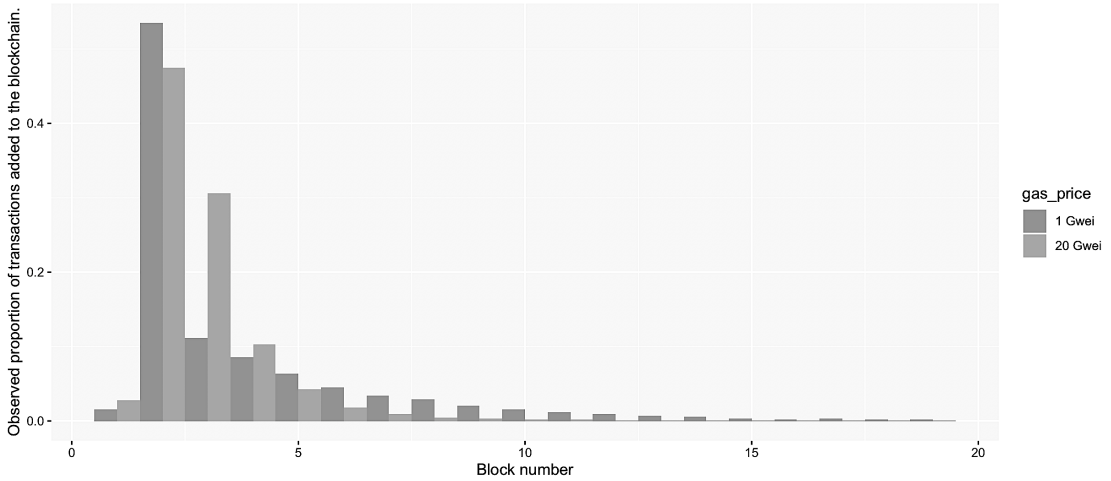



Figure 4.20: Histogram of observed data.

### 4.4.6 Evaluation of Oracles' Prediction

The Gas Oracles claim that at least 98% of transactions will be included at most into the next n blocks, if the Gas price of the transactions is equal to or greater than the Gas price they recommend. The Gas Oracles' predictions are four, one for each category. The n blocks are equal to 2, 4, 20 and 120 respectively for the categories *'fastest'*, *'fast'*, *'average'*, and *'safeLow'*. To verify the Gas Oracles' predictions, we collected a sample of *1.1M* transactions and for each category we calculated the proportion of transactions added within the 2-nd, 4-th, 20-th and 120-th blocks. We found the following percentage of transactions for each category: 92% for *'fastest'*, 89% for *'fast'*, 87% for *'average'* and 94% for *'safeLow'*. The results suggest that the Gas Oracles' predictions might be wrong, as all the percentages are lower than 98%, i.e. the percentage declared by the Oracles. However it might be claimed that the Gas Oracles are actually right, as the discrepancy is just due to statistical fluctuations occurring in the 100 blocks latency time the Oracles take to recalculate the lambda value. We therefore tested both the hypotheses: the hypothesis that the Oracles successfully predict the Gas price to pay to the miners (null hypothesis) and the hypothesis that their predictions are wrong (alternative hypothesis).

Equations 4.7, 4.8 respectively represent the null and the alternative hypotheses.

$$(H_0)\ \forall\ cat \in \{\text{'fastest'}, \text{'fast'}, \text{'average'}, \text{'safeLow'}\} : p >= 98\% \tag{4.7}$$

$$(H_a)\ \forall\ cat \in \{\text{'fastest'}, \text{'fast'}, \text{'average'}, \text{'safeLow'}\} : p < 98\% \tag{4.8}$$

Listing A2 represents the R code used to test the null hypothesis that the Oracles' prediction are right within a frame of 100 blocks time.

Listing A2: Test of Equal or Given Proportions

```
1  prop.test(x = transactions, p = 0.98, correct = FALSE,
2            alternative = ``less'')
```

The variable x represents a two-dimensional table with 2 columns, which respectively provide the number of successful events (transactions included in the first n blocks) and failures (transactions included after the n-th block). The variable p represents the expected proportion of successful events ($P_e$), based on the Gas Oracles' predictions. The variable "alternative" specifies the proportion of successful events based on the alternative hypothesis.

Table 4.17 represents the results of the null hypothesis $H_0$ (Eq. 4.7) that the observed proportion ($P_o$) of transactions included in the blocks are equal or greater than the expected proportion ($P_e = 0.98$ ). The table is divided into four sections based on latency (2nd column). In particular, the first section represents the results based on 100 blocks latency, as claimed by the Oracles.

Table 4.17: Alternative Hypothesis: the observed proportion ($P_o$) of transactions included in the blocks at latencies $< 100$ blocks are equal or greater than the expected proportion ($P_e$) of the Gas Oracles.

| Category | Latency | $P_o$ | $P_e$ | X-squared | df | p-value | Decision |
|---|---|---|---|---|---|---|---|
| *'fastest'* | 100 | 0.92 | 0.98 | 1594.1 | 702 | $< .00001$ | Rejected |
| *'fast'* | 100 | 0.84 | 0.98 | 310.9 | 102 | $< .00001$ | Rejected |
| *'average'* | 100 | 0.91 | 0.98 | 698.5 | 389 | $< .00001$ | Rejected |
| *'safeLow'* | 100 | 0.90 | 0.98 | 1463.8 | 1071 | $< .00001$ | Rejected |
| *'fastest'* | 80 | 0.92 | 0.98 | 9212.8 | 601 | $< .00001$ | Rejected |
| *'fast'* | 80 | 0.89 | 0.98 | 763.9 | 89 | $< .00001$ | Rejected |
| *'average'* | 80 | 0.87 | 0.98 | 3295.2 | 301 | $< .00001$ | Rejected |
| *'safeLow'* | 80 | 0.99 | 0.98 | 1053.4 | 994 | .093174 | Accepted |
| *'fastest'* | 60 | 0.96 | 0.98 | 523.2 | 402 | .000042 | Rejected |
| *'fast'* | 60 | 0.99 | 0.98 | 101.3 | 81 | .063046 | Accepted |
| *'average'* | 60 | 0.99 | 0.98 | 241.2 | 207 | .051696 | Accepted |
| *'fastest'* | 4 | 0.99 | 0.98 | 230.1 | 201 | 0.077864 | Accepted |

*$p < 0.05$ means that the hypothesis is rejected, indicating that there is a statistically significant difference between the observed proportion ($P_o$) and the expected proportion ($P_e$).

### 4.4.7 Improving the Oracle Prediction

The Gas Oracles assume that the transactions' distributions have different lambda values of the PMF (Eq. 4.3), based on the Gas price. Figure 4.20 shows that the tail length of transactions' events is inversely proportional to the Gas price. Based on this assumption, the Oracles calculate the $\lambda$ (and as a consequence the Gas price) every time 100 blocks are confirmed on the blockchain (100 blocks latency). We hypothesized that it is possible to improve the Oracles' performance by recalculating the $\lambda$ of the PMF (Eq. 4.3) at intervals of time smaller than 100 blocks latency. Reducing the latency, we might indeed better take into account the possible network changes (**?**). The network might have changed depending not only on the increasing vs. decreasing number of miners and/or transactions in the network, but also and more importantly on the users' actual decisions on the Gas price to pay and time to wait. We therefore reduced the latency for each category, until the null hypothesis $H_0$ (Eq. 4.7) is accepted with a level of significance of $\alpha = 5\%$ ($p < 0.05$). Table 4.17 represents the results of the hypothesis $H_0$ (Eq. 4.7) that the observed proportion ($P_o$) of successful events are equal or greater than the expected proportion ($P_e = 0.98$) at latencies smaller than 100 blocks.

### 4.4.8 Discussion

The transactions data-set consists of over 10 million rows which covers a period of time of 3 months. This is a relatively small fraction compared with the total number of transactions in the same period, which should be around 77 million (`https://etherscan.io/chart/tx`). Of course, the blockchain networks can change for many reasons. Previous research (**?**) shows that the number of transactions moving through the Ethereum network can increase or decrease based on specific users-related events, for instance when a company looks to raise money to create a new coin, app, service, etc. or when it launches an ICO. Another scientific research (**?**) shows that there are several conditions under which mining infrastructures will be active or under which the miners will have no incentives to mine a given cryptocurrency due to the increase of the energy cost or unavailability of solar energy which can be used to make calculations at no cost. These or similar remarks are at the core of the idea that the Oracles' data-centered model might provide wrong predictions, because it does not take into account the network changes depending on the users' actual behaviour or users-related events. This is the main reason why the research presented in this work proposes a model shift, from a data-centered model to a user-oriented model for Oracles' *Gas price* predictions.

#### From a Data-Centered Perspective

The data-centered model is actually used by the Oracles, to provide the users with the predictions every 100 blocks confirmation. The model rely on data on the transactions history of the last 200 blocks confirmed on the blockchain (`https://github.com/ethgasstation/ethgasstation-backend`). The analysis showed that most transactions wait from one to two blocks before being included (see Table 4.13 ). The results of the analysis also showed that the Gas price influences the probability to have the transactions included into the next blocks. Figure 4.16a indeed shows that the shape of both the violin plots becomes larger at the decreasing of the Gas price. In particular, a transaction's Gas price higher than 10 GWei does not guarantee that the transaction is included within 1-2 blocks. The probability is anyway higher when compared to the transactions having a Gas price lower than 10 GWei.

Once data have been cleaned 4.4.4, we investigated the model actually followed by the Oracles to provide a *Gas price* prediction, i.e. the Poisson model. We defined the Poisson Model for the successful events of having a transaction included into a block. We pointed out that not all its conditions (defined in Section 4.4.4) are satisfied. We anyway checked whether the successful transactions were distributed in accordance with the Poisson Model instead of an equiprobable model. First, we tested the hypothesis that the transactions' distribution follow the Equiprobable Model. However, we found strong evidence that the Equiprobable Model does not fit the data, as per p-value = $2.2^{-16}$ inferior to 0.001. Second, although not

all the conditions are met, we tested the alternative hypothesis that the successful transactions' distribution follow the Poisson Model. We found that the alternative hypothesis cannot be rejected with a confidence level of 95%. While in an Equiprobable Model it would make no sense to predict a *Gas price*, the Poisson Model does provide us with a meaningful insight to predict a *Gas price*, as it gives a lambda value which is inversely proportional to the transactions' Gas price. Indeed, as shown in Figure 4.20, the lower the transaction Gas price, the longer the queue of Poisson distribution is.

### To a User-Oriented Perspective

By performing the *Poisson regression model* every 100 blocks, the Gas Oracles do not take into account all the changes in the Ethereum Network in real-time. As hypothesized in H1, the Gas Oracles' predictions based on the data-centered model are not reliable, especially because they compute the prediction every 30 minutes on average. We showed that the Oracles' predictions are not just accidentally wrong, due to possible statistical fluctuations in 100 blocks latency. We indeed found that the Oracles' prediction are actually wrong with a level of confidence of 98%. Moreover, we found that the margin of error is greater than declared (2%) and it is even 13% for the *'fastest'*. The greater the latency, the more probable is that the Poisson model cannot take into account also user-related events or decisions occurring in the network.

Some special scenarios - our model can successfully manage - are those related to the network congestion which can happen when a company looks to raise money to create a new coin, app, or service (**?**). These scenarios are special because they may imply a longer waiting time before the transaction is included and committed, because there are more transactors offering a higher Gas price when compared to a non-congested network state. One of the ways that may be used to reduce the waiting time is to make users aware that a higher fee is needed to have the transaction included into a block in a shorter time than usual. As our model for the fastest category is trained every 4 blocks, it can recommend Gas prices that reflect the current possibly congested situation instead of past situations when there was no congestion.

It is not possible to be sure that the users who submit the transactions are following the Gas Oracles' recommendation. However, it is reasonable to assume that the users who set the Gas price equal to the Gas price suggested by the Oracle are indeed following the Oracle's recommendation. Even if they were not following the Oracle's recommendation, it is likely that the user agrees with the Gas price attached to the transaction and the waiting time. If the users disagree with the waiting time, they would change the Gas price to rely on the expected waiting time. Anyway, in a data-centered model, this would not influence the analysis, as it is based on the statistics about Gas price and waiting time recorded.

However, the analysis of the Gas price of the transactions in the transaction pool

shows that just 16% of the transactions have a Gas price equal to the Gas price suggested by the Gas Oracle. The percentage of transactions having the Gas price equal to the Gas price suggested by the Oracle is distributed among the four categories as follows: 1) 7% *'safeLow'*, 2) 1% *'fast'* and *'average'*, 3) 8% *'fastest'*. Figure 4.19b presents the percentage of Gas price categories used in Ethereum blockchain. Table 4.15 shows how the mode for the *'fast'* and *'average'* categories are the same. This means that most times the Gas price is the same for both categories, in spite of being different categories in terms of execution time. The data analysis of the transactions waiting in the transaction pool to be included into a block also suggests that the categories *'fast'* and *'average'* are not set by the users probably because these categories do not correspond to their interests and/or needs. On the contrary, the categories *'fastest'* and *'safeLow'* are set the most. This means that, as hypothesized in H2, the users set a Gas price in relation to an interval time to include a transaction, which are not fully-fledged predicted by the default categories.

Both the Oracles present the same pattern of results: the distributions of the *'recommended Gas price'* are almost the same for Etherchain and EtherGasStation. The violin plots of both the Oracles also show the presence of two different peaks of *'recommended Gas price'* at the same value. One of the peak corresponds to the third quartile value, *i.e.,* 20 GWei, while the other peak is below the median and it is equal to 10 GWei. Figure 4.19a shows the violin plot of the *Gas price* predictions.

We also showed, as hypothesized in H3, that a reduction of the margin of error in the Gas price prediction can be achieved by reducing the latency, thus considering the current changes of the network. The Oracles' prediction can indeed be improved, by reducing the latency of 100 blocks time. The results confirmed that the margin of error is reduced for all the categories, when the *Poisson regression model* is performed at shorter time intervals. Interestingly, the results also showed that the margin of error depends on the latency and it is different for each category. In particular, the category *'fast'* requires a shorter latency compared with the other categories. The *'fast'* category is indeed more demanding than the others in terms of waiting time, as the Gas Oracles estimate the Gas price to have the transaction included within two blocks at most. On the contrary, in the case of other categories, such as the *'safeLow'*, the Gas price does not need to be predicted so often.

### 4.4.9 Conclusions

The existing Gas Oracles are based on a data-centered model which relies on the analysis of the blocks data history to make the Gas price prediction, without considering any data on the categories the users actually set. To propose a user-oriented model of Gas Oracles' Gas price prediction, the study explored both the overall validity of the Gas Oracles' predictions and the more specific validity of the Gas Oracles' Gas price categories when compared to the categories actually set by the users.

- The study first evaluated the validity of the Gas Oracles' predictions on the Gas price to pay to have the transaction recorded in the blockchain. It revealed that both Etherchain and EtherGasStation predict with a margin of error at least twice as much as the margin of error they declare. For instance, the *'fastest'* category showed a 4% margin of error, while the *'fast'* category showed a 28% margin of error. The user-oriented model proposed in this research gives a prospective contribution to the improvement of the Gas Oracles' predictions to better indicate the categories that correspond to the users' requirements and the Equation that best provides them with a more effective Gas price to set.

- The study shows that the four Gas price categories proposed by both the Oracles do not match the categories actually set by the users and/or companies. As a result of the analysis, we found indeed that less than 1% of transactions set the Gas price suggested by the Gas Oracle in the categories *'average'* and *'fast'*. On the contrary, we found that it is worth predicting the Gas price for the *'fastest'* and *'safeLow'* categories, as they make sense in terms of users' interests.

- The study contributes to the understanding of the Equation the Gas Oracles should use to provide the users' with a better Gas price prediction. The user-oriented model we propose recommends indeed to calculate the Gas price by reducing the latency of 100 blocks time to have a lower margin of error compared to the Oracles' actual one. The study shows that, by reducing the latency to perform the *Poisson regression model*:

  - the error margin of the prediction for the *'fastest'* category is 2% compared to the 4% of the Gas Oracles' prediction. In this case, we performed the *Poisson regression model* every 4 blocks instead of 100 blocks.
  - the error margin of the prediction for the *'average'* and *'fastest'* categories is 1% compared to the 14% and 13% of the Gas Oracles' prediction. In this case, we performed the *Poisson regression model* every 60 blocks instead of 100 blocks.
  - the error margin of the prediction for the *'safeLow'* category is 1% compared to the 4% of the Gas Oracles' prediction. In this case, we performed the *Poisson regression model* every 80 blocks instead of 100 blocks.

  Table 4.17 summarizes the results. The model can provide the users with a better estimation, because it can take into account the current changes of the blockchain and the users' network in real time.

The main threat of the user-oriented model is the need to perform the *Poisson regression model* 25 times more for the category *'fast'* when compared to the data-centered model of the existing Gas Oracles. Indeed, although this user-oriented

model provides better results in terms of error margin when compared to the data-centered model of the Gas Oracles Etherchain and EtherGasStation, it has the disadvantage of being less efficient in terms of computational resources. The major novelty of this study lies in the very idea of a user-oriented model for the Oracles' *Gas price* predictions. Different from the existing model used to produce the Oracles' estimations, the model proposed in the study is aimed to predict the categories actually set by the users, also suggesting them a more precise estimation on the most effective *Gas price* to set for each category.

### 4.4.10 Related Work

Sin Kuang Lo et al. (**?**) investigated the reliability of seven Oracles on different platforms such as Augur, Ms Bletchley, TownCrier and Corda. They discovered that the common causes of failure are the data sources used by the Oracle to make various kinds of predictions such as the weather forecast. These failures can have a serious impact on the economy because many smart contracts perform operations on the basis of these predictions. To meet the users needs, they provided a framework that can be used to assess the Oracles' reliability. The authors' work supports the interesting idea that, providing this framework together with the Gas Oracle, it is possible to help the users' decision making. Differently from their work, in this research, I study the behavior of the Oracles that predict the Gas price in the Ethereum blockchain (**??**).

## 4.5 AI Techniques for Detecting Malicious Smart Contracts

### 4.5.1 Introduction

The Ethereum's network has millions of participants, and some of them could likely be malicious. Usually, a malicious user is defined as a user who abuses his privileges to harm other users. Since we are considering the Ethereum context, we refer to a malicious user as a network participant that tries to convince other participants to invest their money in high-risk contracts, in which there's no guarantee of getting back their Ethers. These types of contracts are obviously considered fraudulent contracts. A particular type of fraud is that of the Ponzi Scheme (**?**), a pyramidal model in which investors are recruited with the promise of easy earnings and high-interest rates relative to the initial investment. Other participants of the scheme can also recruit new investors to increase their earnings. This last point is interesting in the Ethereum context because many of the contracts identified as Ponzi Scheme require that a scheme participant recruits a new investor to get his money back (**?**). Previously we said that a malicious user abuses his privileges. The main privileges provided by blockchain are (**?**):

- **Anonimity**: thanks to inner properties of blockchain the head of the Ponzi Scheme can stay anonymous, and so the other participants of the scheme.
- **Ummutable and potentially unstoppable contracts**: Once a smart contract is deployed, it is not possible to modify it, therefore making impossible to break the scheme. Potentially, smart contract execution can't be stopped by any central authority though some Smart Ponzi put an exit condition, for which that scheme can not be executed anymore. This condition comes true when there are not sufficient funds to pay a participant. Obviously, a malicious user may want to exploit all the advantages provided by the blockchain technology, so this exit condition bring us to think that this types of contract are just either proof of concepts or experiments.
- **Trustworthiness**: Most of the time, a smart contract's source code is available on the blockchain. The availability and the transparency of the source code transmits trust to a user, since a potential participant is more enticed to invest money in a contract which clearly shows how it works instead of a contract which keeps hidden its inner properties.

Actually, some of the identified Smart Ponzi are just experiments (**?**) since the nature of the contract is explicitly explained by the contract's creator with comments or with the names of the variables and functions. Terms like participants, investments, payout, investors are quite common inside these contracts. Despite some of these contracts being just experiments, the problem of possible Ponzi Schemes scams is real, and it is important to prevent it from happening. To do so, one must identify Ponzi contracts and classify them; this is possible because a Ponzi Scheme contract is recognizable by its execution flow and some well-defined rules. The cyclic flow ensures that the head of the pyramid (who is the one that starts the scheme) finds potential investors. Likewise, investors recruit new participants. Funds from new participants are used to pay the previous investors that joined the scheme. The scheme goes on until it is broken. The general rules of a Smart Ponzi are:

1. A minimum investment is required to join the Ponzi scheme.
2. Investors starts to get paid only if funds are sufficient.
3. If new investors are not available anymore, the scheme is broken.
4. If funds are not sufficient to pay the investors, the scheme is broken.

These rules are common to all Smart Ponzi contracts despite some details that can be slightly different. For example, the minimum fee required to join the scheme differs from contract to contract. In addition to these rules, there are some other elements common to all Smart Ponzi. These are:

1. The contract's **owner**, which starts the scheme.

2. **Investors**, which are defined trough their address and their balance.
3. An array or mapping that stores all the scheme participants.
4. A function that allows a new participant to join the scheme.
5. A function to pay the participants.

It is also interesting to notice that there are different categories of Ponzi schemes contracts. Previous work (**?**) defined four categories and a taxonomy of Smart Ponzi. First, we have tree-shaped schemes that use a tree data structure to induce an ordering among users. In this particular category of schemes, the user who has just joined the tree must indicate the participant who invited him. The user who invited the new participant becomes his parent node in the tree. Another category is that of chain-shaped schemes, which are a special case of tree-shaped schemes. These types of schemes multiply the investment by a constant factor. Waterfall schemes have a different logic of money distribution. Here each new investment is poured along the chain of investors so that each can take their share. Eventually, we have Handover schemes, where the fee to join the scheme is determined by the contract. This fee gets an increased value each time a new investor joins the scheme. Looking at the already classified Ponzi Schemes contracts on the Ethereum blockchain, we noticed that a lot of them were published between 2016 and 2017. We aim to build a classifier trained with all the contract that have been spotted in the past few years as well as some which don't belong to this group, as they are not Ponzi schemes. It is interesting to notice that a lot of the already known schemes are very similar between them, and it is clear that a lot of developers took inspiration from others' work. We can find also different versions of the same contract with slight differences (variable names or some unimportant operations).

### 4.5.2 Related Work

The first Ponzi scheme contracts date back to 2015, the year of the launch date of Ethereum, but almost all of the Smart Ponzi contracts of our dataset are dated between 2016-2018. Since they appeared in the blockchain (not only in that of Ethereum but also in that of Bitcoin) the Ponzi scheme and fraudulent contracts in general (**?**) have aroused great interest in the scientific community. This is because these schemes are potentially dangerous for participants who invest their money unconsciously and that the inherent properties and advantages offered by a blockchain make them even more effective. The main advantage that a fraudulent smart contract can take advantage of is that of the guarantee of anonymity offered by the blockchain. This makes a scam virtually impossible to track, which is why such schemes need to be spotted. Currently, several solutions have already been explored to identify Ponzi schemes within the blockchain, both in that of Bitcoin and in that of Ethereum. To recognize Smart Ponzi within Bitcoin, several approaches

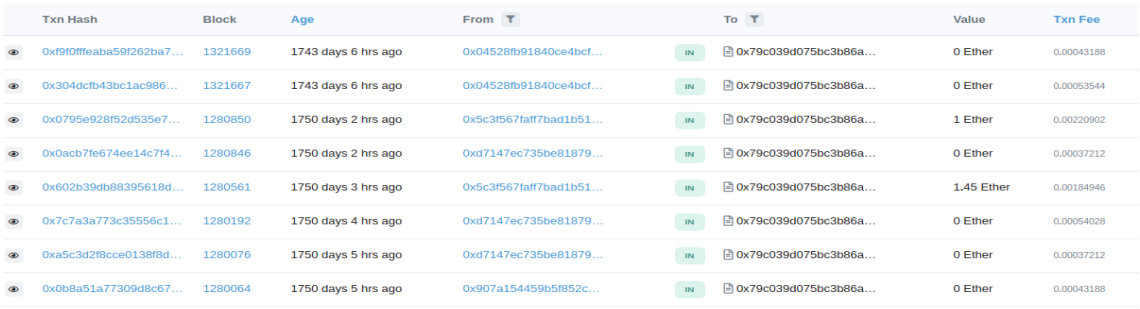
were used, including those oriented to data mining (**?**). The idea is to apply data mining techniques to detect Bitcoin addresses related to Ponzi schemes, considering features like the lifetime of the address, the activity days, the sum of all the values transferred to the address and others; the problem is seen as a binary classification problem, in which a classifier must recognize between 'Ponzi' and 'non-Ponzi'. In the field of Bitcoin, using survival analysis, factors that affect scam persistence have been spotted (**?**). Vasek and Moore found out that when the scammer interacts a lot with their victims, the scam's life is increased, and that scams are shorter-lived when the scammers register their account on the same day that they post about their scam.
The Ethereum blockchain has also aroused great interest in the search for Ponzi scheme contracts. Previous work (**?**) examined all the contracts with a certified source code, determining if they were implementing a Ponzi scheme. They also analyzed the source code of smart contracts (when available) and discovered that most of the contracts share common patterns. For the classification of Smart Ponzi on Ethereum, a proposed model classifies whether a contract implements a Ponzi scheme or not by using features extracted from the transaction history and from the opcodes of smart contracts (**?**). In general, we can say that data science approaches are widely used not only to detect scams but honeypots in general (**?**). Our approach is always based on solving a binary classification problem, but it differs as it is based on natural language processing techniques. In fact, we used the source code (when available) of smart contracts to create our features. The idea is that Smart Ponzi have a well-defined structure and use meaningful terms that identify them. It is very unlikely that there are Ponzi scheme contracts that do not make their source code clear, or that in any case try to make their operation ambiguous, because a participant is more inclined to invest his money in a contract that clearly shows their functioning, rather than in a contract whose functionality is not known. In the next sections we will see in detail our proposed approach for the classification of smart Ponzi schemes.

### 4.5.3 ResearchMethodology

#### Scraping

The use of machine learning concerning cryptocurrencies is becoming increasingly important, especially for the classification of smart contracts and predictions related to the trend in the currency market (**?**). Our idea is to build a machine learning model taking advantage of Natural Language Processing (NLP) techniques. In particular, we want to exploit Text Classification techniques to classify Ponzi schemes contracts, but first, we must determine what textual features will be part of our dataset. All Ponzi schemes contracts have a common execution flow and structure, so, we may take advantage of this information. The first text feature we identified is the source code of the smart contracts when available. We had this feature for

the entire dataset. Another text feature is the contracts' Opcode. We collected Opcodes for the entire dataset since the source code of all the collected contracts was available. The idea is that since Smart Ponzi contracts have a similar structure, their Opcodes will be similar as well. The last text feature consists of pieces of information extracted from contracts' transactions. Again, we took advantage of the cyclic execution flow of a Ponzi scheme and key terms. The flow consists in:



| Txn Hash | Block | Age | From | | To | Value | Txn Fee |
|---|---|---|---|---|---|---|---|
| 0xf9f0fffeaba59f262ba7... | 1321669 | 1743 days 6 hrs ago | 0x04528fb91840ce4bcf... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00043188 |
| 0x304dcfb43bc1ac986... | 1321667 | 1743 days 6 hrs ago | 0x04528fb91840ce4bcf... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00053544 |
| 0x0795e928f52d535e7... | 1280850 | 1750 days 2 hrs ago | 0x5c3f567faff7bad1b51... | IN | 0x79c039d075bc3b86a... | 1 Ether | 0.00220902 |
| 0x0acb7fe674ee14c7f4... | 1280846 | 1750 days 2 hrs ago | 0xd7147ec735be81879... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00037212 |
| 0x602b39db88395618d... | 1280561 | 1750 days 3 hrs ago | 0x5c3f567faff7bad1b51... | IN | 0x79c039d075bc3b86a... | 1.45 Ether | 0.00184946 |
| 0x7c7a3a773c35556c1... | 1280192 | 1750 days 4 hrs ago | 0xd7147ec735be81879... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00054028 |
| 0xa5c3d2f8cce0138f8d... | 1280076 | 1750 days 5 hrs ago | 0xd7147ec735be81879... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00037212 |
| 0x0b8a51a77309d8c67... | 1280064 | 1750 days 5 hrs ago | 0x907a154459b5f852c... | IN | 0x79c039d075bc3b86a... | 0 Ether | 0.00043188 |

Figure 4.21: Example of contracts transactions

1. Contract creation by the pyramid leader.
2. Recruitment of investors.
3. The just recruited investors recruit new participants as well.
4. Payment of the investors.
5. Repeat from point 2 until the scheme is broken.

Considering this execution flow, it's reasonable to think that in Ponzi schemes contracts, we will find transactions performed to execute the function to allow a new participant to join the scheme and transactions that execute functions to pay the investors. Usually, these functions have well-defined names such as join, enter, pay, payout, and similar terms; the idea is to take advantage of these pieces of information to recognize a Smart Ponzi.
The next step in our recognition procedure is the feature extraction. All the text information is available on Smart-Corpus (**?**). All one needs is the contract's address, and then scraping from the web page its relative source code, opcode, and transactions is straightforward. We started from all the collected contracts' addresses, and then we used Python to perform a web scraping task to retrieve all the needed information. In particular, we used the *beautifulsoup* module to perform web scraping and clean the text information from HTML tags and elements.

### Classification

We started from a dataset made of already known Ponzi schemes and of contracts that are not Ponzi schemes. We collected 171 Smart Ponzi contracts and 1500 contracts that are not Ponzi schemes, so our dataset is strongly unbalanced.

We saw the Ponzi schemes contracts classification problem as a binary classification problem and we assigned the binary target variable with value 1 for a Smart Ponzi and 0 for other contracts. Section 4.5.3 provides an overview about the models used for the Ponzi schemes contracts classification problem.

### AST extraction

To have easy access to the contracts' variables, statements, conditions, and all the constructs in general, we extracted all the Abstract Syntax Trees (AST) from our samples. We used python as the programming language to build our solution. Extracting AST from contracts using python is a trivial task thanks to the *solidity_parser* module. The module only requires the user to load the desired contract for which the AST is then extracted in a JSON file after parsing the source code.

### Semantic Information Extraction

To extract information from AST, we must parse it. First, to parse a Solidity contract AST we must know its structure, defined by Solidity's grammar. Checking grammar, we know how our AST fields are defined; some of the types defined are trivial, like *VariableDeclaration*, which contains only a variable name and its type. Other nodes could contain quite complex types and require a deep exploration of the subtree. Types like *BinaryOperation*, *MemberAccess*, and *IndexAccess* (for example) require a recursive exploration of the subtree because they could have operations of the same type (i.e. a BinaryOperation could have another BinaryOperation inside). Once we built our AST parser, we extracted all the potential information relevant from a semantic viewpoint. Variables names, struct names, conditions inside constructs (like for, if, while), and the operations performed inside contracts are all relevant because we saw previously that some terms and some operations are recurrent in smart Ponzi schemes. To extract a semantic document from AST, we took all the labels previously listed and saved them in different structures. We tried to look for the extracted features as much as possible similar to something that suggests a natural language. To do this, we built a dictionary that replaces all the mathematics and logic operators with their semantic meaning (i.e. the symbol '=' is replaced with 'is assigned with value'). To add semantic to the resulting document, we added meaningful words related to the type of construct. For example, before a variable name or a struct name we added 'declaring', and after a struct name we added 'with members' followed by struct members name.Since this approach could sound tricky and probably not so easy to understand, we provide an example of the

'semantization' of the source code. Suppose to have a Solidity line of code like the one below.

```
1 if (msg.value < 50 finney)
```

The entire line of code is an 'IfStatement' field of the Solidity AST. In this specific case the expression inside this IfStatement, which is 'msg.value < 50 finney', is a BinaryOperation. Our parser decomposes the expression as follows: the symbol '<' is the operator of the BinaryOperation, while 'msg.sender' and '50 finney' are respectively the left part of the operation and the right part of the operation. The right part is simple, since it includes just a numerical value and a Solidity's keyword. The left part is not trivial, because it is a MemberAccess expression and it needs further decomposition. In this case, the expression is not too complex, since it has only one identifier, which is 'sender', so exploring the subtree requires only one additional step. Now that we decomposed the IfStatement the parser will build the corresponding line in the semantic document.

```
1 if msg value is less than 50 finney
```

Following this approach for all contracts, we were able to extract semantic documents from contracts' AST and perform text classification.

### Models

The models tested for the Smart Ponzi classification problem are: Decision Tree, Support Vector Machine (SVM),Multinomial Naive Bayes. Decision Trees are among the simplest classifiers and can be used for both classification and regression problems. As the name suggests, they work building a tree representation starting from data. We choose to test the Decision Tree just because it is one of the simplest models and because any missing values in the data do not affect the process of building a tree to any considerable extent. One possible problem is that this particular model is inadequate for applying regression and predicting continuous values (but this is not our case). Other problems consist in the average time required to train the model, which is very long, and that computations can get quite complex, especially when compared with other models. Consequently, despite being simple, the Decision Tree is quite an expensive model. The second model we tested is the SVM, which maps training examples to points in space. This model is memory efficient, and it works well when the two classes are linearly separated. The problem is that this model is not suitable for large datasets, and it is not easy to keep track of decisions taken by the SVM, and therefore, it is not simple to explain the obtained results. The Multinomial Naive Bayes consider a feature vector where a given term represents the number of times it appears or very often i.e. frequency. We choose to test this model because it has a low computational cost, can work with large datasets, and is well known to perform well in text classification problems, making it a perfect choice for our classification problem. The main disadvantage is that using a Naive Bayes is difficult to get the set of independent predictors for developing a model.

Considering the models' inner properties, we can make some considerations about

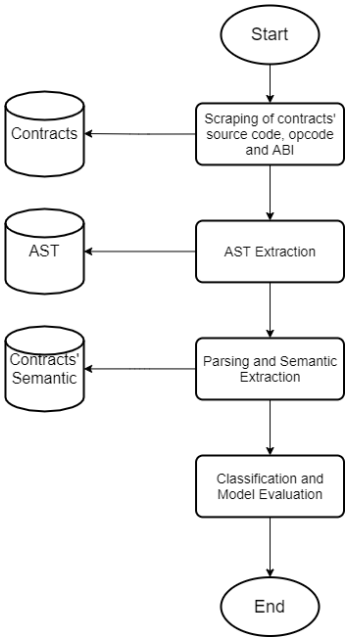



Figure 4.22: Proposed approach's workflow

the expected results. In terms of time complexity, we expect the Decision Tree as the slowest model to train, and the SVM as the fastest. We expect to reach the best performances with the Multinomial Naive Bayes Model, and at least to reach similar results with the Support Vector Machine, which should behave as well in this particular task.

### Classification Methodology

To perform the classification, we built two different CSV files. The first one has contracts' source code and opcode, while the second one has contracts' source code and transactions information. The idea is to test the models' performances making comparisons in terms of class metrics evaluation like accuracy, precision, recall, and f1-score. We want to check if these values are better with a model trained with source code and opcode or with a model trained with source code and transactions. First, we decided to use 80% of our dataset to build the training set and the remaining 20% to build the test set. Since we had fewer samples for the Ponzi schemes contracts, we decided not to use a validation set. Once we built our dataset, we dropped all the empty fields and possible duplicates, and then we converted our collection of documents to a matrix of token counts. Before training and testing the model, we normalized our data removing the words that commonly appear in the English language. We performed any character normalization, and we set a threshold to

Table 4.18: Classification score for Decision Tree model

| Features | Criterion | Target | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|---|
| Source code/Opcode | Gini | Not Ponzi | 0.98 | 0.99 | 0.99 | 0.97833 |
| | Gini | Ponzi | 0.94 | 0.89 | 0.91 | |
| Source code/Opcode | Entropy | Not Ponzi | 0.98 | 1.00 | 0.99 | 0.98555 |
| | Entropy | Ponzi | 1.00 | 0.89 | 0.94 | |
| Source code/Transactions | Gini | Not Ponzi | 0.99 | 1.00 | 0.99 | 0.98916 |
| | Gini | Ponzi | 0.97 | 0.94 | 0.96 | |
| Source code/Transactions | Entropy | Not Ponzi | 0.99 | 1.00 | 0.99 | 0.98916 |
| | Entropy | Ponzi | 0.97 | 0.94 | 0.96 | |

Table 4.19: Classification score for Multinomial Naive Bayes model

| Features | Fit Prior | Target | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|---|
| Source code/Opcode | True | Not Ponzi | 0.97 | 1.00 | 0.99 | 0.97472 |
| | True | Ponzi | 1.00 | 0.81 | 0.89 | |
| Source code/Opcode | False | Not Ponzi | 0.97 | 1.00 | 0.99 | 0.97472 |
| | False | Ponzi | 1.00 | 0.81 | 0.89 | |
| Source code/Transactions | True | Not Ponzi | 0.99 | 1.00 | 1.00 | 0.99277 |
| | True | Ponzi | 1.00 | 0.94 | 0.97 | |
| Source code/Transactions | False | Not Ponzi | 0.99 | 1.00 | 1.00 | 0.99277 |
| | False | Ponzi | 1.00 | 0.94 | 0.97 | |

ignore terms having a document frequency lower than the given threshold.

**Experimental setup used**

To perform our experiments we used a *MSI PL62 7RC* with the following tech specs:

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- RAM: 8 GB
- Cores: 8
- Architecture:64-bits Ubuntu 18.04.4 desktop

We also used Google Colab to train and test our models. In particular, we used a VM with 25 GB of RAM.

### 4.5.4 Results and discussion

Table 4.20: Classification score for Support Vector Machine model

| Features | Loss | Target | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|---|
| Source code/ | Hinge | Not Ponzi | 0.99 | 0.99 | 0.99 | 0.98555 |
| Opcode | Hinge | Ponzi | 0.94 | 0.94 | 0.94 | |
| Source code/ | Squared Hinge | Not Ponzi | 0.99 | 0.99 | 0.99 | 0.98194 |
| Opcode | Squared Hinge | Ponzi | 0.92 | 0.93 | 0.94 | |
| Source code/ | Hinge | Not Ponzi | 0.98 | 1.00 | 0.99 | 0.98194 |
| Transactions | Hinge | Ponzi | 0.97 | 0.89 | 0.93 | |
| Source code/ | Squared Hinge | Not Ponzi | 0.99 | 1.00 | 0.99 | 0.98916 |
| Transactions | Squared Hinge | Ponzi | 0.97 | 0.94 | 0.96 | |

Table 4.18 resumes the results obtained with the Decision Tree model. We tested all the models with two different features. The first text feature joins the source code and the opcode, while the second one joins source code and transactions. Also, we tested our models with different parameters. For example, the *criterion* parameter allows to choose between the *Gini impurity* and *entropy* as functions to measure the quality of a split. Looking at results, we see that in terms of precision, recall, and f1-score, the decision tree behaved well with both the two different text features and with both Gini impurity and entropy. Choosing between the Gini impurity and the Entropy-based information gain doesn't make too much difference because they are pretty much the same. Anyway, selecting the Gini impurity would spare to compute logarithmic functions, which are computationally intensive. Anyway, we can see from the results that the best configuration is a Decision Tree trained with source code and transactions as a text feature. Table 4.19 shows results reached by a Multinomial Naive Bayes model. Again, the model behaved well and reached results not so far from Decision Trees's one, and sometimes slightly better. Again, the best configuration is a Multinomial Naive Bayes trained with source code and transactions. The parameter Fit Prior defines whether to learn class prior probabilities or not, but from the results, we can see that using a uniform prior or not doesn't make any difference. Table 4.20 shows results reached by a SVM model. The *loss* parameter allows to choose betweeen the *hinge* and *squared hinge* loss function. The SVM model behaved well both with source code and opcode and also with source code and transactions as text feature, despite the configuration trained with source code and transactions returns slightly better results.
Now that we showed all the results reached by all of the tested models, we can do further considerations. All the models have behaved well, and they were able to classify Ponzi schemes contracts. As we expected, the best models are the Multinomial Naive Bayes and the SVM, which obtained the best results in terms of precision, recall, and f1-score, reaching both an accuracy of 99pt%. The accuracy reached by our models is almost perfect as evidenced by the ROC curve plotted following a training of an SVM model shown in the Figure 4.23, but we must consider that

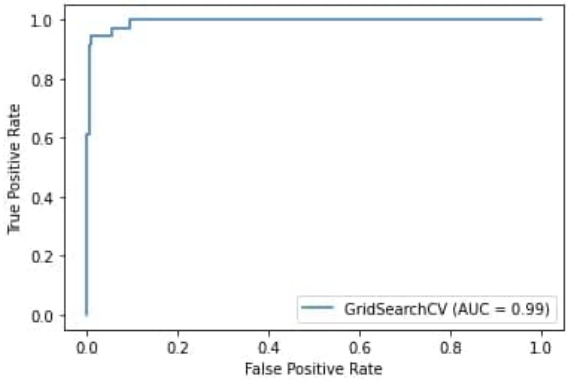



Figure 4.23: Example of ROC curve showing the True Positive and False Positive rate reached by our SVM model

this was also because Ponzi Schemes contracts have a well-defined structure, and many of them are similar in terms of code since a lot of Ethereum developers take inspiration from others' work.

### 4.5.5 Future Work

It must be said that despite all the work that has already been done, Ethereum remains a relatively recent platform, and as a result, many vulnerabilities and honeypots have yet to be identified (**?**). Given these premises, new scam schemes may be implemented by exploiting these vulnerabilities. All Ponzi scheme contracts used as dataset samples date back to 2016-2018, but new contracts may have been introduced in the meantime.We must consider that the Ponzi Schemes contracts analyzed despite are included in a period of three years, they show no differences substantial in implementation. Although there have been several pragma and consequently some changes in the definition of functions and constructs are required, the flow of execution of a Smart Ponzi remains almost unchanged compared to the types already presented in section 4.5.1.
The goal is to use our model to check whether new Ponzi schemes have actually been introduced into the Ethereum blockchain. The Ponzi scheme is just one of the possible scams that can be implemented against the participants of the Ehtereum network. In fact, other possibilities have been identified that allow taking advantage of the inner properties and possibilities offered by blockchain technology to implement scams. One possibility is to expand and make our model more comprehensive than the one presented in this study in such a way as not only to recognize Ponzi schemes but also contracts that are potentially highly damaging, in economic terms, to the participants who use it.

# Chapter 5

# Visualization-based models

## 5.1 Introduction

The previous two chapters have been dedicated to the development of tools to help the interaction of, respectively, expert and non-expert users with the blockchain. The tools proposed in the chapters can certainly help to spread the use of the blockchain among some users, but this might not be enough to introduce it to the whole society. Indeed, many academic studies suggest that the blockchain has implications for nearly all sectors of society such as governance (**?**), political economy (**?**), environmental protection (**?**), law (**?**), healthcare (**?**) just to name a few. Models and tools that facilitate the interoperability among experts in different sectors might be needed to realize all the blockchain implications as useful implementations in the form of practical proposals to solve societal problems.

In the field of blockchain there is a lack of models (**??**) and tools (**?**) that allow experts in different areas, such as experts in law, policy, finance, digital innovation, to cooperate and share knowledge among them. Different research areas own different technical jargons used to communicate scientific knowledge: indeed, while computer scientists need to understand program code, law experts are not expected to understand the program code. Graphic representation could be a means to facilitate the understanding of scientific knowledge among different disciplines (**?**). Figures and diagrams might indeed be a communication medium among different disciplines' languages and expertises. Figures and diagrams not only show the relevant data that support key research findings, but also provide visual information on the interactions among different operations required in scientific reasoning (**??**). Being able to adequately and precisely visualize data is also a pillar on which decisions can be made, as proposed by different dashboards in the market (**?**).

The chapter 5 is specifically dedicated to present a tool and a model based on graphical representation that are intended to facilitate the understanding of the blockchain mechanisms. Section 5.2 presents Smart-Graph (**?**), i.e., a tool able to provide graphical representations for smart contracts on the Ethereum blockchain.

The graphical representation provided by Smart-Graph highlights two relevant aspects for the blockchain software developers and companies: the economical costs required to deploy and execute a smart contract on the blockchain and the maintainability of the smart contract source code. Section 5.3 presents the analysis of two use case studies aimed to devise a comprehensive interdisciplinary model for graphical representation, which is then provided within Miró (**?**). Miró integrates a data-driven approach with an approach that guides users with different expertise in specific domains, to achieve the intended visualization, based on their aims, knowledge and hypotheses. The model can thus provide the users with different expertises a means to collaborate in the blockchain environment.

## 5.2 Smart-Graph

### 5.2.1 Introduction

The use of blockchains is currently explored in a number of scientific fields due to their potential to radically change the ways of economical exchange and the hows of traditional legal entities (**?**). For these reasons, in the Ethereum blockchain, where specific expertise in smart contracts' development is required, also other experts are required, i.e. experts in statistics, machine learning, economics, marketing and finance (**??**). In addition, the complexity of blockchain engineering hampers the understanding of the users, be them laypeople, experts in other fields and sometimes even software developers (**?**). Previous studies showed that graphical representations of data can help the understanding of a subject, making it a faster and easier task (**?**).

In this section, we review some visualization approaches and practical tools previously used to provide technical insights for experts and not-experts in the field of graphical representation of the static aspects of the software. After reviewing and discussing the main approaches, this study proposes a web-based tool to graphically represent smart contracts. Agile methods promote "working software over comprehensive documentation" (**??**). Recent research has shown that agile teams use quite a number of artefacts (**?**). One of these artefacts is Unified Modeling Language (UML) (**?**). UML is a modelling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system (**?**). UML offers a way to visualize a system's architectural blueprints in a diagram, including elements such as the individual components of the system; how the system runs; how entities interact with each other (components and interfaces) (**?**). The study aims to design and build a tool, "Smart Graph", to visualize the smart contracts entities and how they interact. Figure 5.1 shows an example of the graphical representation provided by the tool, "Smart Graph" (**?**).

In this research, we survey both two-dimensional space (2D) and three-dimensional space (3D) visualization techniques, representing the static aspects of the software. Finally, the study presents a web-based tool, "Smart Graph", to generate a 2D graphical representation of the smart contract's source code via a UML class diagram. The graphical representation provided by "Smart Graph" is an augmented version when compared to the previous traditional UML class diagram. Indeed, it allows giving information that is specific to Solidity programming language, such as "Function Modifer" and "Function Fallback" (**??**).

Furthermore, the study presents and discusses the general components of the web-based tool "Smart Graph". The advantage of web-based tools is that the Solidity developers do not need to install any software in their operating system. Solidity is a contract-oriented, high-level language for implementing smart contracts and is designed to target the Ethereum Virtual Machine (EVM) (**?**). The tool presented in this study aims to provide smart contracts programmers with a fast overall view of the smart contract structure and a useful insight into the source code they are

developing.

The rest of the section is structured as follows: Section 5.2.2 describes some work on the graphical representation of data software. Section 5.2.3 presents the research questions and hypothesis of this work. Section 5.2.4 presents the research methodology adopted for developing Smart-Graph. Section 5.2.5 presents the results. Section 5.2.6 presents the conclusion of this research.

## 5.2.2 Background

Smart-Graph aims to highlight two relevant aspects for the blockchain software developers and companies: The relevant aspects are the costs and the maintainability of the source code. These aspects depend on the following properties: on the storage location and on pure functions (and especially the balance between pure and impure functions).

### Data location in Solidity

A smart contract written in Solidity is a collection of source code's functions and data that are stored at a specific address on the Ethereum blockchain. The EVM can access and store information in six places: stack, memory, storage, code, callData and logs.

The table 5.1 reports some properties of the data location, such as the properties of volatile vs. persistent, and the operation that can be performed. For instance the stack memory is volatile. This means that, after the execution of the function, the data are lost. Also the memory location is a volatile memory, but in this case the data are lost after the execution of the contract. In fact, the memory location is used to share data among different functions. Finally, the log location is a persistent memory, such as the store, but differently from the storage, log location is cheaper. It is worth noticing that the current UML class diagram specification cannot provide all these pieces of information.

Table 5.1: Data Location in Solidity

| | Name | Volatile | Persistent | Read | Write | Description |
|---|---|---|---|---|---|---|
| 1 | stack | ✓ | | ✓ | ✓ | EVM Opcodes pop information from and push data onto the stack. |
| 2 | memory | ✓ | | ✓ | ✓ | Information store accessible during a transaction. |
| 3 | storage | | ✓ | ✓ | ✓ | Data in the storage are written in the blockchain (hence they change the state). |
| 4 | code | | ✓ | ✓ | ✓ | Executing code and static data storage. |
| 5 | callData | ✓ | | ✓ | | The data field of a transaction (external function). |
| 6 | logs | | ✓ | | ✓ | Write-only logger/event output. Can be accessed from outside the blockchain. |

One could wonder why the data location is so important. The yellow paper provides information on the cost of every "write and read" operation on a data

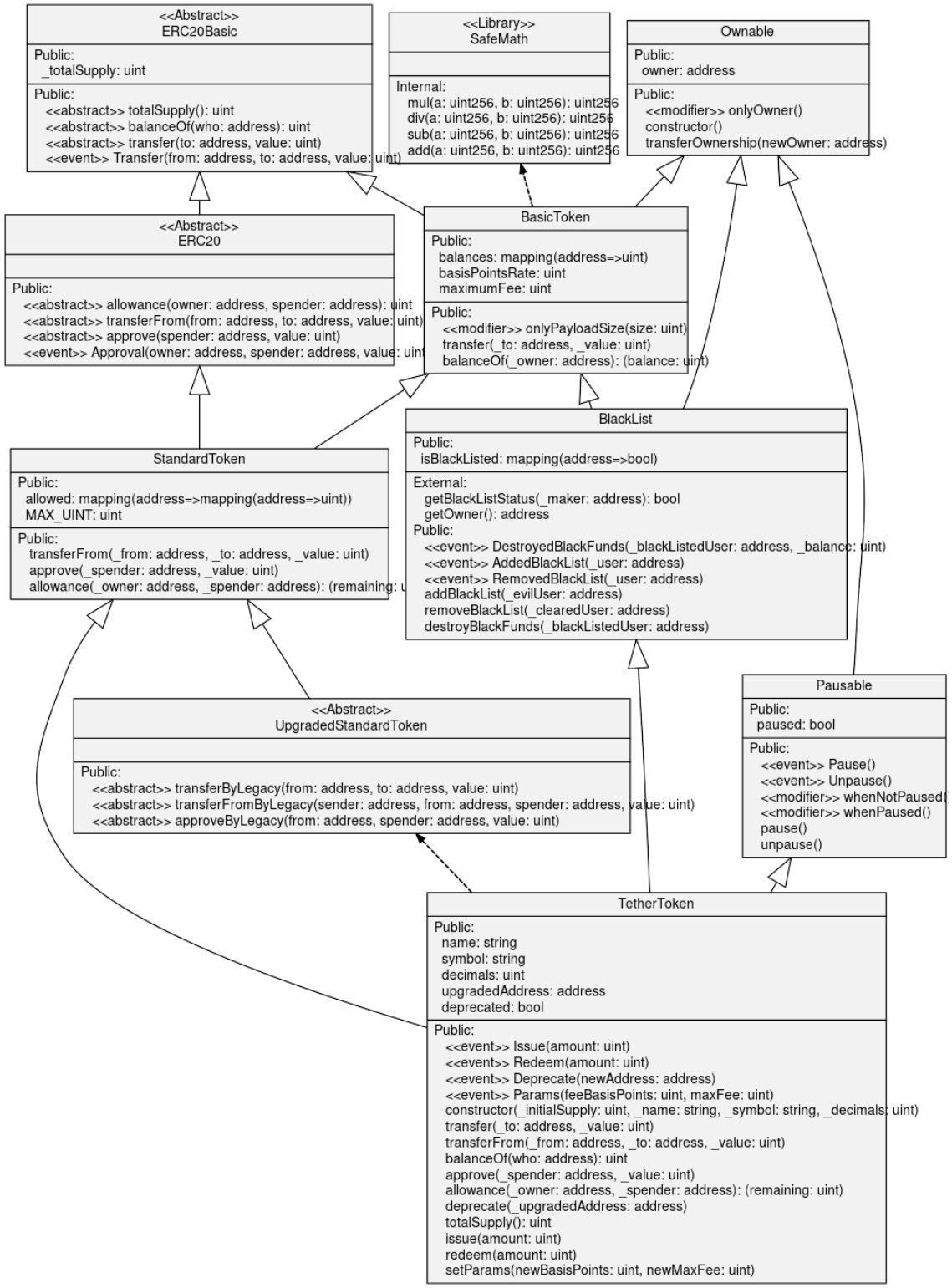



Figure 5.1: UML class diagrams of Tether smart contract.

location in terms of Gas(**?**). Table 5.2 reports the cost of the "write and read" operation in euro. The price is based on the data coming from the EtherGasStation and the price of the euro/ether pairs. The data reported in the table refer to the date of Friday, the fifth of March 2021.

Table 5.2 shows as the storage and the log memory location are both persistent but have different costs. For each operation, two values are reported, one value refers to the price to pay to execute the transaction as soon as possible and the other to the price to pay to execute the transaction in 30 minutes. Table 5.2 shows that the amount of Gas used during a transaction heavily depends on the smart contract data location, and thus also the price to pay to have the transaction executed. As a consequence, the best practice would be to write an optimized code that uses a minimum amount of Gas.

Table 5.2: Cost of the "write and read" operations

| **zone** | **EVM Operation** | **€/Word** | | **€/KB** | | **€/MB** | |
|---|---|---|---|---|---|---|---|
| | | **cheap** | **fast** | **cheap** | **fast** | **cheap** | **fast** |
| Stack | Read | - | - | - | - | 0.44 | 4.44 |
| | Write | - | - | - | 0.01 | 0.67 | 6.67 |
| Memory | Read | - | - | - | 0.01 | 0.67 | 6.67 |
| | Write | - | - | - | 0.02 | 14.89 | 148.88 |
| Storage | Read | - | 0.1 | 0.04 | 0.043 | 44.44 | 444.43 |
| | Write | 0.14 | 1.36 | 4.34 | 43.4 | 4444.32 | 44444.24 |
| Log | Read | - | - | - | - | - | - |
| | Write | - | - | - | - | 0.88 | 13.34 |

### Functions in Solidity 0.8.2

A function written in Solidity programming language consists of a function header and a function body. The function header is made up of six parts: the function declaration keyword, function name, function parameters, the visibility modifier, the behaviour modifier and the return type. In particular, next to the function "visibility" modifier, the function "behaviour" modifier is specified. The function "behaviour" modifier is specific of Solidity programming language and is not present in other programming languages, such as Java. Moreover, the current UML class diagram does not include this piece of information, and thus cannot provide us with information about the behaviour of a function.

Possible values for the function "behaviour" modifier are: view and pure. A view function is a function that has no side effects. This means that a function with such behaviour is not allowed to change the external world, such as the blockchain status. Another property of a view function is that its output can change over time. As view function can read values external to the function, these values can change over time and so also the output can change accordingly. Pure functions are functions close to functions in math. In fact, these functions have no side effects and are

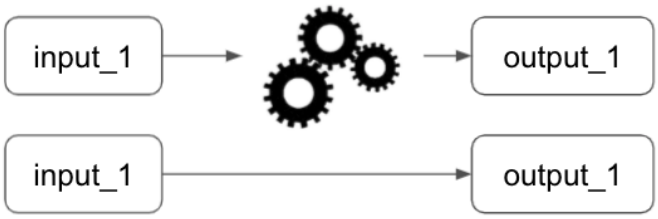



Figure 5.2: Cached Result vs Direct Call.

deterministic. This means that for a given input, a pure function gives the same output regardless the time.

There are several reasons to highlight pure behavoiur of a smart contract function through a graphic representation. Some of these reasons will be listed and discussed below.

- Caching. Highlighting the pure function behaviour is important for a software developer, because the result of a pure function can be cached. In fact, when a compiler detects a pure function, it can automatically optimise the execution by caching the result. These properties come from the fact that a pure function does not depend on time. So the compiler can map the input, which usually passed through a function, directly to a value, as the figure shows 5.2. An advantage of caching the result is that the execution is faster, as shown in this scatter plot 5.3.

- Testing. Another important aspect of pure functions is that they are easy to debug. This figure shows that a pure function depends only on the function's input parameters and not on the external scope. On the contrary, in the case of impure functions, a developer needs to look outside the function's scope to determine what the variable state is at the time the function is called.

- Parallel Computing. Pure functions are easy to run in parallel. Nowadays the CPU manufacturers improve the processor performance by increasing the number of cores. This means that having a source code made of many pure functions can enhance the performance of smart contract execution.

### 5.2.3 Research Questions and Hypothesis

The tool presented in the study has been designed to make the access easy for the users as it does not require any configuration and installation in the users' computers. The users can indeed access the tool from any web browser. Through the tool's web interface, the user can give a smart contract address as an input, and get a augmented UML class diagram of the smart contract address as an output.

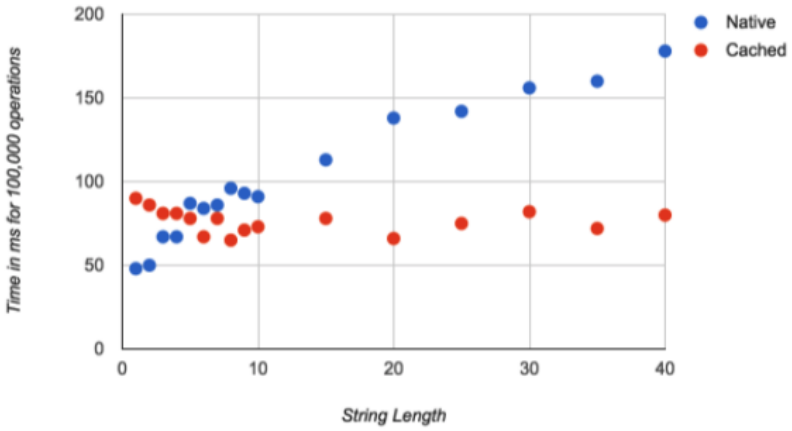



Figure 5.3: Cached Result vs Direct Call Scatter Plot.

The tool is available via a web browser and can be tested at the following link: https://aphd.github.io/smart-graph/. The backend produces the graphical representation of the smart contract, i.e. the augmented UML class diagrams.

The research aims to discuss the strengths and limitations of the UML class diagrams to visually represent the smart contracts' source code. The study considers and analyzes a source code coming from a DApp composed of many smart contracts, and it proposes a web-based tool to overcome the limitations of the UML class diagrams. The study aims to answer the following research questions:

- Q1 Are the UML Class diagrams used by the most popular OOP languages, such as C++ and Java, sufficient to visually represent the smart contracts' source code?
- Q2 Is there a better way to visually represent the smart contracts' source code?

To answer the research questions, the following hypotheses are proposed:

- H1 The current UML class diagrams used by some OOP languages, such as C++ and Java, are not sufficient to visually represent the information of a smart contract. For instance, in the current UML class diagrams' specification, there is no way to visually represent features of the source code that are unique in Solidity, and that could be very important for a smart contract developer, such as the "Receive Ether Function" and the "Fallback Function".
- H2 Given the complexity of some DApps, which are often composed of many smart contracts, a static representation of the source code is not apt to represent the most important information of the source code in an accessible way. An interactive environment for the smart contracts' developers can be useful to better represent the smart contracts' source code, for instance by allowing

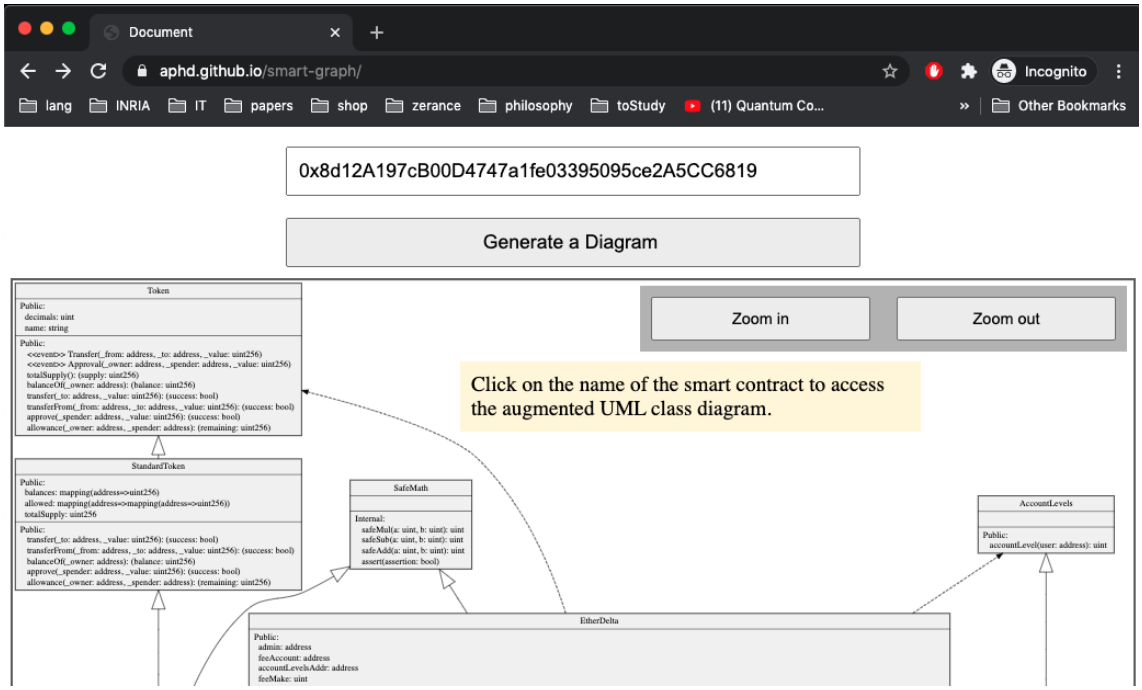



Figure 5.4: The "Smart Graph" GUI accessible through a web browser over the internet.

to visualize the smart contracts' source code through actions, such as drill down or zoom in/out in the diagram.

### 5.2.4 Research Methodology

#### Frontend

The front-end is the platform component visible and accessible to the users, stakeholders included.

The front-end was implemented in a web-based environment. This is an advantage, as the front-end component is platform-independent. This means that the users will only need a web browser to access the platform. Moreover, it will be possible to benefit from cutting-edge web technologies to produce graphical representations and visualize them.

Figure 5.4 shows the "Smart Graph" GUI accessible through a web browser. The GUI is divided into two sections: the "Smart Graph Form", on the top of the page, and the "Smart Graph Diagram Container", in the middle of the page. The "Smart Graph Form" has a text field as an input and a button named "Generate the Diagram" to get the output. The input text field allows the user to write a smart contract's address. When the user clicks on the button, the backend generates the corresponding UML class diagram, which is shown below the form. The "Smart

Graph Diagram Container" is the area where both the classical UML class diagrams and the augmented UML class diagrams are displayed.

In designing the front-end part, a challenge is representing DApps that are composed of many smart contracts. In these cases, the UML class diagrams are visually complex and prone to be information overloaded. To visualize many UML class diagrams, the "Smart Graph Diagram Container" is interactive, as the user can perform different actions within the class diagram. For instance, the GUI presents two buttons, which are accessible in the upper right part of the interface 5.4 and make it possible to zoom in or zoom out a certain part of the UML diagrams. The zoom-in action allows getting details about specific features of the smart contract, by selecting the most important parts of the visual representation and hiding the less relevant parts. Moreover, the user can see the augmented version of the UML diagram, by clicking on the smart contract's name displayed on the top of the UML class diagram, as shown in Figure 5.5.

Figure 5.5 shows the details about the method named "transferOwnership", which belongs to the smart contract "Ownable" displayed in Figure 5.1. The implementation of the "Ownable" smart contract made by the OpenZeppelin developers is available at the following link: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol`. The function "transferOwnership" allows transferring the owner's account, i.e. the account that deployed the smart contract, to a new owner.

Figure 5.6 shows the pipeline of the operations executed by the backend when a user requests the UML class diagrams by providing a specific smart contract address.

### Backend

The backend code is written in javascript because it supports synchronous operations like Promise. The Promise's feature is very important in the backend-part because it allows handling potential blocking operations, such as fetching resources from a server. An instance of potential blocking operation occurs when the user submits the request to the backend of the tool. When this happens, the backend makes a request to Etherscan to get the source code corresponding to the smart contract address specified by the user. Etherscan is an Ethereum block explorer which allows to explore and search the Ethereum blockchain for smart contracts source code (**?**).

The operation of fetching a smart contract source code from the network could be time-consuming, as it can take around 5 seconds. Therefore, we use a feature of JavaScript language named "Promise", in order to timely make the system responsive to other requests. Essentially, a Promise is an object that represents an intermediate state of an operation. There is no guarantee of the precise moment when the operation will be completed and the result will be returned, but there is the guarantee that, when the result is available or when the promise fails, the code provided to the Promise will be executed in order to do something else with a successful result or to gracefully handle a failure case. This is useful to set up a sequence

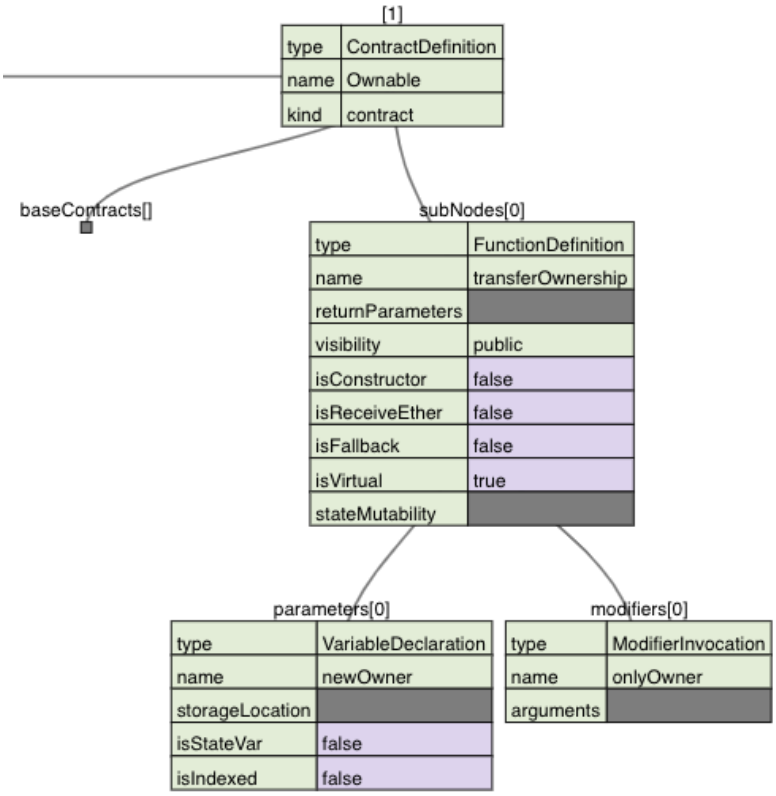



Figure 5.5: trasnferOwnership method visualized as a tree diagram.

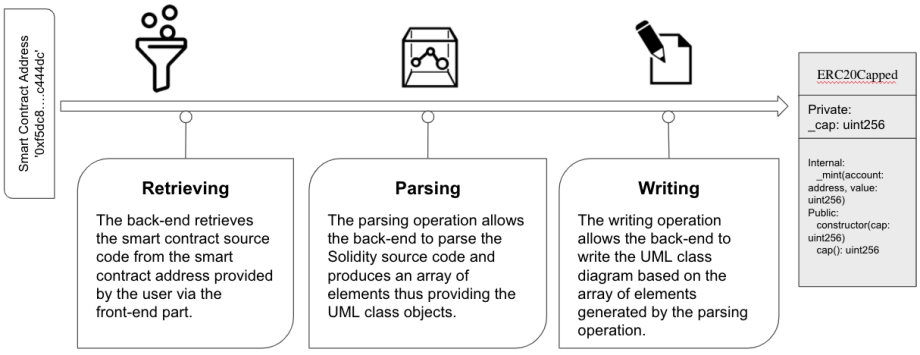



Figure 5.6: Pipeline of the operations executed by the backend when a user requests the UML class diagrams of a specified smart contract's address.

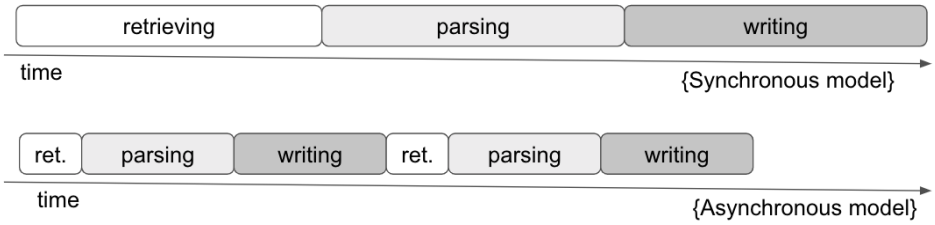



Figure 5.7: Synchronous Model vs Asynchronous Model.

of async operations to correctly work. Figure 5.7 shows the asynchronous operation that can occur during the generation of the augmented UML class diagrams.

The backend supports the following main operations: data retrieving 5.2.4, data parsing 5.2.4 and data writing 5.2.4.

### Data Retrieving

The retrieving operation allows the backend part to retrieve the smart contract source code. The retrieve method takes one argument, the path to the resource to fetch which is made from the following two parts: the Etherscan address and the smart contract address. For instance, a valid resource path is `https://etherscan.io/address/0x00000000219ab540356cbb839cbe05303d7705fa#code`. The retrieve method returns a Promise that resolves to the Response object to that request, whether it is successful or not. Once a Response object is retrieved, the HTML code contained in the Response object is processed to remove the HTML Tags to obtain just the Solidity source code of the smart-contract. The Solidity source code is processed by the backend Parser.

### Data Parsing

The parsing operation allows the back-end to parse the Solidity source code and produces a tree data structure in which each node stores an object of key-value pairs. For instance, the smart contract "renounceOwnership" method has the following nodes of key-value pairs.

```
{
  "type":"FunctionDefinition",
  "name":"renounceOwnership",
  "visibility":"public",
  "isConstructor":false,
  "isReceiveEther":false,
  "isFallback":false,
  "isVirtual":true,
  "stateMutability":null
}
```

### Data Writing

The writing operation allows the back-end to write two versions of UML class diagrams. The first version is similar to the UML class diagram of OOP languages, such as Java and C++. This version of diagram contains information such as the smart contract's attributes, the smart contract's functions, and the relationship type among different smart contracts. The second version is the augmented UML diagram. It displays additional information compared to the first-version diagram, such as information about the presence of "Function Modifer" and "Function Fallback" which are not part of the current unified modeling language specification.

Based on previous studies on graphical representations in different fields (**?**), multiple representations of the same smart contract source code have been preferred over the traditional UML class diagram. This technique allows to quickly visualize complex systems and guarantees better readability. Based on the users' actions, the backend will generate the corresponding augmented version of the UML class diagram.

## 5.2.5 Results and Discussion

Figure 5.1 displays the UML class diagram of a smart contract, following the current UML specification whose details are publicly available at (https://www.omg.org/spec/UML/About-UML/). There are two limitations of this representation type. First, it is very difficult to give a meaningful and comprehensive representation of the DApp in a screen, because is made of many smart contracts having different relationships among each other. Indeed, the UML diagram should display many classes, each one corresponding to different smart contracts. A recent study(**?**) shows that many DApps are made of a number of smart contracts greater than 10, which makes the graphic representation difficult. Moreover, the diagram generated via the UML specification hides some important details which are typical of smart contracts. Indeed, the Solidity programming language, used to write the smart contracts, owns specificities that no other OOP language owns. The current UML class diagram specification does not include specific features and constructs, such as "Receive Ether Function", "Function Modifer", "Function Fallback" and "Pure Function". These specific features could be very important for smart contracts' developers to have a quick insight on the smart contract's specific features. So, as to what concerns Q1, "Are the UML Class diagrams used by the most popular OOP languages sufficient to visually represent the smart contracts' source code?", the answer is negative.

A study regarding different methods to visually represent the source code has been investigated. Some of them involve metaphors, such as the metaphor of the city and the metaphor of the planet solar. This type of visualization could better fit the structure of DApps, because most of them are made of different smart contracts: a recent study(**?**) shows that 20% of the DApps deployed in the Ethereum blockchain

are made of 10 or more smart contracts. In the study an interactive visualization has been proposed to improve the existing web-based tools that provide a static representation of the smart contracts' source code. In this kind of existing tools, the developer must use the horizontal and vertical scroll bar to see the diagram from the left to the right and from up to down, and viceversa. Moreover, the user needs to check the source code to understand the details of the implementation.

As to what concerns Q2, "Is there a better way to visually represent the smart contracts' source code?", the study argued that the novelty of the tool "Smart Graph" is precisely the fact that it allows to visualize additional information, which was not provided by traditional UML class diagrams (**?**). Indeed, in "Smart Graph" the user can drill down into a specific function, to look for information specific to the Solidity programming language. For instance, Figure 5.5 shows a tree diagram displaying all information related to a particular smart contract's function. In this way the developer does not need to inspect the source code, but s/he can get all the information s/he needs just by interacting with the visual representation.

Based on the discussion about the different storage location and the properties of pure functions in Solidity, this is the solution proposed in this study. Smart-Graph is a tool that accepts as input a smart contract source code. It reads some external variables, such as the Gas Oracle and the Ether/Dollar pairs and then it creates three bullet graphs.

These three bullet graphs reflect the needs of different users:

- The user that wants to execute the transaction as soon as possible.
- The user that can wait 5 minutes.
- The user that has no urgency to execute the transaction and can wait longer.

Finally, the developer can drill down the report to check the main properties of the source code that affect the cost of the execution of the smart contract. The further properties that are shown are related to the number of functions, on their behaviour and the data location. Indeed, these properties have an important impact on the costs of transaction execution.

### 5.2.6 Conclusion

The research presented a tool, "Smart-Graph", that produces UML diagrams from source code in Solidity. "Smart-Graph" is a target-oriented tool for experts in the blockchain domain and it provides graphical representations which were already tested in other contexts, i.e. the UML diagram classes. However, when compared to previous UML diagrams, Smart-Graph provides an augmented version of the graphical representation. Indeed, "Smart-Graph" allows the user to inspect the class diagram to look for further details, such as "Fallback Function" or "Function Modifier", which are not accessible in the traditional UML class diagrams. Future

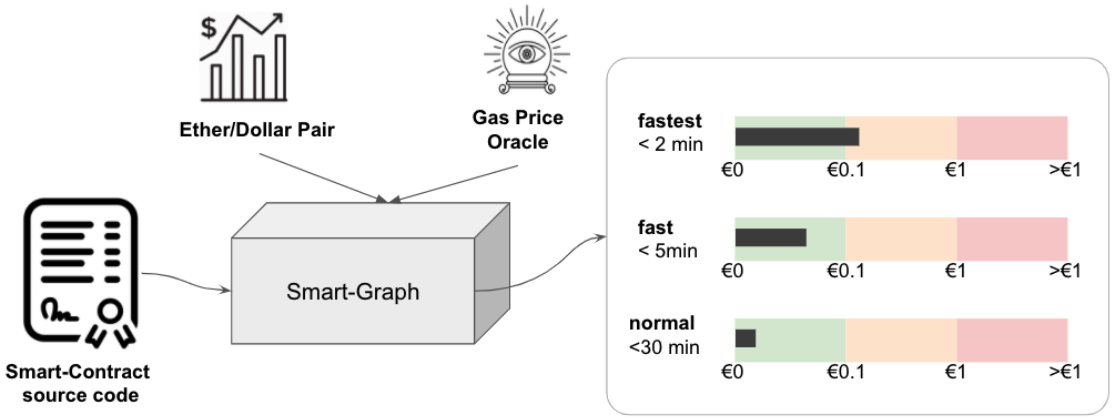



Figure 5.8: For a given Smart-Contract, Smart-Graph displays three bullet graphs. The graphs are produced based on the data taken from the external world (the current Ether/Dollar Pair and the Oracle Gas). The graph is time-dependent.

research should be dedicated to the development of the tool, which could also provide visual representations of smart contracts based on metaphors, thus also targeting people who are not experts in the software development domain.

## 5.3 An Interdisciplinary Model for Graphical Representation

### 5.3.1 Introduction

Graphical representations of data are fundamental for the understanding of scientific knowledge, as readers often rely on what the experts visually represent in their publications to understand the underlying data-set and interpret their potential scientific meaning (**?**). Figures and diagrams not only show the relevant data that support key research findings, but also provide visual information on the interactions among different operations required in scientific reasoning (**??**). Being able to adequately and precisely visualize data is also a pillar on which decisions can be made, as proposed by different dashboards in the market.

Data visualization has various purposes, such as to make abstract thinking on data series or sets more concrete and (mentally) manipulable, to help readers identify and evaluate some features of the data, to let users see the possible underlying trends, patterns, processes, mechanisms, etc. of the phenomena considered and studied (**?**). The way data are visualized can therefore have important epistemic implications for scientific knowledge, as data visualization is not an "interpretation-free" practice, i.e. a neutral process of data presentation in terms of scientific understanding. There are indeed several ways to transform data into a visual format, each of them entailing

different possibilities for data interpretation.

Nowadays data visualization plays a significant role in the large adoption of data-driven and machine learning approaches and techniques. In this frame, the definition of what a visualization is can be object of debate. A visualization could be defined as a reusable component, which is achieved through a dedicated software library. For instance, some software for data visualization are MATLAB and Mathematica. Despite the large amount of tools offered by these software, surprisingly, it is left to the practitioner to actually manipulate the data to achieve a ready-to-be-used graphical representation. Previous research proposed data-driven models that exploit existing software libraries or adopt a framework-agnostic approach (D. A. Keim, 2002 (**?**)) based on data types to be visualized.

The work aims at designing a framework for a software, named Miró, which instead allows the users to produce meaningful graphical representation in an automatic way without the need to manually transform the data. First of all, we aim to verify the benefits and the shortcomings of existing data-driven and problem-driven models, by presenting some case studies. The case studies focus on the problem of visually representing specific data-sets collected in different scientific domains for different (descriptive vs. prescriptive) scientific aims. The case studies suggest that data-driven models can actually provide a visualization that fits the domain knowledge and scientific aims of the experts in the case of descriptive sciences, but present some limitations in the case of prescriptive sciences. Finally, the research draws some conclusion from the case studies, presenting an alternative interdisciplinary perspective for data visualization. A comprehensive model for graphical representation is then presented, which integrates a data-driven approach with an approach that guides the experts on a specific domain field to achieve the intended visualization, based on their aims, knowledge and hypotheses. Miró adopts this interdisciplinary perspective and is based on a visualization engine developed in Pharo and named Roassal (**?**).

### 5.3.2 Data-driven and Problem-driven Models

In the field of data visualization computing, researchers proposed different approaches to a comprehensive data-model, i.e. a model able to provide a meaningful graphical representation of a data-set for some scientific aims. Some authors advocated graphical representation techniques or visualization frameworks (**?**) based on data-driven models. The data-driven model approach is based on the idea that a comprehensive data-model is based on a prior data classification that can guide the automatic creation of a meaningful graphical representation. In general, the data-driven model describes the data characteristics of the data-set, such as the size (the number of rows), the data type (string, number, boolean) and the dimension (the number of the variables to represent), to categorize the data. Keim (**?**) proposed a data-driven visualization model based on the data types to be visualized, the visualization technique and the technique of visual interaction with data, ranging from

standard and projection to distortion and "link&brush".

Other authors, especially in the context of big data visualization, proposed graphical representation techniques based on a problem-driven model (**?**). The problem-driven model provides the researchers with the possibility to perform specific tasks on specific variables of the data-set, such as visualizing a variable distribution, performing a linear regression between two variables to see an eventual relationship via a scatter plot, comparing their composition via a pie chart, etc.

On the one hand, adopting a problem-driven model does not necessarily mean abandoning data-driven models. The problem-driven model may be tightly linked to the data-driven model, because the data-driven model imposes constraints on the graphical representation of data which might conditioning how the problem can be solved. For instance, in the case of time series, there are graphs that are less appropriate than others or that are simply wrong depending on the data classification: the time data-type is indeed a constraint given or inferred from the data-driven model. On the other hand, a graphical representation that is guided only by a data-driven model would not allow the users to further act on data to have their final intended graphical representation. In the software where a problem-driven model is also envisaged, the user can interfere with the final graphical representation of the data. The user can indeed act on and guide the graphical representation to be produced.

e main disadvantage of the problem-driven model is that it might be negatively influenced by the users' previous hypotheses or scientific aims. On the contrary, a data-driven model is neutral under this respect: of course it is based on a prior classification, but the users might not know it. Without the users' interference, the final graphical output of a data-driven model might indeed have the advantage of questioning the researchers' prior goals and solicit a belief revision. Especially when a graphical output is unexpected and not corresponding to previous scientific goals, it might bring about further research or action.

Both the models assume that the data-set contains the information useful to produce a meaningful graphic representation. This may not always be the case. Scientific studies based on data-sets make use of graphical representations to better interpret their results. Among these studies, it is possible to find descriptive as well as prescriptive studies. The former aim to describe phenomena as they are, observing, recording, classifying, and comparing them (**?**). The latter aim to provide the conditions for how phenomena should be, thus supporting inferences for data interpretation and decision and/or action to perform on data. Of course, a scientific study could be both descriptive and prescriptive, also depending on the scientific goals. The development of new decision-aiding technology should be tailored for both (**?**), also in the case of graphical representation (**?**). The study is therefore driven by the question on how a model should be to provide a meaningful graphical representation of a data-set to support the inferences and/or the decision a researcher wants to draw, in both the case of descriptive and prescriptive scientific studies.

In the study we propose a general distinction between a model for descriptive studies and a model for prescriptive studies. Within these two models, it is possi-

ble to specify sub-models, specific for scientific domain and particular data types involved in the study (**?**). Both the models can be used whenever a study has both descriptive and prescriptive scientific aims, as it is often the case.

### 5.3.3 Research Questions and Hypotheses

The study aims to discuss the strengths and limitations of existing models for data visualization, by considering and discussing some case studies coming from publications of different scientific domains and having different scientific aims.

The research addresses the following questions: Q1) Are data-driven models sufficient for a software to help the researchers to automatically create the intended visual form for a data-set? Q2) In the case the data-driven models are not sufficient, what could be the best way to overcome their limitations? Q3) Can the existing libraries or programs fit a data-driven model perspective and at the same time overcome their shortcomings?

To answer the research questions, we advanced the following hypotheses: H1) The data-driven models might support the creation of meaningful graphical representation only for some specific scientific aims, such as the researchers' aims to provide a descriptive data analysis. H2) For scientific aims going beyond descriptive analysis, the existing data-driven models might not be sufficient. The data-driven models might need to be integrated into a more comprehensive and interdisciplinary data-model to overcome their eventual limitations. H3) Existing software libraries are data-driven and might not be sufficient to help researchers to find the intended visual form for prescriptive scientific aims. They might need further implementation to allow the users to perform different manipulation on data, such as transformation, accommodation and integration with complementary data, to achieve the intended graphical output.

Several different real-world scenarios and case studies support the hypotheses mentioned above (**??**), a couple of which are discussed in the following Section 5.3.4.

### 5.3.4 Case Studies Evaluation

We analyzed data-sets which are representative of two different scientific approaches: 1) descriptive and 2) prescriptive studies. In particular we provide a detailed analysis of some case studies, coming from 1) the domain of software metrics, in the wider field of AI, and 2) the field of human mobility and sustainable development. The analysis can be extended to further case studies in different scientific domains.

#### Descriptive Case Studies

As to descriptive scientific studies, we considered first of all the case of a study on the performance evaluation of different frameworks in AI (**?**). The case study proposes a set of meaningful visual representations of a benchmark data-set for the performance

evaluation of different Deep Learning (DL) models and frameworks. The Authors calculated the accuracy and the throughput of five classification problems for the DL models and frameworks. The output data-set was made of a series of two categorical data (the name of the framework and the DL model) and two physical data.

We selected this study for three reasons: 1) The work aims to provide a significant graphical representation of the performance metrics of different frameworks; 2) The work also aims to extend the graphical representation to other frameworks, to be applied to other works and thus be generalized. 3) The study's data-set presents a number of variables and categories, which are not trivial to represent as a whole to obtain a meaningful graphical representation (**?**).

When analyzing the study case, we found that there is a data-driven model, specifically Keim's data-model, that provides us with a significant representation of the data-set, without any accommodation and/or transformation of the data and, more importantly, without any addition of further information by the user. Indeed, by applying Keim's data-model, the data-set is well within multi-dimensional category and so the meaningful graphical representation technique should be a "heat-map graph", where the colour is represented by the categorical data and the two physical data (accuracy and throughput) are represented in a 2D coordinate system. Therefore, as to what concerns Q1, "Do data-driven models support the creation of meaningful graphical representation", the answer is positive. As the Keim's data-model is sufficient to have a proper graphical representation, we do not need to cope with Q2 on how to improve it for this specific case study. As to what concern Q3, the existing libraries for producing data visualizations alone cannot give that expected output, even though based on a data-driven model. However, throughout a data-driven model such as the Keim's model and some accommodation of the data, the existing libraries could provide the expected automatic visual representation, starting from the raw data-set.

Other descriptive case studies concern, for instance, static programming analysis and focus on the correlation between numerical variables, such as the number of lines of code, cohesion, coupling or cyclomatic complexity (**?**) and categorical variables, such as the name of the package included in the analyzed software. This type of studies' authors often choose to represent their data-sets via a bar graph where the bar length represents the numerical value and the categorical variable is represented by the different color of the bar or by a label. Also in these cases, the graphical output can thus be provided by a data-driven model such as Keim's model. The analysis can be extended to other descriptive case studies in different disciplines (e.g. biology (**?**), and sociology (**?**)), where Keim's data-model is sufficient to provide the categorization for descriptive scientific aims.

### Prescriptive Case Studies

In the case of prescriptive scientific studies we first considered an interdisciplinary study on human mobility (**?**). The Authors collected the data using smartphones

and smartwatches worn by several participants over 2 weeks. Through these devices, they collected three kinds of data: 1) motion sensor data, 2) physiological data, 3) environmental data. For the purposes of this case study, we are interested in the second data-set collecting information about electrocardiographic (ECG) data, such as heart beat and blood pressure. The data-set has the following characteristics: 1) data are multidimensional, as each row of the data set contains both spatial coordinates (longitude and latitude) and physiological data (heart rate, in beats per minute), provided by the optical heart rate sensor of the smartwatch; 2) the row data series consists of over 1 millions of data.

One of the purposes of the research study was to use physiological data to infer the user's stress and emotion level to identify places within a University campus area that are perceived as dangerous by the majority of participants. We selected this research for the following reasons:

- The research covers different domains: mobile computing, sensing systems, human mobility profiling and cardiology.
- As in the previous case study, the data-set presents a number of variables and categories, which are not trivial to represent in an overall meaningful graphic representation.

If we apply the Keim's model to the data-set, the graphic representation output is a "heat-map chart", where the position is represented in a 2D-coordinate system and the heart rate beat is represented by color hue. This type of representation may not be enough meaningful for the aims of the study, when based only on the data-set collected by the devices. Indeed, the data-set is not per se sufficient to have a meaningful representation: the danger zones' classification needs other, additional data, such as the normal resting heart rate range and the dangerous heart rate range, to be properly represented.

Figure 5.9 shows the graphical representation produced considering the additional data, the normal and dangerous heart rate ranges. These additional data are used to represent the different zones on the map with colors having different opacity (color with opacity 1 for the dangerous zones and transparent color for the zones considered safe).

Therefore, as to what concerns Q1, the answer is that the data-driven model is not sufficient to give the intended graphic representation. Indeed the authors considered complementary data that are not merely added to the existing categories considered by the data-driven model, but rather organize in a higher-order structure and provide the cues to interpret the data-set to have a meaningful representation of the zones considered dangerous. The complementary data do shape the authors' interpretation of the data-set as they provide some intervals (the heartbeat rates intervals), as conditions to classify dangerous vs. safety zones. Indeed, the graphical representation 5.9 can be prescriptively used by experts in urban development for strategic planning to improve safety in public places.

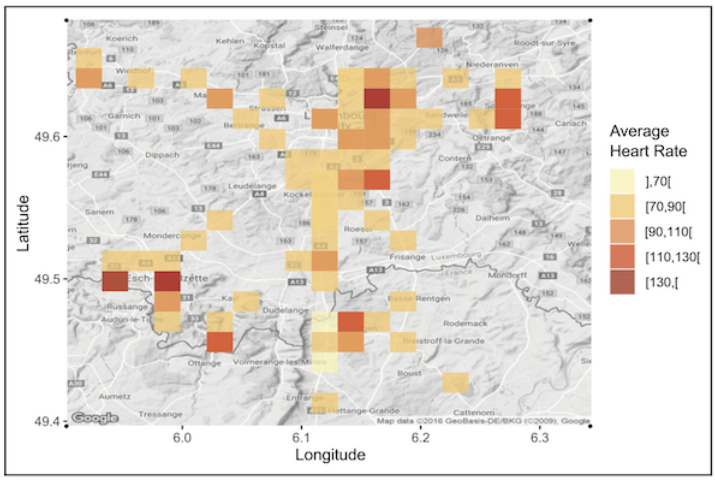



Figure 5.9: Places that are perceived as dangerous by the majority of users through the use of colors with different shades.

As to Q2, the solution to overcome the limitations of the data-driven model could be the possibility of inserting further data types into the data-set, relating the average heartbeat rates stored in the original data-set with the heartbeat rates intervals considered normal and dangerous. Furthermore, the data must be re-sampled taking into account the new knowledge, the normal resting heart rate range, coming from a different domain, the cardiology. However, this solution requires specific knowledge from the cardiology domain which may be different from the researchers' knowledge performing the data analysis.

Finally, regarding Q3, data visualization libraries alone cannot help to obtain the expected output. Indeed, different tasks should be foreseen to achieve the intended outcome through a software, including the data visualization libraries:

- the program should make use of a data-driven model, such as the Keim's model.
- the program should give the user the possibility to add other data type. In the prescriptive case study, the data-type are intervals (conditioning the interpretation of the other data), also coming from a different scientific domain, i.e. cardiology.
- the program should give the researchers the possibility to further categorize the data-set via the additional knowledge. The program must provide the data-set with an higher-order structure to achieve the graphic representation meaningfully corresponding to the authors' scientific aims.
- Once adopting this workflow, the program might use the data visualization library to generate the intended graphic representation.

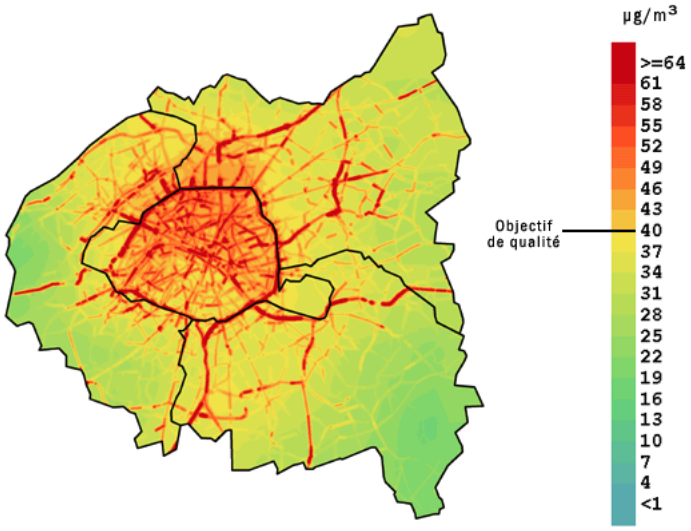



Figure 5.10: Areas affected by air pollution.

Another example of prescriptive studies concern the correlation between air pollution and respiratory illnesses (**?**). The research findings come from data belonging to different domains such as 1) prescriptive data conditions in health information systems, 2) the air quality index (AQI) data provided by the World Health Organization (WHO), and 3) the descriptive data coming from particular air pollution electrical sensors. The descriptive data alone, in particular the concentration of microscopic particles with a diameter of 2.5 $\mu$m or less, are not sufficient to produce a graphical representation apt to meet the prescriptive aims of the study (see Figure 5.10), i.e the sustainable development program in urban and rural areas affected by air pollution.

### 5.3.5 An Interdisciplinary Model

In the field of graphical representation, interdisciplinary models have been proposed to cope with the limitations of both previous data-driven and problem-driven models. For instance, Hall et al. (**?**) proposed a trans-disciplinary model which allow the experts in a particular domain to be supported by visualization experts. Their work is very interesting as the interaction between experts with skills in different domains could greatly influence the production of meaningful graphical representations to display cues for scientific findings.

However, the prescriptive case study examined in this research cannot be solved through this trans-disciplinary approach. Of course a competence in visualization is welcome, but cannot per se highlight the conditions of meaningfulness, which come from another scientific domain in the prescriptive case studies. Therefore an interdisciplinary model is needed which integrates knowledge and practice coming from different scientific domains in the process of visualization. Figure 5.11 proposes

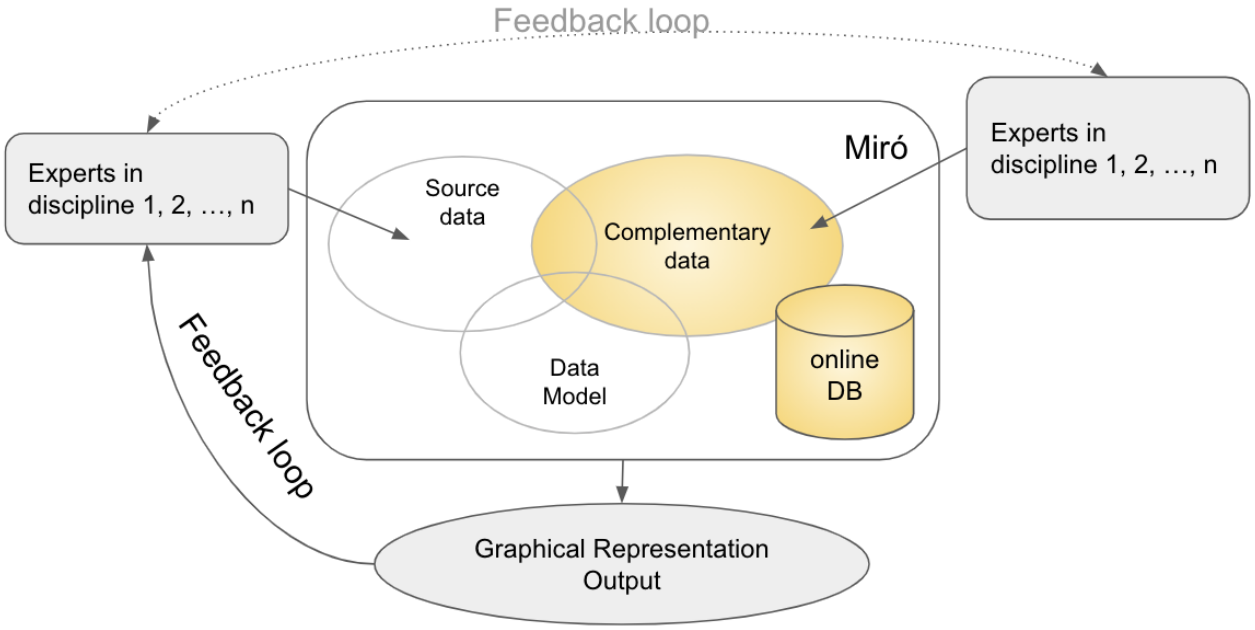



Figure 5.11: Interdisciplinary Model

the main elements of the interdisciplinary model.

- The source domain/s is/are the domain/s from which the data are collected.
- The complementary domain/s is/are the domain/s from where to collect the data required to interpret the source domain/s data.
- The blended domain (**?**) is given by the intersection between the source domain/s and the complementary domain/s, where some new insight could emerge.
- The data model is the model driving the software in the process of data categorization and visualization.

As a solution to the prescriptive case studies examined, we propose an interdisciplinary problem-driven approach for the visualization of data coming from different domains. For the aims of descriptive studies, the source domain and the data-driven model are usually sufficient to have meaningful graphical representations. The prescriptive case studies instead show the limits of both data-driven and problem-driven model, as there are scientific aims for which it is not sufficient having both the data models and the data coming from a scientific domain to obtain meaningful graphic reports for the research findings.

In prescriptive studies, two further processes - not envisaged in previous data-driven and problem-driven models - are needed to have meaningful graphical representations of the source data:

- A selection process: when the data collected by the researchers in the source domain are not sufficient, other specific data selected from a different scientific domains might be needed to interpret the source data. These data might indeed be the condition of meaningfulness for data interpretation, and thus for the visual output of the software.

- A transformation process: specific tasks might be needed for the re-interpretation of the data in light of the selected complementary data and the scientific aims of the study. For instance, the source data might need to be re-sampled considering the complementary knowledge.

The scientist's insight needs, therefore, to be entered as complementary data in any software's visual framework, which in turn should make it possible to enter them, interacting with the scientist. In the prescriptive case studies, the interdisciplinary approach is driven by the interaction among experts in different domains (mobile computing and cardiology) and guides the production of graphical representations, meaningfully representing the areas perceived as dangerous (see Figure 5.9). The insertion of the relevant complementary data might come not only from experts of another domain, but also from online interactions among experts in different domains and/or online web-based crowd-sourcing selected by the expert users themselves.

This interdisciplinary model might then overcome the limitations of both the data-driven and the problem-drive models, especially when it automatically proposes the complementary data based on the scientific aims of the expert and the relative missing expertise, which could come from an expert in another domain. This approach is the framework for Miró, a software intended to be a guide to build meaningful graphical representations for both descriptive and prescriptive studies, based on a data-set coming from the source domain/s and on a data model eventually able to provide complementary online data. Differently from softwares based on previous data-driven and/or problem-driven models, the Miró's interdisciplinary model allows the user to insert data or select data coming from complementary domain/s, and transform the source data-set to have the intended graphical representation.

In the case study requiring data from both human mobility and cardiology, when the participants to the experiment are considered as a group, their information provides other meaningful cues to identify critical geographical or temporal points. For example, the two figures coming from the prescriptive case studies represent respectively 1) the places that are implicitly perceived as dangerous or risky by most users and 2) the most polluted areas of a city. By analyzing the data-sets and their graphical representations, it emerges that there are data (fields) that make sense only within one or more interval/s [a, b]. Often, the interval information is neither provided within the data-set nor within the single scientific discipline and thus the interval must be set by the scientist and/or by another expert. This needs to be contemplated by the dashboard developer. For instance, in study 1), the heart-rate belongs to the health domain and make the place dangerousness meaningful only if the average value is above a certain threshold. The threshold needs to be provided

by a scientist (also following the scientific practices of his/her scientific field), it is not provided by the data-set per sé, especially in interdisciplinary prescriptive studies like 1).

Some data actually come from the data-sets, some other data come from the scientist's interpretation of the data in light of the scientific hypotheses in her/his study. The latter should be provided by the scientist and a dashboard should make it possible to enter them. Prescriptive scientific studies are more likely to need interval information as a condition of meaningfulness to make sense of the data-sets when compared to descriptive scientific studies, which can instead provide meaningful graphical representations based on traditional models. Of course, scientific studies can be both descriptive and prescriptive: Miró can provide a meaningful graphical representation also for these studies as it does not abandon traditional models, but it instead proposes further functionalities.

### 5.3.6 Conclusion and Future Works

This research shows how important might be an interdisciplinary data model, especially in prescriptive studies, to have a software able to provide meaningful graphical representations of data. In the case of descriptive studies, existing models - data-driven models and/or problem-driven models - might be sufficient to produce meaningful graphical representations when providing the data coming from the source domain/s. In the case of prescriptive studies, the existing models might fail to produce meaningful graphical representations when just the collected data coming from the source domain/s are provided. The research proposed an interdisciplinary approach to overcome the limitations of the existing models via a software-expert interaction. In this framework, the software allows the users to reinterpret and transform the collected source data in the light of the scientific knowledge coming from (online) interaction with other experts or data-sets coming from complementary domain/s. The graphical representation is made meaningful in the blended domain, thus providing a visual support for new findings.

# Chapter 6

# Conclusion

Blockchain technology has been around for more than ten years and countless papers on interesting applications of blockchain technology have been written and published in scientific journals (????). Many experts mention the potential applications of the blockchain for the aims of industry, market, agency, or governmental organizations (????). There are indeed many scientific works that explain the advantages and benefits for our society in adopting this technology in different areas: food traceability analysis (???), healthcare records management (?), identity management system (?), intellectual property protection (?), verifiable electronic voting (???). Blockchain can thus be an innovative and a revolutionary solution in many sectors of our society and it can improve our lives by making some everyday tasks more efficient and transparent (?).

Despite the countless scenarios where the blockchain can be applied to improve our quality of life, there are only few examples where this technology has been applied in a real-world scenario (?). This work embraces the idea that the blockchain technology still needs several improvements in different areas before becoming popular and adopted by many companies, both private and public. According to previous research, some areas for improvement are the costs associated with the blockchain development (??). Moreover, it has been suggested that the interaction with this new technology may still be difficult (?). To address these problems, different models and tools to facilitate the interaction with this technology have been proposed in the dissertation.

First, the dissertation proposed two applications aimed at expert users, namely PASO (?) and Smart-Corpus (?). Based on the latter, the dissertation reported the results of a corpus analysis on the source code clone practice in smart contracts (?). Second, the dissertation provided some insights into the design and the development of tools for non-expert users to overcome the barriers that can limit the access to a wider public. These user-centered tools can facilitate the interaction with the blockchain and the understanding of the possibilities that this technology can open in everyday life (??). Finally, the dissertation proposed an interdisciplinary model and tool, namely Miró (?), which allows experts in different areas, such as experts

in law, policy, finance, digital innovation, to cooperate and share their knowledge in the field of blockchain technology.

The tools and models proposed in the dissertation might be crucial in enabling the widespread adoption and development of blockchain technologies for academic and private users. Indeed, the purpose of the work was to develop user-centered tools with the aim of ensuring that blockchain is widely available through public and open-source code libraries. Some of the tools presented in the dissertation, based on the academic and non-academic citations on their use, are already helping to ensure that the full blockchain potential is reached and that further developments can be made to make the blockchain technology diffused, used and adopted within the society.

# Contents

# List of Figures

# Bibliography


AA.VV. (2020). Oracle-gas-price-source-code, may 2020.

Abras, C., Maloney-Krichmar, D., Preece, J., et al. (2004). User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, 37(4):445–456.

Abu-Elezz, I., Hassan, A., Nazeemudeen, A., Househ, M., and Abd-Alrazaq, A. (2020). The benefits and threats of blockchain technology in healthcare: A scoping review. *International Journal of Medical Informatics*, page 104246.

Adhami, S., Giudici, G., and Martinazzi, S. (2018). Why do businesses go crypto? An empirical analysis of initial coin offerings. *Journal of Economics and Business*, 100(C):64–75.

Adler, J., Berryhill, R., Veneris, A., Poulos, Z., Veira, N., and Kastania, A. (2018). Astraea: A decentralized blockchain oracle. In *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*, pages 1145–1152. IEEE.

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

Al-Jaroodi, J. and Mohamed, N. (2019). Blockchain in industries: A survey. *IEEE Access*, 7:36500–36515.

Alimadadi, S., Mesbah, A., and Pattabiraman, K. (2016). Understanding asynchronous interactions in full-stack javascript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180. IEEE.

Altman, E., Menasché, D., Reiffers-Masson, A., Datar, M., Dhamal, S., Touati, C., and El-Azouzi, R. (2020). Blockchain competition between miners: A game theoretic perspective. *Frontiers in Blockchain*, 2:26.

Amani, S., Bégel, M., Bortin, M., and Staples, M. (2018). Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 66–77, New York, NY, USA. Association for Computing Machinery.

Andoni, M., Robu, V., Flynn, D., Abram, S., Geach, D., Jenkins, D., McCallum, P., and Peacock, A. (2019). Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 100:143–174.

Anjum, A., Sporny, M., and Sill, A. (2017). Blockchain standards for compliance and trust. *IEEE Cloud Computing*, 4(4):84–90.

Antonucci, F., Figorilli, S., Costa, C., Pallottino, F., Raso, L., and Menesatti, P. (2019). A review on blockchain applications in the agri-food sector. *Journal of the Science of Food and Agriculture*, 99(14):6129–6138.

Antunes, N. and Vieira, M. (2009). Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306. IEEE.

Ayed, A. B. (2017). A conceptual secure blockchain-based electronic voting system. *International Journal of Network Security & Its Applications*, 9(3):1–9.

Azaria, A., Ekblaw, A., Vieira, T., and Lippman, A. (2016). Medrec: Using blockchain for medical data access and permission management. In *2016 2nd International Conference on Open and Big Data (OBD)*, pages 25–30. IEEE.

Bach, L. M., Mihaljevic, B., and Zagar, M. (2018). Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE.

Backus, J. (1978). Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Baclawski, K., Kokar, M. K., Kogut, P. A., Hart, L., Smith, J., Letkowski, J., and Emery, P. (2002). Extending the unified modeling language for ontology development. *Software and Systems Modeling*, 1(2):142–156.

Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, page 86, USA. IEEE Computer Society.

Baker Effendi, S., van der Merwe, B., and Balke, W.-T. (2020). Suitability of graph database technology for the analysis of spatio-temporal data. *Future Internet*, 12(5):78.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., and Kontogiannis, K. (2000). Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107.

Balzer, M., Deussen, O., and Lewerentz, C. (2005). Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 165–172, New York, NY, USA. ACM.

Baralla, G., Ibba, S., Marchesi, M., Tonelli, R., and Missineo, S. (2018). A blockchain based system to ensure transparency and reliability in food supply chain. In *European conference on parallel processing*, pages 379–391. Springer.

Baralla, G., Pinna, A., Tonelli, R., Marchesi, M., and Ibba, S. (2021). Ensuring transparency and traceability of food local products: A blockchain application to a smart tourism region. *Concurrency and Computation: Practice and Experience*, 33(1):e5857.

Bartoletti, M., Carta, S., Cimoli, T., and Saia, R. (2020). Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277.

Bechtel, W. and Abrahamsen, A. (2005). Explanation: a mechanist alternative. *Studies in history and philosophy of biological and biomedical sciences*, 36(2):421—441.

Bellare, M. and Rogaway, P. (1993). Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73.

Bentov, I., Lee, C., Mizrahi, A., and Rosenfeld, M. (2014). Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review*, 42(3):34–37.

Berdik, D., Otoum, S., Schmidt, N., Porter, D., and Jararweh, Y. (2021). A survey on blockchain for information systems management and security. *Information Processing & Management*, 58(1):102397.

Bergel, A., Maass, S., Ducasse, S., and Girba, T. (2014). A domain-specific language for visualizing software dependencies as a graph. In *2014 Second IEEE Working Conference on Software Visualization*, pages 45–49.

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. (1994). The world-wide web. *Communications of the ACM*, 37(8):76–82.

Bistarelli, S., Mazzante, G., Micheletti, M., Mostarda, L., and Tiezzi, F. (2020). Analysis of ethereum smart contracts and opcodes. In Barolli, L., Takizawa, M., Xhafa, F., and Enokido, T., editors, *Advanced Information Networking and Applications*, pages 546–558, Cham. Springer International Publishing.

Bracciali, A., Chatzigiannakis, I., Vitaletti, A., and Zecchini, M. (2019). Citizens vote to act: Smart contracts for the management of water resources in smart cities. In *2019 First International Conference on Societal Automation (SA)*, pages 1–8. IEEE.

Bragagnolo, S., Rocha, H., Denker, M., and Ducasse, S. (2018). Ethereum query language. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, page 1–8, New York, NY, USA. Association for Computing Machinery.

Brant, J. and Roberts, D. (2009). The smacc transformation engine: How to convert your entire code base into a different programming language. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 809–810, New York, NY, USA. ACM.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Breu, R., Hinkel, U., Hofmann, C., Klein, C., Paech, B., Rumpe, B., and Thurner, V. (1997). Towards a formalization of the unified modeling language. In *European Conference on Object-Oriented Programming*, pages 344–366. Springer.

Brown, R. and Vári, A. (1992). Towards a research agenda for prescriptive decision science: The normative tempered by the descriptive. *Acta Psychologica*, 1-3:33–48.

Budish, E. (2018). The economic limits of bitcoin and the blockchain. Technical report, National Bureau of Economic Research.

Buterin, V. (2013). Ethereum: a next generation smart contract and decentralized application platform. In *White paper*.

Buterin, V. (2014). A next generation smart contract & decentralized application platform. *Ethereum White Paper*, pages 1–36.

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37).

Cachin, C. and Vukolić, M. (2017). Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*.

Cadwalladr, C. (2016). Google, democracy and the truth about internet search. *The Guardian*, 4(12):2016.

Camino, R., Torres, C. F., Baden, M., and State, R. (2019). A data science approach for honeypot detection in ethereum. *arXiv preprint arXiv:1910.01449*.

Carter, J. L. and Wegman, M. N. (1979). Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154.

Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2020a). A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43.

Chen, T., Li, X., Luo, X., and Zhang, X. (2017). Under-optimized smart contracts devour your money. In *Saner'17 - Early Research Achievements*.

Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X., Lui, J. C.-S., Lin, X., and Zhang, X. (2020b). Understanding ethereum via graph analysis. *ACM Trans. Internet Technol.*, 20(2).

Chen, W., Zheng, Z., Ngai, E. C.-H., Zheng, P., and Zhou, Y. (2019). Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access*, 7:37575–37586.

Chen, Y., Oney, S., and Lasecki, W. S. (2016). Towards providing on-demand expert support for software developers. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 3192–3203.

Chidamber, S. R. and Kemerer, C. F. (1991a). Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 197–211, New York, NY, USA. Association for Computing Machinery.

Chidamber, S. R. and Kemerer, C. F. (1991b). Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 197–211, New York, NY, USA. Association for Computing Machinery.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Chodorow, K. (2013). *MongoDB: The Definitive Guide*. O'Reilly Media, Inc.

Conboy, K. and Fitzgerald, B. (2010). Method and developer characteristics for effective agile method tailoring: A study of xp expert opinion. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):1–30.

Concas, G., Marchesi, M., Murgia, A., Pinna, S., and Tonelli, R. (2010). Assessing traditional and new metrics for object-oriented systems. *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pages 24–31.

Concas, G., Monni, C., Orrù, M., and Tonelli, R. (2013). A study of the community structure of a complex software network. In *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 14–20.

Courtois, N. T. (2014). On the longest chain rule and programmed self-destruction of crypto currencies.

Coxe, S., West, S., and Aiken, L. (2009). The analysis of count data: A gentle introduction to poisson regression and its alternatives. *Journal of personality assessment*, 91:121–36.

Crafa, S., Di Pirro, M., and Zucca, E. (2019). Is solidity solid enough? In *International Conference on Financial Cryptography and Data Security*, pages 138–153. Springer.

Crosby, M., Pattanayak, P., Verma, S., Kalyanaraman, V., et al. (2016). Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71.

Dahse, J. and Holz, T. (2014). Simulation of built-in php features for precise static code analysis. In *NDSS*, volume 14, pages 23–26. Citeseer.

Dannen, C. (2017). *Introducing Ethereum and solidity*, volume 1. Springer.

de Villiers, A. and Cuffe, P. (2020). A three-tier framework for understanding disruption trajectories for blockchain in the electricity industry. *IEEE Access*, 8:65670–65682.

Decker, C. and Wattenhofer, R. (2013). Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10.

Destefanis, G. (2021). Design patterns for smart contract in ethereum. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pages 121–122. IEEE.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., and Hierons, R. (2018). Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE.

Diogo, M., Cabral, B., and Bernardino, J. (2019). Consistency models of nosql databases. *Future Internet*, 11(2):43.

Dorri, A., Kanhere, S. S., and Jurdak, R. (2017). Towards an optimized blockchain for iot. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 173–178. IEEE.

Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE.

Ducasse, S., Rocha, H., Bragagnolo, S., Denker, M., and Francomme, C. (2019a). SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis. In *Blockchain and Web 3.0: Social, economic, and technological challenges*. Routledge.

Ducasse, S., Rocha, H., Bragagnolo, S., Denker, M., and Francomme, C. (2019b). SmartAnvil: Open-Source Tool Suite for Smart Contract Analysis. In *Blockchain and Web 3.0: Social, economic, and technological challenges*. Routledge.

Dunphy, P. and Petitcolas, F. A. (2018). A first look at identity management schemes on the blockchain. *IEEE Security & Privacy*, 16(4):20–29.

Dyson, S. F., Buchanan, W. J., and Bell, L. (2020). Scenario-based creation and digital investigation of ethereum erc20 tokens. *Forensic Science International: Digital Investigation*, 32:200894.

Easley, D., O'Hara, M., and Basu, S. (2017). From mining to markets: the evolution of bitcoin transaction fees. *SSRN Electronic Journal*, pages 1–55.

Efanov, D. and Roschin, P. (2018). The all-pervasiveness of the blockchain technology. *Procedia Computer Science*, 123:116 – 121. 8th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2017 (Eighth Annual Meeting of the BICA Society), held August 1-6, 2017 in Moscow, Russia.

Eklund, P. and Haemmerlé, O., editors (2008). *Conceptual Structures: Knowledge Visualization and Reasoning*. Springer Berlin Heidelberg.

Eklund, P. W. and Beck, R. (2019). Factors that impact blockchain scalability. In *Proceedings of the 11th International Conference on Management of Digital EcoSystems*, MEDES '19, page 126–133, New York, NY, USA. Association for Computing Machinery.

et al., M. A. (2017). zeppelin_os: An open-source, decentralized platform of tools and services on top of the evm to develop and manage smart contract applications securely.

Ethereum Community (2016). Ethereum homestead documentation.

Ethereum Foundation (2020). Solidity documentation release 0.6.12.

Faye, S., Bronzi, W., Tahirou, I., and Engel, T. (2017). Characterizing user mobility using mobile sensing systems. *International Journal of Distributed Sensor Networks*, 13(8):155014771772631.

Fenton, N. E. (1991). *Software Metrics: A Rigorous Approach*. Chapman & Hall, Ltd., GBR.

Fenton, N. E. and Neil, M. (2000). Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 357–370, New York, NY, USA. Association for Computing Machinery.

Few, S. (2006). *Information Dashboard Design*. OReilly.

Forkan, A. R. M., Kimm, G., Morshed, A., Jayaraman, P. P., Banerjee, A., and Huang, W. (2019). Aqvision: A tool for air quality data visualisation and pollution-free route tracking for smart city. In *2019 23rd InfoVis*, pages 47–51.

Forward, A. and Lethbridge, T. C. (2002). The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA. Association for Computing Machinery.

Fusco, F., Lunesu, M. I., Pani, F. E., and Pinna, A. (2018). Crypto-voting, a blockchain based e-voting system. In *KMIS*, pages 221–225.

Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 147–156, New York, NY, USA. Association for Computing Machinery.

Galvez, J. F., Mejuto, J. C., and Simal-Gandara, J. (2018). Future challenges on the use of blockchain for food traceability analysis. *TrAC Trends in Analytical Chemistry*, 107:222–232.

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. (2015). The spack package manager: bringing order to hpc software chaos. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Garrett, J. J. (2010). *The elements of user experience: user-centered design for the web and beyond*. Pearson Education.

Gatteschi, V., Lamberti, F., Demartini, C., Pranteda, C., and Santamaría, V. (2018). Blockchain and smart contracts for insurance: Is the technology mature enough? *Future Internet*, 10(2):20.

Gervais, A., Karame, G. O., Wüst, K., Glykantzis, V., Ritzdorf, H., and Capkun, S. (2016). On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16.

Gill, T. G. (1995). Early expert systems: Where are they now? *MIS quarterly*, pages 51–81.

Giudici, P. and Abu-Hashish, I. (2018). What determines bitcoin exchange prices? a network var approach. *Finance Research Letters*.

Godfrey, P., Gryz, J., and Lasek, P. (2016). Interactive visualization of large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2142–2157.

Gousios, G. and Spinellis, D. (2017). Mining software engineering data from github. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 501–502. IEEE Press.

Granger, C. W. J. (1969). Investigating Causal Relations by Econometric Models and Cross-Spectral Methods. *Econometrica*, 37(3):424–438.

Granger, C. W. J. (1981). Some properties of time series data and their use in econometric model specification. *Journal of Econometrics*, 16(1):121–130.

Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., and Smaragdakis, Y. (2018). Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA).

Greene, W. (2011). Models for counts of events. In *Econometric Analysis*, chapter 18, pages 802–828. Pearson Education, 7th edition.

Greenfield, J. and Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27.

Grimaldi, D. A. and Engel, M. S. (2007). Why Descriptive Science Still Matters. *BioScience*, 57(8):646–647.

Gürkaynak, G., Yılmaz, İ., Yeşilaltay, B., and Bengi, B. (2018). Intellectual property law and practice in the blockchain realm. *Computer law & security review*, 34(4):847–862.

Hackius, N. and Petersen, M. (2017). Blockchain in logistics and supply chain: trick or treat? In *Digitalization in Supply Chain Management and Logistics: Smart and Digital Solutions for an Industry 4.0 Environment. Proceedings of the Hamburg International Conference of Logistics (HICL), Vol. 23*, pages 3–18. Berlin: epubli GmbH.

Hall, K. W., Bradley, A. J., Hinrichs, U., Huron, S., Wood, J., Collins, C., and Carpendale, S. (2019). Design by immersion: A transdisciplinary approach to problem-driven visualizations. *IEEE transactions on visualization and computer graphics*, 26(1):109–118.

Hansen, C. (2014). *Scientific visualization : uncertainty, multifield, biomedical, and scalable visualization*. Springer, London.

Hausmann, J. H. and Kent, S. (2003). Visualizing model mappings in uml. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, page 169–178, New York, NY, USA. Association for Computing Machinery.

Hawlitschek, F., Notheisen, B., and Teubner, T. (2018). The limits of trust-free systems: A literature review on blockchain technology and trust in the sharing economy. *Electronic commerce research and applications*, 29:50–63.

He, N., Wu, L., Wang, H., Guo, Y., and Jiang, X. (2019). Characterizing code clones in the ethereum smart contract ecosystem. *CoRR*, abs/1905.00272.

Hegedűs, P. (2018). Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, pages 35–39, New York, NY, USA. ACM.

Hjalmarsson, F., Hreidharsson, G. K., Hamdaqa, M., and Hjalmtysson, G. (2018). Blockchain-based e-voting system. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 983–986. IEEE.

Hofman, W., Spek, J., and Brewster, C. (2017). Applying blockchain technology for hyperconnected logistics. In *4th International Physical Internet Conference, 4th-6th July 2017 Graz University of Technology, Graz Austria*.

Hölbl, M., Kompara, M., Kamišalić, A., and Nemec Zlatolas, L. (2018). A systematic review of the use of blockchain in healthcare. *Symmetry*, 10(10):470.

Iacobucci, E. and Ducci, F. (2019). The google search case in europe: Tying and the single monopoly profit theorem in two-sided markets. *European Journal of Law and Economics*, 47(1):15–42.

Ibba, S., Pinna, A., Lunesu, M., Marchesi, M., and Tonelli, R. (2018). Initial coin offerings and agile practices. *Future Internet*, 10(11):103.

Jaccheri, L. and Osterlie, T. (2007). Open source software: A source of possibilities for software engineering education and empirical software engineering. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, page 5, USA. IEEE Computer Society.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670.

Kanda, R. and Shudo, K. (2019). Estimation of data propagation time on the bitcoin network. In *Proceedings of the Asian Internet Engineering Conference*, AINTEC '19, pages 47–52, New York, NY, USA. ACM.

Kassab, M., DeFranco, J., Malas, T., Destefanis, G., and Neto, V. V. G. (2019a). Investigating quality requirements for blockchain-based healthcare systems. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 52–55. IEEE.

Kassab, M., DeFranco, J., Malas, T., Neto, V. V. G., and Destefanis, G. (2019b). Blockchain: A panacea for electronic health records? *2019 IEEE/ACM 1st International Workshop on Software Engineering for Healthcare (SEH)*, pages 21–24.

Kassab, M. H., DeFranco, J., Malas, T., Laplante, P., Neto, V. V. G., et al. (2019c). Exploring research in blockchain for healthcare and a roadmap for the future. *IEEE Transactions on Emerging Topics in Computing*.

Keim, D. A. (2002). Information visualization and visual data mining. *IEEE Trans. Vis. Comput. Graph*, 8(1):1–8.

Kerren, A., Stasko, J., Fekete, J.-D., and North, C. (2008). *Information Visualization: Human-Centered Issues and Perspectives*. Lecture notes in computer science. Springer.

Kim, I., Cho, G., Hwang, J., Li, J., and Han, S. (2010). Visualization of neutral model of ship pipe system using x3d. In *Lecture Notes in Computer Science*, pages 218–228. Springer Berlin Heidelberg.

Kiviat, T. I. (2015). Beyond bitcoin: Issues in regulating blockchain tranactions. *Duke LJ*, 65:569.

Kobryn, C. (2002). Will uml 2.0 be agile or awkward? *Commun. ACM*, 45(1):107–110.

Kochovski, P., Gec, S., Stankovski, V., Bajec, M., and Drobintsev, P. D. (2019). Trust management in a blockchain based fog computing platform with trustless smart oracles. *Future Generation Computer Systems*, 101:747 – 759.

Kondo, M., Oliva, G. A., Jiang, Z. M. J., Hassan, A. E., and Mizuno, O. (2020). Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform. *Empirical Software Engineering*, 25(6):4617–4675.

Kousser, T. and McCubbins, M. D. (2004). Social choice, crypto-initiatives, and policymaking by direct democracy. *S. Cal. L. Rev.*, 78:949.

Kratzke, N. (2020). Volunteer down: How covid-19 created the largest idling supercomputer on earth. *Future Internet*, 12(6):98.

Kshetri, N. and Voas, J. (2018). Blockchain-enabled e-voting. *IEEE Software*, 35(4):95–99.

Kube, N. (2018). Daniel drescher: Blockchain basics: a non-technical introduction in 25 steps.

Kuhn, A., Ducasse, S., and Gîrba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and software technology*, 49(3):230–243.

Lee, J. Y. (2019). A decentralized token economy: How blockchain and cryptocurrency can revolutionize business. *Business Horizons*, 62(6):773 – 784. Digital Transformation and Disruption.

Li, X., Jiang, P., Chen, T., Luo, X., and Wen, Q. (2020). A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192.

Liskov, B. and Zilles, S. (1974). Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA. Association for Computing Machinery.

Liskov, B. H. (1972). A design methodology for reliable software systems. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), page 191–199, New York, NY, USA. Association for Computing Machinery.

Liu, X., Muhammad, K., Lloret, J., Chen, Y.-W., and Yuan, S.-M. (2019). Elastic and cost-effective data carrier architecture for smart contract in blockchain. *Future Generation Computer Systems*, 100:590 – 599.

Lo, S. K., Xu, X., Staples, M., and Yao, L. (2020). Reliability analysis for blockchain oracles. *Computers and Electrical Engineering*, 83:106582.

Loeliger, J. (2012). *Version control with Git*. O'Reilly Media, Sebastopol, Calif.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *CCS'2016 (ACM Conference on Computer and Communications Security)*.

Mahling, A., Herczeg, J., Herczeg, M., and Böcker, H.-D. (1988). Beyond visualization: Knowing and understanding. In *Lecture Notes in Computer Science*, pages 16–26. Springer.

Marai, G. E. (2018). Activity-centered domain characterization for problem-driven scientific visualization. *IEEE Trans. Vis. Comput. Graph*, 24(1):913–922.

Marchese, A. and Tomarchio, O. (2021). An agri-food supply chain traceability management system based on hyperledger fabric blockchain.

Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., and Tigano, D. (2020a). Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15. IEEE.

Marchesi, L., Marchesi, M., and Tonelli, R. (2020b). Abcde–agile block chain dapp engineering. *Blockchain: Research and Applications*, 1(1-2):100002.

Marchesi, M., Marchesi, L., and Tonelli, R. (2018). An agile software engineering method to design blockchain applications. In *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia*, pages 1–8.

Medeiros, I., Neves, N., and Correia, M. (2015). Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69.

Mense, A. and Flatscher, M. (2018). Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications*, page 375–380, New York, NY, USA. Association for Computing Machinery.

Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. (2002). Peer-to-peer computing.

Min, H. (2019). Blockchain technology for enhancing supply chain resilience. *Business Horizons*, 62(1):35–45.

Möser, M. and Böhme, R. (2015). Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *Financial Cryptography Workshops*, pages 19–33.

Murgia, A., Tonelli, R., Marchesi, M., Concas, G., Counsell, S., McFall, J., and Swift, S. (2012a). Refactoring and its relationship with fan-in and fan-out: An empirical study. *Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR*, pages 63–72.

Murgia, A., Tonelli, R., Marchesi, M., Concas, G., Counsell, S., McFall, J., and Swift, S. (2012b). Refactoring and its relationship with fan-in and fan-out: An empirical study. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 63–72. IEEE.

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system.

Nakamoto, S. et al. (2008). Bitcoin: a peer-to-peer electronic cash system (2008).

Nofer, M., Gomber, P., Hinz, O., and Schiereck, D. (2017). Blockchain. *Business & Information Systems Engineering*, 59(3):183–187.

O'leary, D. E. (1991). Design, development and validation of expert systems: A survey of developers. *Validation, verification and test of knowledge-based systems*, pages 3–20.

Oliva, G. A., Hassan, A. E., and Jiang, Z. M. (2020a). An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, 25(3):1864–1904.

Oliva, G. A., Hassan, A. E., and Jiang, Z. M. J. (2020b). An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, pages 1–41.

Ølnes, S., Ubacht, J., and Janssen, M. (2017). Blockchain in government: Benefits and implications of distributed ledger technology for information sharing.

Ortu, M., Destefanis, G., Kassab, M., Counsell, S., Marchesi, M., and Tonelli, R. (2015). Would you mind fixing this issue? an empirical analysis of politeness and attractiveness in software developed using agile boards. volume 212.

Ortu, M., Orrú, M., and Destefanis, G. (2019). On comparing software quality metrics of traditional vs blockchain-oriented software: An empirical study. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 32–37. IEEE.

Osgood, R. (2016). The future of democracy: Blockchain voting. *COMP116: Information security*, pages 1–21.

O'Donovan, P. and O'Sullivan, D. T. J. (2019). A systematic analysis of real-world energy blockchain initiatives. *Future Internet*, 11(8):174.

Peck, M. E. (2017). Blockchain world-do you need a blockchain? this chart will tell you if the technology can solve your problem. *IEEE Spectrum*, 54(10):38–60.

Piazza, F. S. (2017). Bitcoin and the blockchain as possible corporate governance tools: Strengths and weaknesses. *Bocconi Legal Papers*, 9:125.

Pierro, G. and Tonelli, R. (2020). Paso: A web-based parser for solidity language analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 16–21.

Pierro, G., Tonelli, R., and Marchesi, M. (2020a). Smart-corpus: an organized repository of ethereum smart contracts source code and metrics.

Pierro, G. A. (2020). Oracles data-set.

Pierro, G. A. (2021). Smart-graph: Graphical representations for smart contract on the ethereum blockchain. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 708–714. IEEE.

Pierro, G. A., Bergel, A., Tonelli, R., and Ducasse, S. (2020b). An Interdisciplinary Model for Graphical Representation. In *CIFMA 2020 - 2nd International Workshop on Cognition: Interdisciplinary Foundations, Models and Applications*, Amsterdam / Virtual, Netherlands.

Pierro, G. A., Bergel, A., Tonelli, R., and Ducasse, S. (2020c). An interdisciplinary model for graphical representation. pages 147–158.

Pierro, G. A., Castriotta, M., and Talarico, E. (2019). Aind survey.

Pierro, G. A. and Rocha, H. (2019). The influence factors on ethereum transaction fees. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, pages 24–31, Piscataway, NJ, USA. IEEE Press.

Pierro, G. A. and Rocha, H. (2019a). The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 24–31. IEEE.

Pierro, G. A. and Rocha, H. (2019b). The influence factors on ethereum transaction fees. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, pages 24–31, Piscataway, NJ, USA. IEEE Press.

Pierro, G. A., Rocha, H., Tonelli, R., and Ducasse, S. (2020). Are the gas prices oracle reliable? a case study using the ethgasstation. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–8.

Pierro, G. A., Rocha, H., Tonelli, R., and Ducasse, S. (2020a). Are the gas prices oracle reliable? a case study using the ethgasstation. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–8. IEEE.

Pierro, G. A. and Tonelli, R. (2020a). Paso. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*.

Pierro, G. A. and Tonelli, R. (2020b). Paso: A web-based parser for solidity language analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 16–21.

Pierro, G. A. and Tonelli, R. (2021). Analysis of source code duplication in ethreum smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 701–707. IEEE.

Pierro, G. A., Tonelli, R., and Marchesi, M. (2020b). An organized repository of ethereum smart contracts' source codes and metrics. *Future Internet*, 12(11):197.

Pierro, G. A., Tonelli, R., and Marchesi, M. (2020c). Smart-corpus: an organized repository of ethereum smart contracts source code and metrics. *arXiv preprint arXiv:2011.01723*.

Pilkington, M. (2016). Blockchain technology: principles and applications. In *Research handbook on digital transformations*. Edward Elgar Publishing.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., and Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, 7:78194–78213.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., and Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, 7:78194–78213.

Pinzón, C. and Rocha, C. (2016). Double-spend attack models with time advantange for bitcoin. *Electronic Notes in Theoretical Computer Science*, 329:79 – 103. CLEI 2016 - The Latin American Computing Conference.

Pointcheval, D. and Stern, J. (2000). Security arguments for digital signatures and blind signatures. *Journal of cryptology*, 13(3):361–396.

Porru, S., Pinna, A., Marchesi, M., and Tonelli, R. (2017a). Blockchain-oriented software engineering: challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171. IEEE.

Porru, S., Pinna, A., Marchesi, M., and Tonelli, R. (2017b). Blockchain-oriented software engineering: Challenges and new directions. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 169–171. IEEE Press.

Pradhan, A., Stevens, A., and Johnson, J. (2017). Supply chains are racing to understand blockchain–what chief supply chain officers need to know. *Gartner*.

Ranganthan, V. P., Dantu, R., Paul, A., Mears, P., and Morozov, K. (2018). A decentralized marketplace application on the ethereum blockchain. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 90–97.

Richey, S. and Taylor, J. B. (2017). *Google and Democracy: Politics and the Power of the Internet*. Routledge.

Rieger, M. and Ducasse, S. (1998). Visual detection of duplicated code. In *ECOOP Workshops*.

Rimba, P., Tran, A. B., Weber, I., Staples, M., Ponomarev, A., and Xu, X. (2017). Comparing blockchain and cloud services for business process execution. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 257–260. IEEE.

Rocha, H., Ducasse, S., Denker, M., and Lecerf, J. (2017). Solidity parsing using smacc: Challenges and irregularities. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, pages 1–9.

Roux, O. and Bourdon, J., editors (2015). *Computational Methods in Systems Biology*. Springer International Publishing.

Saberi, S., Kouhizadeh, M., Sarkis, J., and Shen, L. (2019). Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135.

Salerno, J., Yang, S. J., Nau, D., and Chai, S.-K., editors (2011). *Social Computing, Behavioral-Cultural Modeling and Prediction*. Springer Berlin Heidelberg.

Salimitari, M., Chatterjee, M., and Fallah, Y. P. (2020). A survey on consensus methods in blockchain for resource-constrained iot networks. *Internet of Things*, page 100212.

Savelyev, A. (2017). Contract law 2.0:'smart'contracts as the beginning of the end of classic contract law. *Information & Communications Technology Law*, 26(2):116–134.

Savelyev, A. (2018). Copyright in the blockchain era: Promises and challenges. *Computer law & security review*, 34(3):550–561.

Schmidt, S., Jung, M., Schmidt, T., Sterzinger, I., Schmidt, G., Gomm, M., Tschirschke, K., Reisinger, T., Schlarb, F., Benkenstein, D., et al. (2018). Unibright-the unified framework for blockchain based business integration. *White paper, April*.

Scott, B. (2016). How can cryptocurrency and blockchain technology play a role in building social and solidarity finance? Technical report, UNRISD Working Paper.

Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2004). A relational approach to software metrics. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1536–1540, New York, NY, USA. ACM.

Seshagiri, P., Vazhayil, A., and Sriram, P. (2016). Ama: static code analysis of web page for the detection of malicious scripts. *Procedia Computer Science*, 93:768–773.

Shala, B., Trick, U., Lehmann, A., Ghita, B., and Shiaeles, S. (2020). Blockchain and trust for secure, end-user-based and decentralized iot service provision. *IEEE Access*, 8:119961–119979.

Shen, C. and Pena-Mora, F. (2018). Blockchain for cities—a systematic literature review. *IEEE Access*, 6:76787–76819.

Sidorenko, E. L. (2018). The legal status of cryptocurrencies in the russian federation. *Economics, taxes and law*, 11(2):129–137.

Sikorski, J. J., Haughton, J., and Kraft, M. (2017). Blockchain technology in the chemical industry: Machine-to-machine electricity market. *Applied energy*, 195:234–246.

Silva, P., Vavricka, D., Barreto, J., and Matos, M. (2020). Impact of geo-distribution and mining pools on blockchains: A study of ethereum. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 245–252.

Singh, H. J. and Hafid, A. S. (2019). Transaction confirmation time prediction in ethereum blockchain using machine learning. `https://arxiv.org/pdf/1911.11592`.

Singh, H. J. and Hafid, A. S. (2020). Prediction of transaction confirmation time in ethereum blockchain using machine learning. In Prieto, J., Das, A. K., Ferretti, S., Pinto, A., and Corchado, J. M., editors, *Blockchain and Applications*, pages 126–133, Cham. Springer International Publishing.

Sovbetov, Y. (2018). Factors influencing cryptocurrency prices: Evidence from bitcoin, ethereum, dash, litcoin, and monero. *Journal of Economics and Financial Analysis*, 2:1–27.

Stinson, D. R. and Paterson, M. (2018). *Cryptography: theory and practice*. CRC press.

Sun, J., Tang, P., and Zeng, Y. (2020). Games of miners. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 1323–1331, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Svajlenko, J. and Roy, C. K. (2015). Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140.

Swan, M. (2015). *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.".

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First monday*.

Tapscott, A. and Tapscott, D. (2017). How blockchain is changing finance. *Harvard Business Review*, 1(9):2–5.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345.

Thummalapenta, S., Cerulo, L., Aversano, L., and Di Penta, M. (2010). An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, pages 9–16, New York, NY, USA. ACM.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16.

Tilley, S. and Huang, S. (2003). A qualitative assessment of the efficacy of uml diagrams as a form of graphical documentation in aiding program understanding.

In *Proceedings of the 21st Annual International Conference on Documentation*, SIGDOC '03, page 184–191, New York, NY, USA. Association for Computing Machinery.

Tilley, S. R., Müller, H. A., and Orgun, M. A. (1992). Documenting software systems with views. In *Proceedings of the 10th Annual International Conference on Systems Documentation*, SIGDOC '92, page 211–219, New York, NY, USA. Association for Computing Machinery.

Tonelli, R., Destefanis, G., Marchesi, M., and Ortu, M. (2018a). Smart contracts software metrics: a first study. *arXiv preprint arXiv:1802.01517*.

Tonelli, R., Ducasse, S., Fenu, G., and Bracciali, A. (2018b). 2018 ieee 1st international workshop on blockchain oriented software engineering (iwbose). In *2018 IEEE 1st International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*.

Tonelli, R., Pinna, A., Baralla, G., and Ibba, S. (2018c). Ethereum smart contracts as blockchain-oriented microservices. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–2.

Tran, H., Menouer, T., Darmon, P., Doucoure, A., and Binder, F. (2019). Smart contracts search engine in blockchain. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, ICFNDS '19, New York, NY, USA. Association for Computing Machinery.

Turner, M. and Fauconnier, G. (1999). A mechanism of creativity. *Poetics Today*, 20.

van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106.

Vasin, P. (2014). Blackcoin's proof-of-stake protocol v2. *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, 71.

Velasco-Montero, D., Fernandez-Berni, J., Carmona-Galan, R., and Rodríguez-Vázquez, Á. (2018). Optimum selection of dnn model and framework for edge inference. *IEEE Access*, 6:51680–51692.

Vujičić, D., Jagodić, D., and Ranđić, S. (2018). Blockchain technology, bitcoin, and ethereum: A brief overview. In *2018 17th international symposium infoteh-jahorina (infoteh)*, pages 1–6. IEEE.

Weber, I., Gramoli, V., Ponomarev, A., Staples, M., Holz, R., Tran, A. B., and Rimba, P. (2017). On availability for blockchain-based systems. pages 64–73.

Wei, H. and Li, M. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040.

Werbach, K. (2018). Trust, but verify: Why the blockchain needs the law. *Berkeley Tech. LJ*, 33:487.

Weyuker, E. J. (1988). The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675.

Whitaker, M. D. and Pawar, P. (2020). Commodity ecology: From smart cities to smart regions via a blockchain-based virtual community platform for ecological design in choosing all materials and wastes. In *Blockchain Technology for Smart Cities*, pages 77–97. Springer.

Wiener, N. (1956). The theory of prediction. In *Modern mathematics for engineers, Series I*. Beckenham, E. F.

Wong, M. L. and Leung, K. S. (1997). Evolutionary program induction directed by logic grammars. *Evol. Comput.*, 5(2):143–180.

Wood, G. (2018). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Yellow Paper. Byzantium Version e94ebda*, pages 1–39.

Wood, G. (2019). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper. Byzantium version 7e819ec*, pages 1–39. [Online]. Available: `https://ethereum.github.io/yellowpaper/paper.pdf`.

Wright, A. and De Filippi, P. (2015). Decentralized blockchain technology and the rise of lex cryptographia. *Available at SSRN 2580664*.

Wüst, K. and Gervais, A. (2018). Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54. IEEE.

Yli-Huumo, J., Ko, D., Choi, S., Park, S., and Smolander, K. (2016). Where is current research on blockchain technology?—a systematic review. *PloS one*, 11(10):e0163477.

Zacks, J. and Tversky, B. (1999). Bars and lines: A study of graphic communication. *Memory and Cognition*, 27(6):1073–1079.

Zhang, H., Li, Y.-F., and Tan, H. B. K. (2010). Measuring design complexity of semantic web ontologies. *J. Syst. Softw.*, 83(5):803–814.

Zhang, Y., Kasahara, S., Shen, Y., Jiang, X., and Wan, J. (2018). Smart contract-based access control for the internet of things. *IEEE Internet of Things Journal*, 6(2):1594–1605.

Zhao, J. L., Fan, S., and Yan, J. (2016). Overview of business innovations and research opportunities in blockchain and introduction to the special issue.

Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J., and Imran, M. (2020). An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491.

Zheng, Z., Xie, S., Dai, H.-N., Chen, X., and Wang, H. (2018). Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375.

Zhou, Y. and Davis, J. (2005). Open source software reliability model: An empirical approach. In *Proceedings of the Fifth Workshop on Open Source Software Engineering*, 5-WOSSE, page 1–6, New York, NY, USA. Association for Computing Machinery.

Zhu, J., Zhuang, E., Ivanov, C., and Yao, Z. (2011). A data-driven approach to interactive visualization of power systems. *IEEE Transactions on Power Systems*, 26(4):2539–2546.

Zyskind, G., Nathan, O., et al. (2015). Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184. IEEE.