



Universidade Federal De Pernambuco  
Centro De Informática  
Programa De Pós-Graduação Em Ciência da Computação

Juliandson Estanislau Ferreira

**Specification is Law:** Safe Creation and Upgrade of Ethereum Smart Contracts

Recife

2022

Juliandson Estanislau Ferreira

**Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts**

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

**Área de Concentração:** Engenharia de Software e Linguagens de Programação

**Orientador (a):** Doutor Augusto Cezar Alves Sampaio

**Coorientador (a):** Doutor Pedro Ribeiro Gonçalves Antonino

Recife

2022

Catálogo na fonte  
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

F383Se Ferreira, Juliandson Estanislau

*Specification is law: safe creation and upgrade of ethereum smart contracts*  
/ Juliandson Estanislau Ferreira. – 2022.  
80 f.: il., fig., tab.

Orientador: Augusto Cezar Alves Sampaio.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn,  
Ciência da Computação, Recife, 2022.

Inclui referências.

1. Verificação formal. 2. Contratos inteligentes. 3. Ethereum. 4. Solidity. 5.  
Criação segura. 6. Atualização segura I. Sampaio, Augusto Cezar Alves  
(orientador). II. Título.

005.1

CDD (23. ed.)

UFPE - CCEN 2022-159

**Juliandson Estanislau Ferreira**

**“Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts”**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de software e linguagens de programação.

Aprovado em: 12/09/2022.

**BANCA EXAMINADORA**

---

Prof. Dr. Alexandre Cabral Mota  
Centro de Informática / UFPE

---

Profa. Dra. Fabíola Gonçalves Pereira Greve  
Departamento de Ciência da Computação / UFBA

---

Prof. Dr. Augusto Cezar Alves Sampaio  
Centro de Informática / UFPE  
(**Orientador**)

## ACKNOWLEDGEMENTS

I would like to thank God for giving me the opportunity to be a master's student at Centro de Informática (CIn) in Universidade Federal de Pernambuco (UFPE), a quality higher education institution.

I would like to express my deep gratitude to my mentors Augusto Sampaio and Pedro Antonino. The completion of this work was only possible thanks to their insightful suggestions and ideas, patience in the most difficult moments of the journey and for believing in my work.

Additionally, I thank the professors Sérgio Soares and Waldemar Neto for the great attention paid and for the lessons that were very important for the conclusion of the case study in this work.

I would also like to thank the Postgraduate Program in Computer Science at CIn and all professors for creating opportunities, promoting a multicultural environment and social interaction and sharing knowledge. Thanks to you, CIn is an academic center of excellence.

I thank professor Bill Roscoe, FACEPE and The Blockhouse Technology Ltd for the financial support and the necessary infrastructure for the development of the research. And also all members of the FormalBlocks group for all philosophical debates and support on this intense academic journey.

Finally, I would like to express my gratitude to my parents, Maria Ferreira and Isael Ferreira for their constant support in my decisions and unconditional encouragement. I also thank my fiancée, Carolina Santos, who with patience, tenderness and an immense heart, was with me in all the moments of the master's degree, whether they were sad or happy. I am happy to have this great woman by my side to face all the challenges that will come.

"Why? Because it is how it is." (SOUZA, 2015, p.55).

## ABSTRACT

Smart contract evolution is crucial for the success of decentralized applications, and current methods and processes are not well suited to handle these drivers of change, as the knowledge about the software is predominantly stored in informal documents. In addition, they are the building blocks of the "code is law" paradigm: the smart contract's code indisputably describes how its assets are to be managed - once it is created, its code is typically immutable. Faulty smart contracts present the most significant evidence against the practicality of this paradigm; they are well-documented and resulted in assets worth vast sums of money being compromised. To address this issue, the Ethereum community proposed (i) tools and processes to audit/analyse smart contracts, and (ii) design patterns implementing a mechanism to make contract code mutable. Individually, (i) and (ii) only partially address the challenges raised by the "code is law" paradigm. In this work, we combine elements from (i) and (ii) to create a systematic framework that moves away from "code is law" and gives rise to a new "specification is law" paradigm. It allows contracts to be created and upgraded but only if they meet a corresponding formal specification. We explain how formal verification techniques can be used to ensure safety properties of smart contracts during their evolution. Although formal verification methods have the potential of being used in several application fields, we focus on ensuring compliance with its specifications. The process consists of three phases: Formal requirements specification, verification, and deployment. All steps are planned and executed in an integrated way and together they form a framework capable of fostering safe evolution and make it more reliable and secure. The framework is centered around *a trusted deployer*: an off-chain service that formally verifies and enforces specification conformance. We have prototyped this framework, and investigated its applicability to contracts implementing three widely used Ethereum standards: the ERC20 Token Standard, ERC3156 Flash Loans and ERC1155 Multi Token Standard, with promising results.

**Keywords:** formal Verification; smart contracts; ethereum; solidity; safe deployment; safe upgrade.

## RESUMO

A evolução de contratos inteligentes é crucial para o sucesso de aplicações descentralizadas, os métodos e processos atuais não são adequados para lidar com esses drivers de mudança, pois o conhecimento sobre o software está predominantemente armazenado em documentos informais. Além disso, eles são os blocos de construção do paradigma "code is law": o código do contrato inteligente descreve indiscutivelmente como seus ativos devem ser gerenciados - uma vez criado, seu código é imutável. Contratos inteligentes com bugs apresentam a evidência mais significativa contra a praticidade desse paradigma; normalmente eles estão bem documentados e mesmo assim grandes somas de ativos foram comprometidas. Para resolver esse problema, a comunidade Ethereum propôs (i) ferramentas e processos para auditar/analisar contratos inteligentes e (ii) padrões de design que implementam um mecanismo para tornar o código do contrato mutável. Individualmente, (i) e (ii) abordam apenas parcialmente os desafios levantados pelo paradigma "code is law". Neste trabalho, combinamos elementos de (i) e (ii) para criar uma estrutura sistemática que se afasta do "code is law" e dá origem a um novo paradigma "specification is law". Ele permite que contratos sejam criados e atualizados, mas somente se eles atenderem a uma determinada especificação formal. Explicamos como as técnicas formais de verificação podem ser usadas para garantir as propriedades de segurança dos contratos inteligentes durante sua evolução. Embora os métodos formais de verificação tenham potencial para serem utilizados em diversos campos de aplicação, focamos em garantir a conformidade com suas especificações. O processo consiste em três fases: especificação de requisitos formais, verificação e implantação. Todas as etapas são planejadas e executadas de forma integrada e juntas formam uma estrutura capaz de promover uma evolução segura e torná-la mais confiável. O framework está centrado em *trusted deployer*: um serviço off-chain que verifica e reforça formalmente conformidade de especificação. Prototipamos essa estrutura e investigamos sua aplicabilidade a contratos que implementam três padrões Ethereum amplamente utilizados: o ERC20 Token Standard, ERC3156 Flash Loans e ERC1155 Multi Token Standard, com resultados promissores.

**Palavras-chaves:** verificação formal; contratos inteligentes; ethereum; solidity; criação segura; atualização segura.



## LIST OF FIGURES

Figure 1 – Blockchain Structure . . . . .	18
Figure 2 – Smart contract example . . . . .	25
Figure 3 – ToyWallet contract example . . . . .	27
Figure 4 – Buggy ToyWallet with specification . . . . .	30
Figure 5 – Trusted deployer architecture . . . . .	33
Figure 6 – ToyWallet specification. . . . .	35
Figure 7 – Proxy Structure . . . . .	38
Figure 8 – ToyWallet proxy . . . . .	40
Figure 9 – Trusted registry . . . . .	41
Figure 10 – Trusted Deployer behaviour . . . . .	42
Figure 11 – Trusted Deployer Architecture . . . . .	44
Figure 12 – Screenshot Create Smart Contract . . . . .	47
Figure 13 – Screenshot Upgrade Smart Contract . . . . .	48
Figure 14 – Merged ERC20 Contract . . . . .	50
Figure 15 – ERC20 Proxy . . . . .	51
Figure 16 – ERC20 specification . . . . .	57
Figure 17 – Buggy ERC20 transferFrom function . . . . .	58
Figure 18 – Buggy ERC20 allowance function . . . . .	58
Figure 19 – transferFrom function before refactoring . . . . .	59
Figure 20 – Successful refactoring of the transferFrom function . . . . .	59
Figure 21 – ERC3156 specification . . . . .	61
Figure 22 – Buggy flashLoan function . . . . .	62
Figure 23 – Buggy flashFee and maxFlashLoan functions . . . . .	62
Figure 24 – ERC1155 specification . . . . .	63
Figure 25 – balanceOfBatch function before refactoring . . . . .	64
Figure 26 – Successful refactoring of the balanceOfBatch function . . . . .	65
Figure 27 – Buggy ERC1155 safeBatchTransferFrom function . . . . .	65
Figure 28 – Buggy ERC1155 safeTransferFrom function . . . . .	66

## LIST OF TABLES

Table 1 – 0xMonorepo Commit History . . . . .	49
Table 2 – ERC20 Results . . . . .	58
Table 3 – ERC3156 Results . . . . .	60
Table 4 – ERC1155 Results . . . . .	64

## CONTENTS

<b>1</b>	<b>INTRODUCTION . . . . .</b>	<b>12</b>
1.1	CONTRIBUTIONS . . . . .	15
1.2	OUTLINE . . . . .	17
<b>2</b>	<b>BACKGROUND . . . . .</b>	<b>18</b>
2.1	BLOCKCHAIN . . . . .	18
2.2	SMART CONTRACTS . . . . .	23
2.3	ETHEREUM AND SOLIDITY . . . . .	25
2.4	FORMAL VERIFICATION WITH SOLC-VERIFY . . . . .	29
<b>3</b>	<b>SAFE SMART CONTRACT DEPLOYMENT . . . . .</b>	<b>32</b>
3.1	TRUSTED DEPLOYER FRAMEWORK . . . . .	32
3.2	VERIFIER . . . . .	34
3.3	UPGRADER . . . . .	37
3.4	REGISTRY . . . . .	40
<b>4</b>	<b>TOOL SUPPORT AND APPLICATION TO A REAL COMMIT HISTORY . . . . .</b>	<b>43</b>
4.1	TRUSTED DEPLOYER TOOL . . . . .	43
4.2	SMART CONTRACT CREATION PROCESS . . . . .	46
4.3	SMART CONTRACT UPGRADING PROCESS . . . . .	47
4.4	TOOL APPLICATION TO A SMART CONTRACT COMMIT HISTORY . . . . .	48
<b>5</b>	<b>CASE STUDIES: ERC20, ERC1155, AND ERC3156 . . . . .</b>	<b>52</b>
5.1	CONTEXT . . . . .	52
5.2	PROCESS OVERVIEW . . . . .	54
5.3	ERC20 . . . . .	55
5.4	ERC3156 . . . . .	59
5.5	ERC1155 . . . . .	62
5.6	RESULTS AND DISCUSSION . . . . .	66
5.7	LIMITATIONS AND THREATS . . . . .	67
<b>6</b>	<b>CONCLUSION . . . . .</b>	<b>69</b>
6.1	CONTRIBUTIONS . . . . .	69
6.2	RELATED WORK . . . . .	70

6.3 FUTURE WORK . . . . . 72

REFERENCES . . . . . 74

## 1 INTRODUCTION

Blockchain was introduced to the world in 2009 with the publication of the article **Bitcoin: A Peer-to-Peer Electronic Cash System** (NAKAMOTO, 2009). Since then, interest in this paradigm has grown significantly, being considered today by many experts as a high-grade innovative technology, since it is reshaping conventional industry and business processes and it has the potential to profoundly impact the way society is organized.

A *smart contract* is a stateful reactive program that is stored in and processed by a trusted platform, typically a blockchain, which securely executes such a program and safely stores its persistent state. The use of a trusted platform guarantees that it is computationally infeasible to tamper with its execution or persistent state in an undetectable way. Smart contracts were created to provide an unambiguous, automated, and secure way to manage digital assets. The terms of the contract would be written as a set of instructions and would be executed without the need for a regulatory entity, allowing anonymous parties to make transactions, without the need for an intermediary (SZABO, 1996). *Smart contracts* are meant to be building blocks of the “code is law” paradigm, indisputably describing how its assets are to be managed. To implement this paradigm, many smart contract platforms - including Ethereum, the platform we focus on - disallow the code of a contract to be changed once deployed, effectively enforcing a notion of *code/implementation immutability*.

This requirement of code immutability, however, has drawbacks. Firstly, if the code is found to be incorrect after being deployed, the contract cannot be patched. There are many examples of real-world contract instances with such flaws that have been exploited, putting at risk their digital assets. The increased valuation of these assets, nowadays, is a significant incentive to perpetrators of such attacks. Secondly, the execution of a contract function has an explicit cost to be paid by the participant that requests it, typically in the cryptocurrency underlying the platform, that is calculated based on the contract’s implementation. Platform participants would, then, benefit from contracts being updated to a functionally-equivalent but more cost-effective implementation, which is disallowed by this sort of code immutability.

To overcome this limitation, the Ethereum community has adopted the *proxy pattern* (OPEN-ZEPPELIN, 2021), a mechanism by which one can mimic contract upgrades by splitting a contract into two instances: the *proxy instance* holds the contract’s persistent state, and the *implementation instance* the code associated with its functions. The proxy instance stores a

pointer to the current implementation instance, and relies on the code deployed on the latter to execute. Hence, an update can be carried out (i.e. mimicked) by: (a) deploying a new implementation instance, and (b) changing the proxy instance to point to this new instance. To avoid arbitrary updates, a specific platform participant, the maintainer of the contract, is given solo access to performing (b) and is expected to carry out only reasonable updates.

The simple application of this pattern, however, presents a number of potential issues. Firstly, the use of this mechanism allows for the patching of smart contracts but it does not address the fundamental underlying problem of correctness. Once an issue is detected, it can be patched but (i) it may be too late, and (ii) what if the patch is faulty too? Secondly, it typically gives an, arguably, unreasonable amount of power to the maintainer of this contract. Such maintainers can change the code that is executing without any oversight. They could, for instance, change the contract implementation to allow them to transfer the contract's digital assets to their own private wallet. One could implement a mechanism by which a number of participants could vote for an upgrade, and a given quorum would be a precondition for the update to be executed - preventing such arbitrary updates - but even such an approach has undesirable consequences. The voters could collude to upgrade the contract to some implementation that suits them alone, misrepresenting, then, the interests of the contract instance's stakeholders. The main flaw of such an approach is, arguably, the fact that no guarantees are enforced by this updating process; the contract implementations can change rather arbitrarily as long as the right participants have approved the change. In such a context, the "code is law" paradigm is in fact nonexistent.

In addition to the aforementioned problems, the development of smart contract-based applications still represent a great challenge, since there are conflicts with the traditional software development life cycles models, usually followed by software engineers, especially on the testing and maintenance cycles (HUISMAN; GUROV; MALKIS, 2020). The development of these kinds of applications is a relatively new activity in software engineering, so it lacks best practices for writing safe code and tools/techniques to verify code correctness. Vitalik Buterin, creator of the ethereum platform and great enthusiast of decentralized technologies, once stated:

Buterin, Vitalik (VitalikButerin). "Most instances of smart contract bugs I've seen have nothing to do with turing completeness vs decidability. More logic errors and typos<sup>1</sup>." 04/20/2017, 5:52. Tweet.

<sup>1</sup> This tweet can be found in: <https://twitter.com/vitalikbuterin/status/868751724311216128>

In the last few years, there has been a significant increase in the volume of interest in smart contract evolution. The main reason for this interest is the fact that people are increasingly dependent on these systems to develop their activities. Evolving and maintaining them is therefore crucial because any software must meet the expectations of the stakeholders. These expectations are usually expressed through requirements which are a set of descriptions of how the system to be developed must behave, or a set of properties, or system attributes, or limitations of the software development process itself. Requirements management is the first and most fundamental step in the lifecycle of the software development process, as even if the system is well designed and coded, but poorly specified, it will certainly lead to great losses, therefore it is necessary to verify that the systems satisfy the contracts, standards, or specifications that have been proposed. Software changes can happen in two ways, maintenance when the software changes with the objective of correcting, adapting or amplifying the software functionality and evolution when the software is modified to adapt it to changing stakeholders needs.

In this context, systems composed of smart contracts must be treated in a special way, as they are immutable after the deployment and need to be built completely at once, in addition they must be able to withstand years of security attacks with code that cannot be modified. Although smart contracts are present in an increasing range of applications, their development is still not so simple. For the application to be successful, the contracts that compose it must be extensively planned, taking into account all possible exceptions (ALCHEMY, 2018).

In order to solve these problems, a great effort has been spent on the proposal and creation of several verification tools, as well as formal verification and model checking strategies that can be applied to smart contracts in order to discover bugs before the contract is deployed (RODLER et al., 2021a). Securify (TSANKOV et al., 2018a), SmartCheck (TIKHOMIROV et al., 2018a), Slither (FEIST; GRIECO; GROCE, 2019), Oyente (LUU et al., 2016a), Gasper (CHEN et al., 2017), SASC (ZHOU et al., 2018), MAIAN (NIKOLIC et al., 2018) and Solidifier (ANTONINO; ROSCOE, 2021b) are tools/frameworks created in order to help developers to find vulnerabilities and ensure compliance, but they do not automate the safe, evolution process of smart contracts. Although it is a consensus among academic researchers that the use of formal analysis tools can help to increase the quality and reliability of software, it is still a cumbersome task to integrate them in the daily software-development process (HUISMAN; GUROV; MALKIS, 2020).

To address these issues, we propose a *systematic deployment framework* that requires contracts to be formally verified before they are created and upgraded; we target the Ethereum

platform and smart contracts written in Solidity. We propose a *verification framework* based on the *design by contract methodology* (MEYER, 1992). Ethereum already has a mechanism by which the community can propose and agree on smart contract specifications, called Ethereum Request for Comments (ERCs) - see ERC20 (VOGELSTELLER; BUTERIN, 2015), for instance. They describe the function signatures of a compliant contract implementation together with a brief textual, informal, explanation on how the functions should behave. The format that we propose for our specifications is very similar to that, except that we use a formal notation to capture the behaviour of public functions. Our framework also relies on the proxy pattern to carry out updates but in a rather sophisticated way. We rely on a *trusted deployer*, which is an off-chain service, to vet contract creations and updates. It only allows these operations to be carried out if the given implementation meets the expected specification - the contract specification is set at the time of contract creation and remains unchanged during its lifetime. We also provide a mechanism to test whether a contract has been deployed via our framework so that participants can be certain they are executing a contract with the expected behaviour.

Our framework promotes a paradigm shift where the specification is immutable instead of the implementation/code. Thus, it moves away from "code is law" and proposes the "*specification is law*" paradigm - enforced by formal verification. This new paradigm addresses all the concerns that we have highlighted: arbitrary code updates are forbidden as only conforming implementations are allowed, and buggy contracts are prevented from being deployed as they are vetted by a formal verifier. Thus, contract stakeholders can rely on the guarantee that contract implementations always conform to their corresponding specifications.

We have prototyped our verification framework, and evaluated how its application could bring benefits to the traditional Solidity smart-contract development life cycle. For this evaluation, we investigated real-world contracts implementing three widely used Ethereum contract standards: the ERC20 Token Standard, EIR3156 Flash Loans and ERC1155 Multi Token Standard. For some selected samples, we show how our framework could have prevented an erroneous implementation from being deployed, or that, even after changes, the implementation would still meet the expected specification.

## 1.1 CONTRIBUTIONS

In this work, we develop a smart contract safe evolution and deployment process that clearly outlines how evolve a smart contract, through its life cycle, the code while maintaining



its desired properties defined as a specification. This process is an important step towards simplifying strategic and operational activities. Our purpose is to create a methodology that can simplify the development life cycle, support stakeholders needs and assure software security and quality. The methodology is based on two techniques: Safe evolution and Safe deployment. This combination allowed us to create a systemic approach that incorporates all stages of the development process and takes into account all the specifics of each step. Through this systemic approach we are able to find some of the main errors that may arise during the process of developing smart contracts: logic bugs, reentrancy attacks, arithmetic error of integers, and bad coding patterns. In addition the process is consistent, since it was created to deal with the peculiarities of smart contracts development. Our approach also decreases the costs and risks of the evolution process, since the vulnerabilities are found in the early stage of the process, increasing the security and credibility of the technology. In order to achieve the general objective, we broke it into three specific objectives, such as:

- *Verification framework.* We propose a systematic verification framework based on the *design by contract* methodology that is implemented by the trusted deployer's verifier. We propose the concept of a contract specification, which specifies what are the member variable declarations and public function signatures together with annotations constraining the behaviour of these functions.
- *Trusted deployer infrastructure.* We propose an off-chain *trusted deployer* service and a companion on-chain trusted registry as centrepieces of our framework that allows for the *safe creation, execution and evolution* of smart contracts. We combine formal verification with advanced features of the Ethereum blockchain to implement this framework.
- *Evaluation of our framework on real contracts.* We evaluate our framework on a number of real-world contracts. Furthermore, we have a detailed application of our framework on some samples where we illustrate the benefits that our framework could have in real life. Even though formal verification is a very computationally-intensive process, our evaluation demonstrates that the sort of application we propose seems very tractable.

We propose a limited notion of evolution for smart contracts at the moment: only the implementation of public functions can be upgraded and the persistent state data structures are fixed. However, we are looking into new types of evolution where the data structure of the contract's persistent state can be changed, as well as the interface of the specification, provided

the projected behaviour with respect to the original interface is preserved, based on notions of class (LISKOV; WING, 1994) and process (DIHEGO; ANTONINO; SAMPAIO, 2013) inheritance. We present an infrastructure that applies to the Ethereum platform and for contracts written in Solidity. However, the ideas proposed here should be applicable to similar platforms and to Ethereum contracts written in languages other than Solidity. In this dissertation, we do not focus on how to establish *trust* in the trusted deployer - we simply assume it is a trusted third party. Instead, we focus on the functional aspect of our framework.

## 1.2 OUTLINE

This dissertation is organized in five chapters as follows.

In Chapter 2, we present a literature review exploring the concepts related to smart contracts, solidity, formal verification and the *solc-verify* verification tool. All these concepts explored served as a conceptual basis for the development of the proposed framework.

In Chapter 3, we present an architecture of the tool, explaining the characteristics of each of its design elements: verification framework and trusted deployer infrastructure.

In Chapter 4, we present all the tools and processes used to implement the *Trusted Deployer*. We also present how to use the tool for safe creation and upgrade of smart contracts and its application to a real development scenario.

In Chapter 5, we present and discuss the results obtained by applying our framework to contracts on public repositories that implement the ERC20, ERC3156 and ERC1155 standards.

In Chapter 6, we present the final considerations about the work, the limitations, the threats and we point some directions for future work.

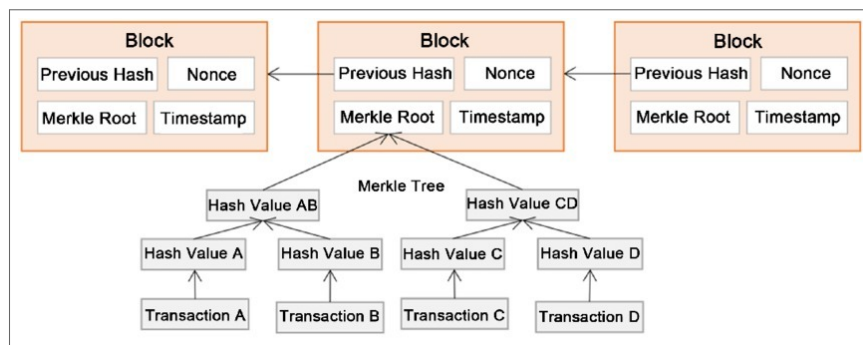
## 2 BACKGROUND

This chapter introduces some background for the following chapters. We present an important description of the concepts and definitions related to blockchain, ethereum, solidity and formal verification with solc-verify.

### 2.1 BLOCKCHAIN

Blockchain is a chronologically ordered chain of blocks, also referred to as a distributed ledger managed in a decentralized manner. The chaining is done by adding the hash (mathematical function that transforms an input of any length into an irreversible output of a fixed size) of the previous block to the current block. This mechanism makes it infeasible to tamper the data stored, since a transaction cannot be changed without changing its block and all the following blocks (AITZHAN; SVETINOVIC, 2016). Furthermore, for any attack to succeed, an adversary must take control over 51% of the whole network computational power. As the network grows and acquires new mining nodes, it makes less likely that chances of this kind of attack taking place, because the bigger the network, the more hash power is needed to take control of it (MCSHANE, 2021). Figure 1 presents the basic elements that make up each block and how they are organized.

Figure 1 – Blockchain Structure



Source: BAHGA; MADISETTI (2016)

Each block contains a timestamp whose function is to help to determine the moment in which the block has been mined and validated by the blockchain network. Although timestamps of the blocks are not accurate, it still serves as a parameter to adjust the mining difficulty, based on how long it takes to extract blocks for a certain period.

Inside each block header, there is a special field called nonce, an acronym for "number only used once", is a number that miners must find out before adding a new block to the chain. Each time a new nonce is tried, a new hash is obtained. This process is necessary as there is no way to know in advance the result of the SHA-256 hashing algorithm. In addition, the blockchain network uses the block hash value to adjust the network difficulty, which tends to be proportional to the number of participating miners. If a nonce generates a valid hash, it is called a golden nonce (OLIVEIRA et al., 2013). Finding a golden nonce is a highly complex mathematical task, since it is a random number and, thus, requires a significant amount of trial-and-error and computer power. In proof-of-work consensus protocols, a miner has to find a hash with a specific structure. Normally, this means a hash starting with a specific number of zeros. By modulating this number of starting zeros, one can set the *difficulty* of the *hash puzzle*. The more starting zeros are required the more difficult it is to find a nonce that will yield such a hash. Usually a reward is offered to the miner who solves the puzzle (FRANKENFIELD, 2021). Once the problem is solved a new block is generated, the miner notifies the network and the process restarts. Merkle tree is a data structure that stores on its leaves the hash value of each transaction of the block. Each node is created by repeatedly calculating hashing pairs of its child nodes until there is only one left which is called the merkle root, used as a summary of all transactions. This technique is widely used in computer science to verify the consistency of large amounts of data efficiently, since it allows to verify an individual transaction without having to search the entire block (CHEN; CHOU; CHOU, 2019).

One of the main advantages of the technology is that it is a new form of decentralized information that can be applicable in a wide range of industries besides financial services. It is usually referred to as a "shared" or "distributed" ledger since all network participants have a copy and share responsibility for keeping it up to date. This is the main difference between blockchain and centralized systems, where control is normally maintained by an institution (YERMACK, 2017). The transactions contained therein are transparent, auditable, reliable, anonymous and there is no need for intermediaries. Its decentralized nature makes it a technology that promotes equality, freedom and new forms of interaction between people (SWAN, 2015). Blockchain represents a major technological advance when compared to the relational models developed in the 70s. Although these relational models have evolved significantly in recent years, there is great difficulty in finding a model that allows databases from different organizations to communicate easily, which normally makes the processes more expensive and bureaucratic. Some models have been proposed to facilitate cooperation between

databases. Hub and Spoke where control of the network is transferred to an intermediary (WANG; YANG; YANG, 2017) and Peer-to-Peer often presents inconsistencies as each machine records its transactions in its own database (KUMAR; SUBBARAO, 2019). Blockchain takes the best of these two models, as each node maintains a copy of the database and its consensus algorithm ensures data consistency. Because it is decentralized, there is no single point of failure, in addition there is no monopoly by any institution, which avoids censorship and increases the transparency of information.

Although Bitcoin was not the first digital payment method that was developed, it was the first to solve the double-spending problem, which is the risk that a digital currency can be spent twice.

This kind of problem is unique to digital assets because digital information can be reproduced relatively easily. Being a decentralized system, blockchain does not need a third-party trusted authority. Instead, to guarantee the reliability and consistency of the data and transactions, blockchain adopts the decentralized consensus mechanism which is sorely needed to establish mutual trust between anonymous parties. A typical Blockchain system usually consists of six layers (WANG et al., 2018), namely:

- *Data layer* : Describes how the information is structured in the blockchain (MICHELIN, 2019). This layer includes data related to blocks, encrypted messages included in transactions, the timestamp, and other underlying data.
- *Network layer*: Blockchain system adopts P2P protocol which is fully distributed and therefore resilient to single point of failure (SPoF) related issues. All nodes are connected in a topological structure without any hierarchy so they are equal, and each peer plays the role of client and server (ANTONPOULOS, 2014) therefore, it does not depend on a central authority to control, coordinate or manage exchanges between peers.
- *Consensus layer*: This layer can encapsulate various types of consensus protocols. The importance of the consensus algorithm is due to the fact that each node modifies its local copies, so it is necessary to have an agreement with the rest of the network to reach a consensus on the state of the ledger. Some of the nodes may be unreliable and therefore require fault tolerance support. The most common consensus algorithms are PoW (Proof of Work) and PoS (Proof of Stake).

- *Incentive layer*: The nodes that participate in the verification and accounting of network data in a decentralized system have an interest of their own: to maximize the rewards obtained. Therefore, the network must have incentive mechanisms to offer to miners. Such incentives ensure the security and effectiveness of decentralized ecosystems and promote the development of the system in a virtuous circle.
- *Contract layer*: This layer encapsulates various types of script codes, algorithms, smart contracts and therefore is the basis for programming and manipulating Blockchain systems. Some platforms, including Bitcoin, use non-Turing-complete scripting language, which means they have no flow control, loops or conditionals. After Bitcoin, more complex and flexible scripting languages for smart contracts emerged, such as Solidity, which allows Blockchain to support a wider range of applications for financial and social systems.
- *Application layer*: On the Ethereum platform, in addition to transactions with the ether cryptocurrency, it also supports decentralized applications (dApps) that run on a P2P network rather than a single server. These applications communicate with the Blockchain, in which it manages the state of all the actors in the network. The smart contract represents the core logic of a dApp. They can process information from sensors or external events and help Blockchain manage the state of the ledger.

The decentralized nature of a blockchain means incentives for attackers to make fraudulent transactions or even reversing legitimate ones, which can create competing or inaccurate ledgers. At its core blockchains are replicated state machines (RSMs) capable of transitioning to a new state based on external interactions and documenting previous states under a Byzantine fault tolerance model (LEE; NIKITIN; SETTY, 2020). They enforce the notion of a “world computer”, so all nodes in the network work together and share a common database. An important property from this definition is that there must be a single valid blockchain state, which is accessible to anyone connected to the network. In this sense, a blockchain can be thought of as a kind of distributed database with a single shared state, which can be updated via the addition of blocks of transactions (SHORISH, 2018).

The two main models used to represent how the state is recorded are *Unspent Transaction Outputs (UTXO)* and *Account-based*. In the UTXO model, the movement of assets is recorded as a directed acyclic graph (DAG). Following a newly created transaction, UTXOs are generated

from the output to represent the value transferred and are then assigned to the receiver's public address. A user can use any UTXOs that are assigned to their private key for further transactions, but once the transaction is verified and stored in the ledger, the UTXO will be considered spent and is no longer usable. This model does not incorporate accounts at the protocol level but uses wallets to maintain balance by recording the sum of all UTXOs for an account (COWLING et al., 2006) (VASSANTLAL et al., 2020).

The account-based transaction model represents assets as balances within accounts. The model tracks the global shared state of asset balances for all accounts on the network. The two account types seen in this model are the private-key user accounts and global smart contract accounts, both of which contain executable code, account balance and internal memory. When a transaction is sent and verified on the ledger, the sender's balance is decreased and the receiving account balance is increased by the same value. After a transaction has occurred, the global state is updated to reflect any changes to account balance (CASTRO; LISKOV, 1999) (VASSANTLAL et al., 2020).

Consensus mechanisms play a major role allowing blockchain systems to work together in a distributed, untrusted environment and stay secure. It aims to solve the problem of how to synchronize data across "nodes", since they need to come to a consensus about the state of the ledger, as there is no central authority to assume responsibility. Among the main consensus protocols, we can highlight:

- **Proof of work (PoW):** It is a consensus algorithm that ensures the authenticity of transactions stored on the blockchain. In order to add a new block to the chain a miner should use their computational power to validate transactions in a competition with each other in a race to find a nonce that will produce a block hash that satisfies the restrictions imposed by the protocol (TORRE; SEANG, 2019). Such a method is used to avoid malicious nodes from producing invalid blocks that could result in attacks, such as denial of service (DoS) or spam (PORAT et al., 2017). A new block is considered as legitimate when other miners validate the process, so the proof must be difficult to create and easy to be verified. Currently, this proposal is used in almost all cryptocurrencies. Producing proof of work, in most cases, is a random process, so a lot of trial and error happens before a valid proof of work is generated (FRANKENFIELD, 2021).
- **Proof of stake (PoS)** participants or nodes have to pledge some amount of digital currency before they can create a new block. For each new expected block, a new signer

of the block will be selected, through random criteria, from the list of participants given the amount of stake they have. Usually those with more stake in the blockchain will be able to add blocks more often (RIBERA, 2018). It reduces dramatically the amount of computational work needed to create blocks being therefore more scalable than PoW.

Although it was initially presented as a solution to specific problems related to the financial services industry, blockchain technology can be adapted by any industry where it is necessary to register, confirm and transfer any type of contract or ownership. Actually there are currently several use cases at various stages of maturity, such as: auctions (HAHN et al., 2017), data management systems (ADHIKARI, 2017), financial contracts (BIRYUKOV; KHOVRATOVICH; TIKHOMIROV, 2017), elections (MCCORRY; SHAHANDASHTI; HAO, 2017) and trading platforms (NOTHEISEN; GÖDDE; WEINHARDT, 2017).

## 2.2 SMART CONTRACTS

Smart contracts are programs stored on a blockchain that automatically enforce its terms when predetermined conditions are met (ZHENG et al., 2020). This concept was proposed in the 1990s with the publication of *Smart Contracts: Building Blocks for Digital Markets* article, by Nick Szabo (SZABO, 1996). According to Szabo:

"New institutions, and new ways to formalize the relationships that make up these institutions, are now made possible by the digital revolution. I call these new contracts "smart," because they are far more functional than their inanimate paper-based ancestors. No use of artificial intelligence is implied. A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises."  
(SZABO, 1996)

In his work, Szabo predicted that the smart contracts would drastically change how people trade with each other, since they will play a major role for the decentralized economy and can be used to carry out financial transactions, property or anything else of value. Furthermore, just like traditional contracts, smart contracts define rules and penalties around an agreement, but perform the obligations automatically and significantly improves four basic aspects of contracts, described as: observability, verifiability, privacy and enforceability. In a digital centralized economy, it is very common to have an intermediary between two parties in a transaction that acts as a certifier that a transaction actually took place and prevents fraud. In addition, the



intermediary controls all information flow on the network and receives a commission for each transaction carried out.

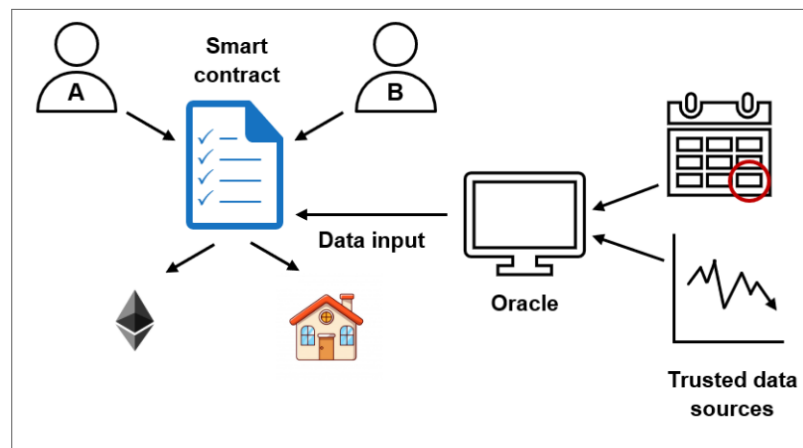
Smart contracts allow anonymous transactions, without the need for a central authority. These characteristics draw attention due to its high degree of independence, in addition, humans do not have a central role, since it is designed to work without a traditional management structure. It is usually idealized and maintained by a community of developers around the world, in fact the main philosophy that governs the principles of a decentralized application is the fact that it never belongs to a specific person or location. Smart contracts must have a predefined behavior and not have the ambiguities of natural language. They are made available to other nodes through the blockchain network and each node processes transactions individually. The role of the network is to update the status of the contract and check that everything went according to its terms.

Blockchain and smart contracts are the foundations of a decentralized economy. Take as example a smart contract that portrays a transaction for trading a property between a buyer and a seller. As shown in Figure 2, the seller registers on the smart contract the property descriptions (such as price, location and size) along with payment terms. The buyer then can verify the authenticity of the information available and submit the payment. The smart contract can also take as input data from external systems (such as interest rate and delivery date) via oracle which is a trusted third-party service that serves as an interface between blockchains and the outside world (BENIICHE, 2020).

This whole procedure is anonymous and no intermediaries are needed, which makes the process cheaper, less bureaucratic, traceable, and auditable. In addition, due to the immutability of blockchains the risk of data tampering is infeasible and the execution of the contract is managed by the network.

One of the most innovative applications is the Decentralized Autonomous Organizations (DAO's), which basically is a network of interconnected smart contracts, capable of performing the same functions as traditional organizations. In this organization, humans do not play the main role, as the system is designed to organize transactions through algorithms. OpenBazaar, for example, is an open source project that provides a platform where anyone can buy or sell products and services in a fully decentralized marketplace. Through this platform people can carry out transactions directly via a peer to peer network without depending on a central regulatory authority. The OpenBazaar protocol is broken down into two pillars: payment and arbitration. The payment protocol is the core pillar and enables buyers to pay sellers

Figure 2 – Smart contract example



Source: MADIR (2018)

for products and services. It benefits users in several aspects when compared to centralized applications because it ensures integrity over the sending and receiving of tokens by holding them until the services or products are properly delivered. The arbitration protocol is designed to solve disputes between buyers and sellers through a moderator. The disputes can arise when the buyer does not release funds from escrow or the buyer is unsatisfied with the contents or delivery of the item. Decentralized organizations have several advantages when compared to traditional organizations. The first is to allow the user to have full control over who has access to his private financial and business information. Another advantage is higher control over business processes as there is no authority to define the rules of operation. Lastly, as it is a decentralized application it does not require middlemen and has no marketplace fees, restrictions, data collection, or censorship. Although they are present in an increasing range of applications, their development is still not so simple because there are still few processes and tools that support their development.

### 2.3 ETHEREUM AND SOLIDITY

Ethereum is an open source, decentralized, distributed computing platform that provides a virtual machine capable of executing smart contracts using blockchain technology. It was created to make life easier for developers who want to create decentralized applications (WOOD, 2014). Unlike the Bitcoin network it was designed to support a wide range of industries although currently most applications are geared towards the financial sector. All this diversity of applications that can be hosted in Ethereum blockchain has turned it into one of the most

popular smart contract platforms.

A participant in Ethereum controls an account, each of which has an *address*, *private* and *public keys* associated with it. The private key generation process is offline and does not require any interaction with the Ethereum network. Each private key is represented as an hexadecimal 64 digits string and it is generated by feeding a string of random bits into a 256-bits hashing algorithm such as Keccak-256 or Sha-256 (MUSUMECI, 2018). The public key is calculated from the private key using the elliptic curve digital signature algorithm (ANTONOPOULOS; WOOD, 2021). This means that a public key is generated from a set of two coordinates combined in a way that satisfies the elliptic curve equation. An ethereum account is akin to the ones in traditional banking systems and a participant that controls it also controls the balance associated with it. Thus, they can send a *transaction* to Ethereum requesting the transfer of some amount of the balance associated with one of its addresses. Aside from these addresses that are managed by external entities, Ethereum also allows addresses to be managed by a *program* (a smart contract). In addition to a balance, these *smart contract* addresses have some code and data associated with them. While the former defines the functions offered by the contract, the latter captures its persistent state.

Solidity is an object-oriented, high-level and statically-typed language widely used for implementing smart contracts for the Ethereum Virtual Machine (EVM) and other blockchain platforms. It was designed to look like javascript syntax and it has support for storage and memory variables or objects, data types, inheritance, mappings, structs, arrays and basic types such as booleans, strings and integers which can be signed and unsigned (RAJ, 2021). A contract in Solidity is a concept very similar to that of a class in object-oriented languages, and a contract instance a sort of long-lived persistent object and all of them are compiled down to EVM bytecode and deployed on the blockchain having it reside at a particular address, thus allowing external entities to trigger its behavior through function calls (AZZOPARDI; ELLUL; PACE, 2018a).

We introduce the main elements of Solidity using the *ToyWallet* contract in Figure 3. It implements a very basic "wallet" contract that participants and other contracts can rely upon to store their Ether. The *member variables* of a contract define the persistent state of the contract. This example contract has a single member variable *accs*, a mapping from addresses to 256-bit unsigned integers, which keeps track of the balance of Ether each "client" of the contract has in the *ToyWallet*; the integer *accs[addr]* gives the current balance for address *addr*, and an address is represented by a 160-bit number.

Figure 3 – ToyWallet contract example

```

1  contract ToyWallet {
3      mapping (address => uint) accs;
      event Transferred(address indexed owner, uint256 value);
5
      modifier checkBalance(uint value) {
7          require(accs[msg.sender] >= value, "Insufficient Funds");
          _;
9      }
      function deposit () payable public {
11         accs[msg.sender] = accs[msg.sender] + msg.value;
      }
13     function withdraw (uint256 value) public checkBalance {
        bool ok = msg.sender.send(value);
15         require(ok);
        accs[msg.sender] = accs[msg.sender] - value;
17         emit Transferred(msg.sender, value);
19     }
}

```

Public functions describe the operations that participants and other contracts can execute on the contract. The contract in Figure 3 has *public functions* *deposit* and *withdraw* that can be used to transfer Ether into and out of the *ToyWallet* contract, respectively. In Solidity, functions have the implicit argument *msg.sender* designating the caller's address, and *payable* functions have the *msg.value* which depict how much *Wei* - the most basic (sub)unit of Ether - is being transferred, from caller to callee, with that function invocation; such a transfer is carried out implicitly by Ethereum. For instance, when *deposit* is called on an instance of *ToyWallet*, the caller can decide on some amount *amt* of Wei to be sent with the invocation. By the time the *deposit* body is about to execute, Ethereum will already have carried out the transfer from the balance associated to the caller's address to that of the *ToyWallet* instance - and *amt* can be accessed via *msg.value*. Note that, as mentioned, this balance is part of the blockchain's state rather than an explicit variable declared by the contract's code. One can programmatically access this implicit balance variable for address *addr* with the command *addr.balance*.

Solidity's construct *require(condition)* aborts and reverts the execution of the function in question if *condition* does not hold - even in the case of implicit Ether transfers. The call *addr.send(amount)* sends *amount* Wei from the currently executing instance to address *addr*; it returns *true* if the transfer was successful, and *false* otherwise. The function *withdraw* has a modifier which is used to change the behaviour of a function. It creates additional features or apply some restriction on function. It can be executed before or after the original function call. For instance, the *require* statement declared in the modifier *checkBalance* requires the

caller to have the funds they want to withdraw, whereas the requires declared in the function *withdraw* requires the *msg.sender.send(value)* statement to succeed, i.e. the *value* must have been correctly withdrawn from *ToyWallet* to *msg.sender*. Then, the account balance of the caller (i.e. *msg.sender*) is updated in *ToyWallet* to reflect the withdrawal. The final statement in this function has an event which is used to inform the calling application about the current state of the contract. Events in the Solidity plays an important role, acting not only as a mere logging mechanism, but also as a means of communication between the decentralized applications and the users of those services (HAJDU; JOVANOVIĆ; CIOCARLIE, 2020).

We use the transaction *create-contract* as a means to create an instance of a Solidity smart contract in Ethereum. In reality, Ethereum only accepts contracts in the *EVM bytecode* low-level language - Solidity contracts need to be compiled into that. The processing of a transaction *create-contract(c, args)* creates an instance of contract *c* and executes its constructor with arguments *args*. Solidity contracts without a constructor (as our example in Figure 3) are given an implicit one. A *create-contract* call returns the address at which the contract instance was created. We omit the *args* when they are not relevant for a call. We use  $\sigma$  to denote the state of the blockchain where  $\sigma[ad].balance$  gives the balance for address *ad*, and  $\sigma[ad].storage.mem$  the value for member variable *mem* of the contract instance deployed at *ad* for this state. For instance, let  $c_{tw}$  be the code in Figure 3, and  $addr_{tw}$  the address returned by the processing of *create-contract(c<sub>tw</sub>)*. For the blockchain state  $\sigma'$  immediately after this processing, we have that: for any address *addr*,  $\sigma'[addr_{tw}].storage.accs[addr] = 0$  and its balance is zero, i.e.,  $\sigma'[addr_{tw}].balance = 0$ . We introduce and use this intuitive notation to present and discuss state changes as it can concisely and clearly capture them. There are many works that formalise such concepts (HILDENBRANDT et al., 2018; WANG et al., 2020; ANTONINO; ROSCOE, 2021a).

A transaction *call-contract* can be used to invoke contract functions; processing *call-contract(addr, func\_sig, args)* executes the function with signature *func\_sig* at address *addr* with input arguments *args*. When a contract is created, the code associated with its non-constructor public functions is made available to be called by such transactions. The constructor function is only run (and available) at creation time. For instance, let  $addr_{tw}$  be a fresh *ToyWallet* instance and *ToyWallet.deposit* give the signature of the corresponding function in Figure 3, processing the transaction *call-contract(addr<sub>tw</sub>, ToyWallet.deposit, args)* where  $args = \{msg.sender = addr_{snd}, msg.value = 10\}$  would cause the state of this instance to be updated to  $\sigma''$  where we have that  $\sigma''[addr_{tw}].storage.accs[addr_{snd}] = 10$  and

$\sigma''[\text{addr}_{tw}].\text{balance} = 10$ . So, the above transaction has been issued by address  $\text{addr}_{snd}$  which has transferred 10 Wei to  $\text{addr}_{tw}$ .

## 2.4 FORMAL VERIFICATION WITH SOLC-VERIFY

Formal verification refers to applying a rigorous mathematical process to prove the correctness of a system with respect to a certain formal specification or property. This technique is based on the idea of a static analysis of the set of possible scenarios represented by an abstract mathematical model and providing a formal proof based on the predefined requirements (MISSON, 2019) and it is used to guarantee that a given system meets its functional and non-functional requirements (CLARKE; WING., 1996). This is an important step in an attempt to ensure that the system will behave as expected. With mathematical methods, one can explore all behavioral possibilities of the system, obtaining a result whether or not an event may happen and it is a method widely used in various industries during the system development process. The main advantage of this method is to facilitate the identification of faults before its implementation. Therefore, many tools have emerged to support the development of secure smart contracts and to aid the analysis of deployed ones. There are several analysis methods that are used by these tools. Static analysis refers to a class of methods that examine the source code or bytecode of a contract without executing it, dynamic analysis means to observe a contract while executing it in the original context and symbolic execution means to execute code using symbolic instead of concrete values for the variables.

The modular verifier *solc-verify*, a source-level verification tool for solidity smart contracts (HAJDU; JOVANOVIĆ, 2020b; HAJDU; JOVANOVIĆ, 2020a) was created to help developers to formally check that their smart contracts behave as expected. Solc-verify supports in-code annotations to specify contract properties. Input contracts are also manually annotated with contract *invariants* and their functions with *loop invariants*, *pre-* and *postconditions*. An annotated Solidity contract is then translated into a Boogie program which is verified by the Boogie verifier (BARNETT et al., 2005; LEINO, 2008).

Its modular nature means that *solc-verify* verifies functions locally/independently, and function calls are abstracted by the corresponding function's specification, rather than their implementation being precisely analysed/executed. These specification constructs have their typical meaning. An invariant is valid if it is established by the constructor and maintained by the contract's public functions, and a function meets its specification if and only if from a state

satisfying its pre-conditions, any state successfully terminating respects its postconditions. So the notion is that of partial correctness. Note that an aborted and reverted execution, such as one triggered by a failing *require* command, does not successfully terminate.

Figure 4 illustrates a specification for an alternative version of the *ToyWallet*'s smart contract. *Solc-verify* provides a *sum* function over collections (arrays and mappings) (HAJDU; JOVANOVIĆ, 2020a). The contract is annotated with the invariant *sum\_uint (accs) == address (this).balance*. When the balance of the contract is updated by *deposit* or *withdraw* a verification is performed to evaluate the consistency between the contract's and wallets balance; if the invariant is violated solc-verify reports an error. The postconditions of the *withdraw* function specify that the balance of the instance and the wallet balance associated with the caller must decrease by the withdrawn amount and no other wallet balance must be affected by the call.

Figure 4 – Buggy ToyWallet with specification

```

1  /** @notice invariant __verifier_sum_uint(accs) == address(this).balance*/
   contract ToyWallet {
3
   mapping (address => uint) accs;
5   event Transferred(address indexed owner, uint256 value);
7
   modifier checkBalance(uint value) {
       require(accs[msg.sender] >= value, "Insufficient Funds");
9       _;
   }
11
   /**
13   * @notice postcondition address(this).balance == __verifier_old_uint(address(
       this).balance) + msg.value
   * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
       sender]) + msg.value
15   * @notice postcondition forall (address addr) addr == msg.sender ||
       __verifier_old_uint(accs[addr]) == accs[addr]
   */
17   function deposit () payable public {
       accs[msg.sender] = accs[msg.sender] + msg.value;
19   }
21
   /**
23   * @notice postcondition address(this).balance == __verifier_old_uint(address(
       this).balance) - value
   * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
       sender]) - value
   * @notice postcondition forall (address addr) addr == msg.sender ||
       __verifier_old_uint(accs[addr]) == accs[addr]
25   * @notice emits Transferred
   */
27   function withdraw (uint256 value) public checkBalance {
       bool ok = msg.sender.send(value);
29       require(ok);
       accs[msg.sender] = accs[msg.sender] - value;
31       emit Transferred(msg.sender, value);
   }
33 }

```

This alternative implementation uses `msg.sender.call.value(value)("")` instead of `msg.sender.send(value)`. While the latter only allows for the transfer of `value` Wei from the instance to address `msg.sender`, the former *delegates control* to `msg.sender` in addition to the transfer of `value`.<sup>1</sup> If `msg.sender` is a smart contract instance that calls `withdraw` again during this control delegation, it can withdraw all the funds in this alternative *ToyWallet* instance - even the funds that were not deposited by it. This *reentrancy* bug is detected by *solc-verify* when it analyses this alternative version of the contract. A similar bug was exploited in what is known as the DAO attack/hack to take over US\$53 million worth of Ether (SIEGEL, 2016; VOLLMER, 2016; ATZEI; BARTOLETTI; CIMOLI, 2017).

*Solc-verify* also allows to define events as part of the specification. This is an important feature since Solidity does not impose limitations on the emitted events so a contract could emit events that do not correspond to state changes. So any function annotated with *emits* must emit an event.

---

<sup>1</sup> In fact, the function `send` also delegates control to `msg.sender` but it does in such a restricted way that it cannot perform any relevant computation. So, for the purpose of this paper and to simplify our exposition, we ignore this delegation.



### 3 SAFE SMART CONTRACT DEPLOYMENT

This chapter describes the design and development process of the trusted deployer framework - used to promote the secure creation and evolution of smart contracts. In addition, it delimits the scope of the framework by pointing out what are the functionalities available and what is out of scope. We present the theoretical bases on which the *trusted deployer framework* (an off-chain service that formally verifies and enforces a notion of conformance between a specification and an implementation) was built. We also discuss all principles needed to deploy, upgrade and manage the smart contract versions. Our framework is generic and, in principle, can integrate with any verification tool that supports design by contract.

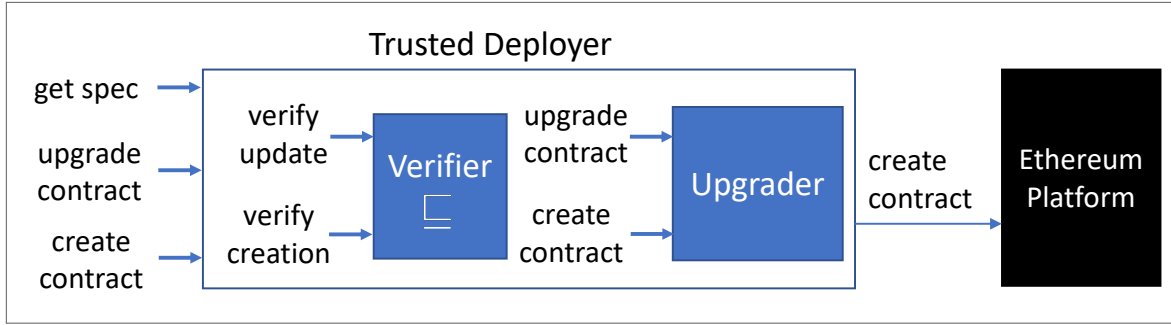
#### 3.1 TRUSTED DEPLOYER FRAMEWORK

We propose a framework for the *safe creation and upgrade of smart contracts* based around a *trusted deployer*. This entity is trusted to only create or update contracts that have been verified to meet their corresponding specifications. A smart contract development process built around it prevents developers from deploying contracts that have not been implemented as intended. Thus, stakeholders can be sure that contract instances deployed by this entity, even if their code is upgraded, comply with the intended specification.

Our trusted deployer targets the Ethereum platform, and we implement it as an off-chain service. Generally speaking, a trusted deployer could be implemented as a smart contract in a blockchain platform, as part of its consensus rules, or as an off-chain service. In Ethereum, implementing it as a smart contract is not practically feasible as a verification infrastructure on top of the EVM (BUTERIN, 2014) would need to be created. Furthermore, blocks have an upper limit on the computing power they can use to process their transactions, and even relatively simple computing tasks can exceed this upper limit (WüST et al., 2020). As verification is a notoriously complex computing task, it should exceed this upper limit even for reasonably small systems. Neither can we change the consensus rules for Ethereum.

We present the architecture of the *trusted deployer infrastructure* in Figure 5. The trusted deployer relies on an internal *verifier* that implements the functions *verify-creation*<sub>□</sub> and *verify-upgrade*<sub>□</sub>, and an *upgrader* that implements functions *create-contract* and *upgrade-contract*; we detail what these functions do in the following. The deployer's *create-contract*

Figure 5 – Trusted deployer architecture



(*upgrade-contract*) check that an implementation meets its specification by calling *verify-creation*<sub>⊆</sub> (*verify-upgrade*<sub>⊆</sub>) before relaying this call to the upgrader's *create-contract* (*upgrade-contract*) which effectively creates (upgrades) the contract in the Ethereum platform. The *get-spec* function can be used to test whether a contract instance has been deployed by the trusted deployer and which specification it satisfies.

The trusted deployer maintains an internal persistent variable *registry* mapping addresses of instances to the specification they meet. This service offers functions *create-trusted-contract*, and *upgrade-trusted-contract* as per Algorithm 1, and function *get-spec* to access *registry*. A value of type *Option* $\langle T \rangle$  is either of the form *Some*(*t*) where *t*  $\in T$  or a value *None*. The trusted deployer relies on an internal *verifier* that implements the functions *verify-creation*<sub>⊆</sub> and *verify-upgrade*<sub>⊆</sub>, and an *upgrader* that implements functions *create-upgradable-contract* and *upgrade-upgradable-contract*.

The *verifier* is used to establish whether an implementation meets a specification. A verification framework is given by a triple  $(\mathcal{S}, \mathcal{C}, \sqsubseteq)$  where  $\mathcal{S}$  is a language of smart contract specifications,  $\mathcal{C}$  is a language of implementations, and  $\sqsubseteq \in (\mathcal{S} \times \mathcal{C})$  is a satisfiability relation between smart contracts' specifications and implementations. In this paper,  $\mathcal{C}$  is the set of Solidity contracts and  $\mathcal{S}$  a particular form of Solidity contracts, possibly annotated with contract invariants, that include function signatures annotated with postconditions. The functions *verify-creation*<sub>⊆</sub> and *verify-upgrade*<sub>⊆</sub> both take a specification *s*  $\in \mathcal{S}$  and a contract implementation *c*  $\in \mathcal{C}$  and test whether *c* meets *s* - they work in slightly different ways as we explain later. When an implementation does not meet a specification, verifiers typically return an error report that points out which parts of the specification do not hold and maybe even witnesses/counterexamples describing system behaviours illustrating such violations; they provide valuable information to help developers correct their implementations.

The upgrader is used to create and manage *upgradable smart contracts* - Ethereum does

---

**Algoritmo 1:** Implementation of a trusted deployer.

---

**State:**  $registry : \text{map}(\text{address} \mapsto \mathcal{S})$

```

create-trusted-contract( $s : \mathcal{S}, c : \mathcal{C}, \text{args} : \text{Args}$ ) :  $\text{Option}\langle \text{address} \rangle$ 
if  $\text{verify-creation}_{\sqsubseteq}(s, c)$  then
  |  $\text{addr} = \text{create-upgradable-contract}(c, \text{args});$ 
  |  $\text{registry}[\text{addr}] = s;$ 
  | return  $\text{Some}(\text{ins});$ 
end
return  $\text{None};$ 

upgrade-trusted-contract( $\text{addr} : \text{address}, c : \mathcal{C}$ ) :  $\text{bool}$ 
if  $\text{verify-upgrade}_{\sqsubseteq}(s, c)$  for  $s \equiv \text{registry}[\text{addr}]$  then
  |  $\text{upgrade-upgradable-contract}(\text{addr}, c);$ 
  | return  $\text{true};$ 
end
return  $\text{false};$ 

```

---

not have built-in support for contract upgrades. Function *create-contract* creates an upgradable instance of contract  $c \in \mathcal{C}$  - it returns the Ethereum address where the instance was created - whereas *upgrade-contract* allows for the contract's behaviour to be upgraded. The specification used for a successful contract creation will be stored and used as the intended specification for future upgrades. Only the creator of a *trusted contract* can update its implementation.

Note that once a contract is created via our trusted deployer, the instance's specification is fixed, and not only its initial implementation but all upgrades are guaranteed to satisfy this specification. Therefore, participants in the ecosystem interacting with this contract instance can be certain that its behaviour is as intended by its developer during the instance's entire lifetime, even if the implementation is upgraded as the contract evolves.

### 3.2 VERIFIER

We propose *design by contract* (MEYER, 1992) as a methodology to specify the behaviour of smart contracts. In this traditional specification paradigm, conceived for object-oriented languages, a developer can specify invariants for a class and pre-/postconditions for its methods. These elements define a *specification contract* between the code using this class and the class implementation itself. Invariants must be established by the constructor and guaranteed by the public methods, whereas postconditions are ensured by the code in the method's body

provided that the pre-conditions are guaranteed by the caller code and the method terminates - we leave total correctness to be addressed in future implementations of our trusted deployer. We propose a specification format that defines what the member variables and signatures of member functions should be. Additionally, the function signatures can be annotated with postconditions, and the specification with invariants; these annotations capture the expected behaviour of the contract. In ordinary programs, a function is called in specific *call sites* fixed in the program's code. Pre-conditions can, then, be enforced and checked in these call sites. In the context of public functions of smart contracts, however, any well-formed transaction can be issued to invoke such a function. Hence, we move away from preconditions in our specification, requiring, thus, postconditions to be met whenever public functions successfully terminate.

Figure 6 – ToyWallet specification.

```

1  /**
   * @notice invariant accs[address(this)] == 0
   */
3  /**
   contract ToyWallet {
5      mapping (address => uint) accs;

7      /**
       * @notice postcondition forall (address addr) accs[addr] == 0
       */
9      constructor() public;

11     /**
13     * @notice postcondition address(this).balance == __verifier_old_uint(address(
        this).balance) + msg.value
       * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
        sender]) + msg.value
15     * @notice postcondition forall (address addr) addr == msg.sender ||
        __verifier_old_uint(accs[addr]) == accs[addr]
       */
17     function deposit () payable public;

19     /**
       * @notice postcondition address(this).balance == __verifier_old_uint(address(
        this).balance) - value
21     * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
        sender]) - value
       * @notice postcondition forall (address addr) addr == msg.sender ||
        __verifier_old_uint(accs[addr]) == accs[addr]
23     */
25     function withdraw (uint256 value) public;
   }

```

Figure 6 illustrates a specification for the *ToyWallet* contract. Invariants are declared in a comment block preceding the contract declaration, and function postconditions are declared in comment blocks preceding their signatures. Our specification language reuses constructs from Solidity and the *solc-verify* specification language, which in turn borrows elements from the Boogie language (BARNETT et al., 2005; LEINO, 2008). Member variables and function

signature declarations are as prescribed by Solidity, whereas the conditions on invariants and postconditions are side-effect-free Solidity expressions extended with quantifiers and the `__verifier_old_x(v)` expression which can only be used in a postcondition, and it denotes the value of  $v$  in the function's execution pre-state.

We target Solidity as the implementation language as it is arguably the most popular language used to create smart contracts. We choose to use Solidity as opposed to EVM bytecode as it gives a cleaner semantic basis for the analysis of smart contracts (ANTONINO; ROSCOE, 2021a) and it also provides a high-level error message when the specification is not met. The satisfiability relation  $\sqsubseteq$  that we propose is as follows.

**Definition 1.** The relation  $s \sqsubseteq c$  holds iff:

- *Syntactic obligation:* a member variable is declared in  $s$  if and only if it is declared in  $c$  with the same type, and they must be declared in the exact same order. A public function signature is declared in  $s$  if and only if it is declared and implemented in  $c$ .
- *Semantic obligation:* invariants declared in  $s$  must be respected by  $c$ , and the implementation of functions in  $c$  must respect their corresponding guards and postconditions described in  $s$ .

The purpose of this work is not to provide a formal semantics to Solidity or to formalize the execution model implemented by the Ethereum platform. Other works propose formalizations for Solidity and Ethereum (HAJDU; JOVANOVIĆ, 2020b; ANTONINO; ROSCOE, 2020; WANG et al., 2020). Our focus is on using the modular verifier *solc-verify* to discharge the semantic obligations imposed by our satisfaction definition.

The *verify-creation* $\sqsubseteq$  function works as follows. Firstly, the syntactic obligation imposed by Definition 1 is checked by a syntactic comparison between  $s$  and  $c$ . If it holds, we rely on *solc-verify* to check whether the semantic obligation is fulfilled. We use what we call a *merged contract* as the input to *solc-verify* - it is obtained by annotating  $c$  with the corresponding invariants and postconditions in  $s$ . If *solc-verify* is able to discharge all the proof obligations associated to this merged contract, the semantic obligations are considered fulfilled, and *verify-creation* $\sqsubseteq$  succeeds.

Function *verify-upgrade* $\sqsubseteq$  is implemented in a very similar way but it relies on a slightly different satisfiability relation and merged contract. While *verify-creation* $\sqsubseteq$  checks that the obligations of the constructor are met by its implementation, *verify-upgrade* $\sqsubseteq$  simply *assumes*

they do. The constructor is only executed - and, therefore, its implementation checked for satisfiability - at creation time. The upgrade process assumes that the constructor's obligations were met, and checks the implementation of the (non-constructor) public functions. The merged contract used in this verification process, then, has a constructor that has the expected signature and specification annotations but whose body has only statement *require(false)*; this statement indicates to *solc-verify* that it does not need to validate the implementation of the constructor.

### 3.3 UPGRADER

Ethereum does not provide a built-in mechanism for upgrading smart contracts, since the code of smart contracts is immutable once it is deployed on the blockchain. However, there are approaches one can simulate this functionality. The first is to deploy every new patched contract to the blockchain and migrate the state of the original contract to it. Nevertheless, this strategy has some drawbacks since it requires access to all the internal state of the old contract. In addition, the state migration between the contracts must be implemented manually (RODLER et al., 2021a). A more feasible approach would be the *proxy pattern* (OPENZEPELIN, 2021).

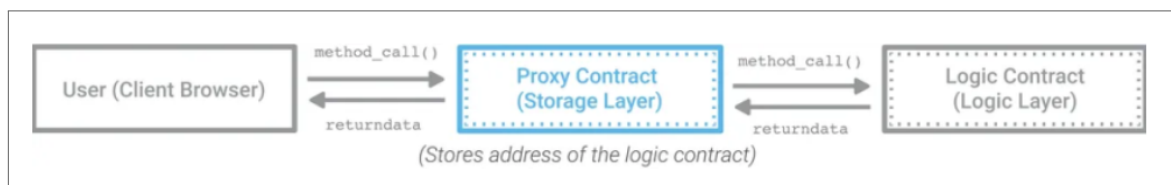
This design pattern splits the contract across two instances: the proxy instance and the implementation instance. The *proxy instance* holds the persistent state and the upgrade logic, relying on the code in an *implementation instance* which is stateless and contains all business logic. The proxy instance is the *de-facto* instance, that is, it should be the target of calls willing to execute the upgradable contract, therefore it should be immutable. It also stores the address of the implementation instance it relies upon, and the interface of the proxy's public functions can be upgraded by changing this address. Our *upgrader* relies on this pattern to deploy *upgradable* contracts.

We illustrate how to create a proxy using the *ToyWalletProxy* contract in Figure 8. Given a contract *c* that meets its specification according to Definition 1, the upgrader creates the Solidity contract *proxy(c)* as follows. It has the same member variable declarations, in the same order, as *c* - having the same order is an implementation detail that is necessary to implement the sort of delegation we use as it enforces proxy and implementation instances to share the same memory layout. In addition to those, it has a new *address* member variable called *implementation* - it stores the address of the implementation instance. The constructor of

$proxy(c)$  extends the constructor of  $c$  with an initial setting up of the variable *implementation*. This proxy contract also has a public function *upgrade* that can be used to change the address of the public-functions-implementation instance. The trusted deployer is identified by a trusted Ethereum address  $addr_{td}$ ; see, for instance, Line 27 in *ToyWalletProxy*. This address is used to ensure calls to *upgrade* can *only* be issued by the trusted deployer. In the process of creating and upgrading contracts the trusted deployer acts as an external participant of the Ethereum platform. We assume that the contract implementations and specifications do not have member variables named *implementation*, or functions named *upgrade* and *\_upgrade* to avoid name clashes.

The proxy instance relies on the low-level *delegatecall* Solidity command to dynamically borrow and execute the function implementations defined in the contract instance at *implementation*. When the contract instance at address *proxy* executes *implementation.delegatecall(sig, args)*, it executes the code associated with the function with signature *sig* stored in the instance at address *implementation* but applied to the *proxy* instance - modifying its state - instead of *implementation*. For instance, in Figure 8, we have that the *ToyWalletProxy* functions *deposit* and *withdraw* borrow and execute the code of functions with the same signature in the instance at address *implementation*. By using this feature, we can change the behaviour of the proxy's public functions by switching *implementation* addresses with the new desired implementation. For each (non-constructor) public function in  $c$  with signature *sig*,  $proxy(c)$  has a corresponding function declaration whose implementation relies on *implementation.delegatecall(sig, args)*. This command was proposed as a means to implement and deploy contracts that act as a sort of dynamic library. Such a contract is deployed with the sole purpose of other contracts borrowing and using their code. The contract  $proxy(c)$  will also include any internal function that is needed by the constructor. Lastly, the function on the implementation contract returns either the data or throws an exception if something went wrong. The proxy architecture pattern is shown in Figure 7.

Figure 7 – Proxy Structure



Source: NADOLINSKI; SPAGNUOLO (2018)

The upgrader function  $create-upgradable-contract(c)$  behaves as follows. Firstly, it issues transaction  $create-contract(c, args)$  to the Ethereum platform to create the initial implementation instance at address  $addr_{impl}$ . Secondly, it issues transaction  $create-contract(proxy(c), args)$ , such that *implementation* would be set to  $addr_{impl}$ , to create the proxy instance at address  $addr_{px}$ . We illustrate this process with our *ToyWallet* contract in Figure 3. For a call,  $create-upgradable-contract(ToyWallet, args)$ , our upgrader would issue transaction  $create-contract(no-constructor(ToyWallet, args)$ , which would create the constructor-less public-functions-implementation instance at address  $addr_{tw}$ . Then it would proceed to issue transaction  $create-contract(ToyWalletProxy, args)$  where  $args$  binds parameter *\_implementation* to  $addr_{tw}$  and, of course,  $proxy(ToyWallet)$  is *ToyWalletProxy*. For the sake of simplicity, we assume that these contract creations cannot fail. Note that both of these transactions are issued by and using the trusted deployer's address  $addr_{td}$ . The upgrader function  $upgrade-upgradable-contract(c)$  behaves similarly, but the second step issues transaction  $call-contract(addr_{px}, upgrade, args)$ , triggering the execution of function *upgrade* in the proxy instance and changing its *implementation* address to the new implementation instance.

Given the behaviour of the trusted deployer and especially its restriction on creation and updates as far as ensuring satisfiability between specification and implementation, we make the following claim.

Let  $\mathcal{I}$  be an invariant implied by the specification  $s$  - that is, a predicate implied by the *invariant* definitions or by each of the post-conditions on public functions - and  $c$  an upgradable (proxy) contract instance with specification  $s$  that has been created and managed by the trusted deployer. For a persistent state  $st$  of this contract instance, i.e., one that can be reached from the zero-initialised persistent state via a non-empty sequence of calls starting with the constructor and followed by public functions, it must be the case that:

1.  $\mathcal{I}$  holds for state  $st$ ; and
2. for any public function  $f$  in  $c$  with post-condition  $\mathcal{P}$  in  $s$ , any state  $st'$  reached after a successfully terminating execution of  $f$  from  $st$  must respect  $\mathcal{P}$ .



Figure 8 – ToyWallet proxy

```

1 contract ToyWalletProxy {
2     mapping (address => uint) accs;
3     address implementation;
4
5     function deposit () payable public {
6         (bool success, bytes memory _) = implementation.delegatecall(
7             abi.encodeWithSignature("deposit()"));
8         require(success);
9     }
10
11    function withdraw (uint256 value) public {
12        (bool success, bytes memory _) = implementation.delegatecall(
13            abi.encodeWithSignature("withdraw(uint256)", value));
14        require(success);
15    }
16
17    constructor(/* contract's original constructor parameters, */ address
18        _implementation) public {
19        /* contract's original constructor code */
20        _upgrade(_implementation);
21    }
22
23    function upgrade(address new_implementation) public {
24        _upgrade(new_implementation);
25    }
26
27    function _upgrade(address new_implementation) internal {
28        require(msg.sender == addrtd);
29        implementation = new_implementation;
30    }
31 }

```

### 3.4 REGISTRY

An external participant in the Ethereum platform can confirm that a given instance was created by the trusted deployer by calling its *get-spec* function. Smart contracts in the Ethereum platform, however, cannot probe external services, such as the trusted deployer. Therefore, an Ethereum smart contract would have no means to check whether an instance that it wants to interact with is safe. As *composability*, namely, this ability of smart contracts to interact and cooperate, is one of the main features of Ethereum, we propose a mechanism by which contracts can programmatically test if a counterpart contract is safe. As part of the trusted deployer infrastructure, we create a *trusted registry*. It is essentially a mirror of the trusted deployer's internal registry implemented as an Ethereum smart contract. This trusted registry has other benefits in its own right. For instance, given its implementation as an Ethereum smart contract, it inherits the non-function properties offered by this platform such as high-

availability and secure execution.

Figure 9 – Trusted registry

```

2  contract TrustedRegistry {
    mapping (address => bytes32) verified_addrs;
4  address maintainer;

6  constructor() public {
    maintainer = msg.sender;
8  }

10 function new_mapping(address addr, bytes32 spec_id) public {
    if (msg.sender == maintainer && spec_id != bytes32(0)) {
12         verified_addrs[addr] = spec_id;
    }
14 }

16 function get_spec(address addr) view public returns (bytes32) {
    return verified_addrs[addr];
18 }
}

```

In the process of setting up the trusted deployer, the *TrustedRegistry* contract in Figure 9 is created as part of its infrastructure at the well-known trusted address `addrreg`. It has mapping *verified\_addr* that associates the proxy instances that have been created by the trusted deployer with the specification they comply to. As an implementation detail, we do not store the specification themselves but rather a small representative as a 32-byte array - it could be, for instance, a cryptographic hash of the specification. We do not allow this representative to be the zeroed array, *bytes32*(0) in Solidity syntax, as we use this value to represent absence of an association. That is, if the result of a call *get\_spec(addr)* is the value *bytes32*(0), it means the address *addr* has not been deployed by the trusted deployer, and hence has no specification associated with it.

Having this trusted registry might seem a small contribution but it brings significant guarantees that cannot be otherwise obtained. In Solidity, and generally in Ethereum, *type* information cannot be programmatically obtained by smart contracts. For instance, when executing a function call, a smart contract cannot check whether the target contract has code associated with the specific function it is trying to call. It simply tries to execute the code associated with a function signature. Solidity's behaviour can be especially problematic in this aspect. They have an implicit execution flow by which a special *fallback* function is executed if none of the other functions in the target contract have the intended signature. Thus, a smart contract

might execute a function on a counterpart that apparently successfully terminates when in fact a completely different unwanted function runs instead. Our trusted deployer acts as a *verification oracle* that pushes into the blockchain, and specifically into the *TrustedRegistry* contract, information not only about the interface of deployed contracts, ensured by the syntactic obligation in Definition 1, but also about their public functions behaviour, ensured by the semantic obligation on the same definition. This information can be programmatically obtained via the *get\_spec* function in *TrustedRegistry*, of course. Amongst other usages, this ability can be harnessed to create a sort of safe contract call. A contract can check that the to-be-called counterpart meets a certain specification and only issue the call if so. For instance, let us assume that we want to write a smart contract function that calls *do\_something* in contract *c*, but only does so if *c* meets specification *expected\_spec*. This behaviour could be achieved by the following snippet:

Figure 10 – Trusted Deployer behaviour

```
1 requires(TrustedRegistry(addrreg).get_spec(address(c)) == expected_spec);
   c.do_something(a, b);
```

A similar approach can be taken by external applications that want to use a safe smart contract. They can instead use the function *get\_spec* of the trusted deployer, and abort their execution if the contract they want to interact with does not meet the expected specification.

## 4 TOOL SUPPORT AND APPLICATION TO A REAL COMMIT HISTORY

This chapter presents the implementation of the main functionalities of the prototype tool that has been developed (Section 4.1), particularly, the smart contract safe creation (Section 4.2) and upgrade (Section 4.3) features. We also present an example of the complete framework in a real development scenario. First we describe the research context, next we scrutinize a scenario based on the 0xMonorepo repository in order to analyse possible occurrences of bugs found in the commit history (Section 4.4). In the next chapter we present a more detailed evaluation of the proposed verification framework.

### 4.1 TRUSTED DEPLOYER TOOL

We developed the *Trusted Deployer*<sup>1</sup> prototype which was built on top of the modular verifier *solc-verify* and it has three main features: creation, verification and upgrading of smart contracts. Its main purpose is to facilitate the work of developers during the development life cycle, so we provide a clear and simple interface to support the use of the proposed framework.

The *Trusted Deployer* was created based on the pattern Representational State Transfer (REST) an architectural style that can be used as a basis for a platform-independent HTTP-based service interface (SUNDVALL et al., 2013). This pattern brought two main benefits to the project: scalability and flexibility, which means that our application can be scaled quickly due to the separation between the client and the server. Model-View-Controller (MVC), a programming paradigm that separates an application into three separate classes: the modeling of the domain, the presentation of information and the business logic (KRASNER; POPE, 1988). With the use of this architecture, we were able to increase the efficiency of the development process, making it simpler. Furthermore, possible changes, whether due to maintenance or the emergence of new requirements, should not affect the entire application. Its back-end was developed using the programming language Rust (version 1.60.0), the Rocket web framework (version 0.5) which eliminate different types of memory-related bugs at compile time. In addition, Rust is extremely fast, secure and reliable. The database management system is PostgreSQL (version 14.1) a secure, robust and open source relational database. We also use open-source libraries to interact with the smart contracts deployed in the Ethereum network

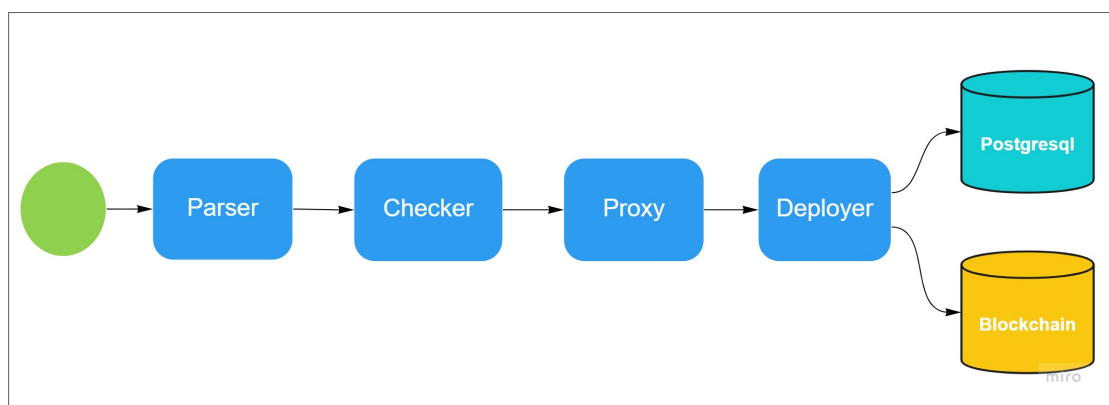
<sup>1</sup> The prototype as well as all instructions for its configuration and use available at <<https://github.com/formalblocks/safeevolution>>.

(web3.js). The front-end was developed using the programming language JavaScript and the React framework (version 17.0.1) which facilitates the process of writing components and facilitates further maintenance and it boosts productivity. We used Git and Github as a version control tool and repository respectively, for both projects. During the studies to formulate the prototype, the need for a secure platform for safe creating and updating smart contracts was identified. From this perspective, the software architecture must satisfy the following requirements:

- **Security:** To create or update a smart contract, it is necessary to create a digital wallet; the agent who wants to use the platform must have public and private keys and all transactions must be signed.
- **Inviolability:** All transactions must be recorded on the blockchain so that smart contracts cannot be tampered with.

Although this work has focused on the Ethereum blockchain because it is widely used, meets the requirements raised for the project and enables the execution of the project without having to allocate large infrastructure resources, it is compatible with any network that uses EVM as their default smart contract engine like Fantom, Binance Smart Chain, Tron and Celo, to mention a few. Figure 11 presents an overview of the tool considering its process workflow and all the modules that make it up. The figure considers only the modules but it will be shown the details of each module and how they adhere to the conceptual bases presented in Chapter 3.

Figure 11 – Trusted Deployer Architecture



Source: The author

### *Parser*

The specification and implementation smart contracts are imported into the tool through files with the .sol extension, then we generate, using the official Solidity compiler, a JSON object with the AST (Abstract Syntax Tree) that is an intermediate language with a graph representation of source code. The transformation from source code to an AST is an important step to create typed in-memory objects that can be manipulated programmatically. Our custom parser helps to analyze the structures in the smart contracts code, create merged smart contracts and perform static analysis.

### *Checker*

It is the module responsible for carrying out all the validations defined for the deployment and evolution of smart contracts. First, we check the syntactic conformance between an implementation and the associated specification smart contracts, that is, whether the signatures of the public functions contained in the contracts are compatible, in addition all the variables as well as their respective types contained in the specification for the declaration of invariants and postconditions must also be the same as those the implementation contract. According to the established rules, the user is not allowed to declare a function with the *upgrade* identifier as it is already defined by the proxy for updating the addresses of the implementation smart contracts.

The user inputs the constructor variables through the web interface, so it is necessary to convert the javascript types to their respective solidity equivalent types. If it is an upgrade operation, it is not allowed to declare constructors in the implementation smart contract. Finally, we generate the merged smart contract, perform the syntactic analysis and collect the results for user feedback. The deployment or upgrade process will not proceed if any of the mentioned requirements are not respected.

### *Proxy*

This module is responsible for setting-up the proxy contract architecture as discussed in Section 3.3. This is possible by separating the logic that stores state and business rules from the implementation contract. For the proxy configuration it is necessary to identify and extract

the signatures of public functions and variables as well as configuring the parameters and the body of the constructor and fill the proxy template.

### *Deployer*

After performing all the previous steps, the *Trusted Deployer* deploys the register, proxy and logic smart contracts on the blockchain in addition to all the addresses of the mentioned smart contracts; the specification file and author are saved in a database. This procedure improves the visualization and manipulation of data by users, as the details of the verification process can be analyzed.

## 4.2 SMART CONTRACT CREATION PROCESS

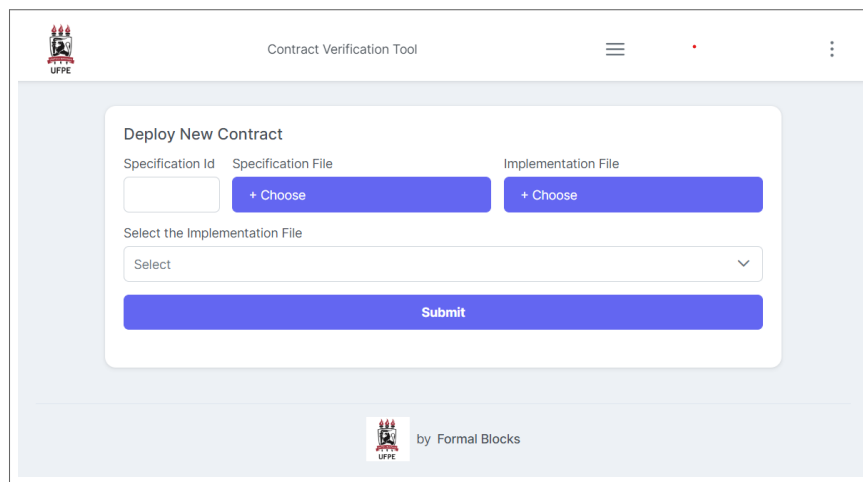
In order to access the features offered by the *Trusted Deployer* tool all users must have a crypto wallet, that is, a tool that allows them to manage their assets. They also should acquire a balance in cryptocurrency (in the case of Ethereum, ETH) in order to pay the transaction costs to the network. The public and private keys will be used to carry out the transactions and it will also ensure that only the user with the appropriate permissions has access to the smart contract. To increase the security of the tool, we do not store or manage the user's private key, instead we use a non-custodial crypto wallet so the user is the only one in control of the funds and contracts deployed. When setting up a non-custodial wallet, a user gets a mnemonic phrase of 12 to 24 words which should be stored somewhere safe. The seed phrase can be used to regain the wallet access.

Although there are different types of non-custodial like the physical ones that can be stored in hardware devices or software that can be installed in a computer, we choose the browser-based type to promote a better experience to users. There are several options, including add-ons in popular browsers like Google Chrome or Mozilla Firefox and even browsers with native wallets like Brave. In this case, the *Trusted Deployer* tool contains a specific page with instructions that must guide users to create and manage a wallet and purchase cryptocurrencies.

To understand the implementation challenges for registering contracts on the blockchain, it is necessary to evaluate the processes related to the Trusted Deployer tool's deployer module. Figure 12 shows a screenshot of the creation smart contract process, according to the workflow, a user starts its operation by a *specification\_id* which is a unique identifier that will ensure for

all participants that interact with the smart contract that a given instance was created by the *Trusted Deployer* by calling its *get-spec* function; therefore it is formally verified and safe. The user must also upload the specification and implementation files. Solidity supports multiple inheritance, so multiple contracts can be inherited by a contract that is known as a derived contract. In this case the user must inform the derived contract to be verified and constructor parameters when they exist.

Figure 12 – Screenshot Create Smart Contract



The screenshot displays the 'Contract Verification Tool' interface. At the top, there is a header with the UFPE logo on the left, the title 'Contract Verification Tool' in the center, and a hamburger menu icon on the right. Below the header, a central white card titled 'Deploy New Contract' contains the following elements:
 

- A 'Specification Id' input field.
- A 'Specification File' section with a '+ Choose' button.
- An 'Implementation File' section with a '+ Choose' button.
- A 'Select the Implementation File' dropdown menu with 'Select' as the current option and a downward arrow.
- A large blue 'Submit' button at the bottom of the card.

 At the bottom of the interface, there is a footer with the UFPE logo and the text 'by Formal Blocks'.

Source: The author

### 4.3 SMART CONTRACT UPGRADING PROCESS

Considering the good practices to handle the wallet described in the previous section, we defined a procedure to upgrade smart contracts (Figure 13). From the moment a connection is established between the wallet and the *Trusted Deployer*, the user has access to the historical data generated from the smart contract deployment. After input of *specification\_id* and implementation smart contract, the tool must check if there is a record of a corresponding specification; if there is, the update process will occur automatically. In this case only the implementation smart contract is sent to the blockchain and its address is registered in the proxy.



Figure 13 – Screenshot Upgrade Smart Contract

The screenshot shows a web application titled "Contract Verification Tool". At the top left is the UFPE logo. The main content area is titled "Upgrade a Contract". It contains two input fields: "Specification Id" and "Implementation File". Next to the "Implementation File" field is a blue button labeled "+ Choose". Below these fields is a dropdown menu labeled "Select the Implementation File" with the word "Select" inside. At the bottom of the form is a large blue button labeled "Submit". The footer of the page contains the UFPE logo and the text "by Formal Blocks".

**Source: The author**

#### 4.4 TOOL APPLICATION TO A SMART CONTRACT COMMIT HISTORY

After discussing the results of the smart contract verification process, we introduce a scenario based on the 0xMonorepo<sup>2</sup> repository which implements the ERC20 pattern and is one of the repositories used during our study. We analysed the whole commit history (see Table 1) in order to find errors or nonconformities and describe what the history of the repository would be if a developer had used our tool since invalid commits would be prevented from being deployed. The idea is to show how our tool implements the architecture depicted in Figure 11; we also demonstrate that the tool supports a safe smart contract development process.

As already explained, the trusted deployer requires that developers have their code formally verified before they can deploy their contracts to a blockchain network. In order to create a smart contract, a developer has to provide its code and specification together. Each specification linked to the created smart contract will be assigned a unique identifier that must be referenced in subsequent updates. The tool uses the solc-verify in background to verify the code against its specification before it proceeds to deploy the smart contract. Figure 14 presents a merged contract, which is the result of the merging of the specification (see Figure 16) and an implementation contract. The verification contract is automatically created from the abstract syntax tree of the contracts after a syntactic check is performed; in this case, the goal is to analyse if there is any discrepancy between the signature of the functions and the data model of the specification and implementation contracts.

The proxy pattern allows the implementation contract to be replaced while the trusted

<sup>2</sup> The 0xMonorepo repository can be found at <<https://github.com/formalblocks/safeevolution>>.

0xMonorepo Repository											
Commit	Time	Output	Commit	Time	Output	Commit	Time	Output	Commit	Time	Output
<a href="#">7d59fa</a>	3.05s	WOP	<a href="#">bb4c8b</a>	2.20s	No errors	<a href="#">89abd7</a>	3.18	No errors	<a href="#">99fbf3</a>	2.90s	No errors
<a href="#">7008e8</a>	3.25s	WOP	<a href="#">897515</a>	3.11s	No errors	<a href="#">ba1485</a>	3.47s	No errors	<a href="#">9b521a</a>	3.45s	No errors
<a href="#">b58bf8</a>	2.93s	WOP	<a href="#">32fead</a>	3.78s	No errors	<a href="#">f21b04</a>	3.32s	No errors	<a href="#">0758f2</a>	3.73s	No errors
<a href="#">548fda</a>	2.85s	WOP	<a href="#">d11811</a>	4.01s	No errors	<a href="#">d2e422</a>	3.51s	No errors	<a href="#">d35a05</a>	3.56s	No errors
<a href="#">6f2cb6</a>	3.70s	No errors	<a href="#">1729cf</a>	3.28s	No errors	<a href="#">a2024d</a>	2.70s	No errors	<a href="#">5813bb</a>	4.20s	No errors
<a href="#">145fea</a>	3.10s	No errors	<a href="#">63abf3</a>	3.44s	No errors	<a href="#">bb3c34</a>	3.41s	No errors	<a href="#">01aeec</a>	3.21s	No errors
<a href="#">1fb643</a>	3.83s	No errors	<a href="#">c84be8</a>	3.20s	No errors	<a href="#">f54591</a>	3.97s	No errors	<a href="#">272125</a>	3.67s	No errors
<a href="#">5198c5</a>	2.50s	No errors	<a href="#">8bce73</a>	2.79s	No errors						

Table 1 – 0xMonorepo Commit History

proxy (see Figure 15) or the access point is never changed. Both contracts are still immutable in the sense that their code cannot be changed, but the logic contract can simply be swapped by another contract. Every proxy must contain all variables (lines 3 to 5) defined in the implementation contract which will store the state as well as the signatures of all its public functions (lines 7 to 35). If there is any constructor in the implementation contract it will be merged with the default constructor of the proxy (lines 42 to 48).

During the analysis of the scenario, 30 commits were verified; it was observed the wrong operator (WOP) error in its first four commits [7d59fa](#), [7008e8](#), [b58bf8](#) and [548fda](#). If the developer had used our tool the error would have been discovered in the first analysis, these deployments would have been prevented, and an error message containing the specific reason would be returned to the developer forcing him to fix the bug. From the results collected from our evolution scenario, one can see that our strategy is effective in identifying errors in the early stage of the process. Our tool abstracts many details of the deployment and upgrade process, making it simpler for platform users when compared to the manual process.

Figure 14 – Merged ERC20 Contract

```

2  /** @notice invariant _totalSupply == __verifier_sum_uint(balances) */
    contract ERC20Token {

4      uint constant MAX_UINT = 2**256 - 1;
      mapping (address => uint) balances;
6      mapping (address => mapping (address => uint)) allowed;
      uint public _totalSupply;
8      event Transfer(address indexed _from, address indexed _to, uint _value);
      event Approval(address indexed _owner, address indexed _spender, uint _value);

10     /** @notice postcondition ((balances[msg.sender] == __verifier_old_uint (balances[msg.
        sender]) - _value && msg.sender != _to) || (balances[msg.sender] ==
        __verifier_old_uint (balances[msg.sender]) && msg.sender == _to) && success) || !
        success
12     * @notice postcondition ((balances[_to] == __verifier_old_uint (balances[_to]) + _value
        && msg.sender != _to) || (balances[_to] == __verifier_old_uint (balances[_to])
        && msg.sender == _to)) || !success
        * @notice emits Transfer */
14     function transfer(address _to, uint _value) public returns (bool success) {
        require(balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]);
16         balances[msg.sender] -= _value;
        balances[_to] += _value;
18         emit Transfer(msg.sender, _to, _value);
        return true; }

20     /** @notice postcondition (( balances[_from] == __verifier_old_uint (balances[_from]) -
        _value && _from != _to) || (balances[_from] == __verifier_old_uint (balances[
        _from]) && _from == _to) && success) || !success
22     * @notice postcondition ((balances[_to] == __verifier_old_uint (balances[_to]) + _value
        && _from != _to) || (balances[_to] == __verifier_old_uint (balances[_to]) && _from
        == _to) && success) || !success
        * @notice postcondition (allowed[_from][msg.sender] == __verifier_old_uint (allowed[
        _from][msg.sender]) - _value && success) || (allowed[_from][msg.sender] ==
        __verifier_old_uint (allowed[_from ][msg.sender]) && !success) || _from == msg.
        sender
24     * @notice postcondition allowed[_from][msg.sender] <= __verifier_old_uint (allowed[
        _from][msg.sender]) || _from == msg.sender
        * @notice emits Transfer */
26     function transferFrom(address _from, address _to, uint _value) public returns (bool
        success) {
        uint allowance = allowed[_from][msg.sender];
28         require(balances[_from] >= _value && allowance >= _value && balances[_to] + _value
            >= balances[_to]);
        balances[_to] += _value;
30         balances[_from] -= _value;
        if (allowance < MAX_UINT) {
32             allowed[_from][msg.sender] -= _value;
        }
34         emit Transfer(_from, _to, _value);
        return true; }

36     /** @notice postcondition (allowed[msg.sender][_spender] == _value && success) || (
        allowed[msg.sender][_spender] == __verifier_old_uint (allowed[msg.sender][_spender
        ]) && !success)
38     * @notice emits Approval */
    function approve(address _spender, uint _value) public returns (bool success) {
40         allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
42         return true; }

44     /** @notice postcondition balances[_owner] == balance */
    function balanceOf(address _owner) public view returns (uint balance) {
46         return balances[_owner]; }

48     /** @notice postcondition allowed[_owner][_spender] == remaining */
    function allowance(address _owner, address _spender) public view returns (uint
        remaining) {
50         return allowed[_owner][_spender]; }
}

```

Figure 15 – ERC20 Proxy

```

1  contract ERC20Proxy {
3      uint256 public _totalSupply;
      mapping(address => uint256) internal balances;
5      mapping(address => mapping(address => uint256)) internal allowed;

7      function approve (address _spender,uint256 _value) public returns (bool success) {
          (bool success, bytes memory bytesAnswer) = implementation.delegatecall(abi.
              encodeWithSignature("approve(address,uint256)",_spender,_value));
9          require(success);
              return abi.decode(bytesAnswer, (bool));
11     }

13     function transfer (address _to,uint256 _value) public returns (bool success) {
          (bool success, bytes memory bytesAnswer) = implementation.delegatecall(abi.
              encodeWithSignature("transfer(address,uint256)",_to,_value));
15         require(success);
              return abi.decode(bytesAnswer, (bool));
17     }

19     function transferFrom (address _from,address _to,uint256 _value) public returns (bool
        success) {
          (bool success, bytes memory bytesAnswer) = implementation.delegatecall(abi.
              encodeWithSignature("transferFrom(address,address,uint256)",_from,_to,_value));
21         require(success);
              return abi.decode(bytesAnswer, (bool));
23     }

25     function balanceOf (address _owner) public returns (uint256 balance) {
          (bool success, bytes memory bytesAnswer) = implementation.delegatecall(abi.
              encodeWithSignature("balanceOf(address)",_owner));
27         require(success);
              return abi.decode(bytesAnswer, ( uint256));
29     }

31     function allowance (address _owner,address _spender) public returns (uint256
        remaining) {
          (bool success, bytes memory bytesAnswer) = implementation.delegatecall(abi.
              encodeWithSignature("allowance(address,address)",_owner,_spender));
33         require(success);
              return abi.decode(bytesAnswer, ( uint256));
35     }

37     Registry registry;
      bytes32 spec;
39     address implementation;
      address author;

41     constructor(Registry _registry, bytes32 _spec, address _implementation ) public {
43         require(_spec != bytes32(0));
            registry = _registry;
            spec = _spec;
            author = msg.sender;
            _upgrade(_implementation);
47     }

49     function upgrade(address new_implementation) public {
        _upgrade(new_implementation);
51     }

53     function _upgrade(address new_implementation) internal {
        require(msg.sender == author);
        bytes32 spec_id = registry.get_spec(new_implementation);
55         require(spec_id == spec);
            implementation = new_implementation;
57     }
}

```

## 5 CASE STUDIES: ERC20, ERC1155, AND ERC3156

To validate our approach, we have carried out three systematic case studies of the ERC20 Token Standard, the ERC1155 Multi Token Standard, and the ERC3156 Flash Loans. We start with the definition of the context of our evaluation (Section 5.1) followed by the process that we have adopted (Section 5.2), and then consider each of the token standards (Sections 5.3 to 5.5); finally, we summarise the overall results of our verification effort in Section 5.6.

### 5.1 CONTEXT

The main purpose of the study is to provide evidence that the proposed framework can be used to ensure the deployment and evolution of smart contracts in a secure manner. The framework must be integrated into a development process and whenever there is any change in the code, one must verify if the change meets the specification, so we intend to answer the following research question: *Does the proposed framework improve on the process of creation and evolution of smart contracts?*

In the first phase, a review of the literature was carried out. The objective was to explore the main features of smart contract development patterns and the most common error types. As a result, we could identify opportunities for the application of the framework in line with the objectives of the study. In the second phase, we identified, documented and validated requirements related to three Ethereum smart contract specifications: ERC20, ERC3156 and ERC1155. Even though these standards were developed by the ethereum community as a way of organizing and disseminating knowledge among stakeholders, there are still problems related to the use of natural language since it is inherently ambiguous; in addition there is a great difficulty in maintaining traceability between requirements and code.

Each of these standards defines a contract interface and is accompanied by an informal description of its functions' behaviours. We chose these standards because they are widely used and in an advanced state of maturity. By structuring the existing requirements in natural language, we were able to extract the formal properties used in the verification process and create a corresponding formal contract specification. Then, we conducted a quantitative analysis, in order to verify the feasibility of the framework, to evaluate its effectiveness, which can be measured by the number of errors found or safe evolutions that have been proven correct,

and efficiency, measured by the time to process the verification.

The objective is to verify whether the correctness and consistency of the properties specified in smart contracts are met by the implementation, and thus determine whether the evolution process was valid or not. First, we transform requirements written in natural languages, into formal specifications; this step will give us all invariants and postconditions needed for analysis. Next, we construct a model which consists of an abstract smart contract with the signature of all the methods that integrate the respective standard, data structures, and all the properties to be verified. Then using the *solc-verify* tool we verify the behaviour of the smart contract through its evolution process; the main objective of this step is to ensure that the upgrade of smart contracts is not carried out arbitrarily but based on a notion of safe evolution, that is, the properties must be preserved regardless of the change in the code. Also, we analyse the efficiency of our methodology. We measure the execution time, in seconds and whether or not the analysis came up with any result.

To validate our approach, we have carried out three systematic case studies of the ERC20 Token Standard, the ERC1155 Multi Token Standard, and the ERC3156 Flash Loans. For the ERC20, we examined 8 repositories and out of 32 commits analysed, our framework identified 8 unsafe commits, in the sense that they did not conform to the specification; for the ERC1155, we examined 4 repositories and out of 18 commits analysed, 5 were identified as unsafe; and for the ERC3156, we examined 5 repositories and out of 18 commits analysed, 7 were identified as unsafe. The contract samples we analysed were extracted from repositories that were public, and presented reasonably complex commit histories that changed the smart contract behavior. The samples also cover aspects of evolution that are related to improving the readability and maintenance of the code, but also optimisations where, for instance, redundant checks executed by a function were removed. The evaluation was carried out on a Lenovo IdeapadGaming3i with the operating system Windows 10, Intel(R) Core(TM) i7-10750 CPU @ 2.60GHz, 8GB of RAM, with Docker Engine 20.15.5.

In order to choose the verifier that would be part of the *Trusted Deployer* and answer research questions, we compiled a comprehensive list of tools for analyzing smart contracts by checking the publications of papers, public github repositories and forums. We investigate 8 tools for static analysis Ethereum smart contracts regarding availability, maturity level and detection of security issues. Although one of the main features of the *Trusted Deployer* is its composability and can be used with a diverse variety of verifiers we choose *Solc-Verify* with Solidity compiler version 0.5.17. This version, in addition to being stable, proved to be quite

efficient in the analysis of smart contracts. It is important to note that *Solc-Verify* works at the source level, and allows users to specify high-level properties such as contract invariants, loop invariants, pre- and post-conditions and assertions. In addition, it can effectively find bugs and prove correctness of non-trivial properties with minimal user effort.

Our framework was able to identify errors of the following categories<sup>1</sup>: Integer Overflow and Underflow (IOU); Non Specification Conformance (NSC), when a function does not meet a specific mandatory requirement defined in its ERC specification; Nonstandard Token Interface (NTI), when the contract does not meet the syntactic restriction defined by the standard; wrong operator (WOP), for instance, when the  $<$  operator would be expected but  $\leq$  is used instead; and Verification Error (VRE), when the verification process cannot be completed or the results are inconclusive. It also established conformance for some of the samples analysed.

## 5.2 PROCESS OVERVIEW

We first analyze standard natural language interfaces of the Ethereum platform, also known as Ethereum Improvement Proposals, which are standards developed by the Ethereum community to specify potential new features or processes (JAMESON, 2022). Although EIPs are a concise way of organizing standards and disseminate the knowledge among the stakeholders, there are still problems related to the use of natural language, because they are inherently ambiguous. All EIPs chosen for the study are part of Ethereum Request for Comments category and are in final update status which indicates a high degree of maturity regarding the discussions on the standard. The specifications for each ERC are expressed in accordance with the terms of the standard RFC 2119, which defines several keywords "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*MAY*" and "*OPTIONAL*" that helps describe the requirements, guide the understanding about it and extract the formal specifications.

We then construct a model with specifications and functional properties; the model consists of abstract smart contracts with the signature of all the public and external functions that integrate the respective interfaces, data structures, and all the properties to be verified expressed as invariants and postconditions. For functions with access modifiers of the private and internal type, they are not part of the specification as they only indirectly contribute to the

<sup>1</sup> All specifications, sample contracts, and scripts used in this evaluation can be found at <https://github.com/formalblocks/safeevolution>.

observable behaviour of a contract. These functions are being indirectly verified when they are called within public function. The verification of the postconditions of these public functions must take into account the behaviour/execution of called private functions.

Then using the *solc-verify* tool we verify the behaviour of smart contracts through their evolution process; the main objective of this step is to ensure that the initial deploy and subsequent upgrades of smart contracts are based on a notion of safe evolution, which means that each implementation must satisfy the properties in the specification. Finally, assuming that the verification was successful, the smart contract implementation is then deployed on the blockchain. Due to the fact that smart contracts are immutable after deployment, we make use of the proxy pattern which allowed us to decouple a contract's state and business rules which helps prevent the loss of relevant information during the safe evolution process.

Although it is not in our scope to define an automated translation from informal requirements to formal requirements, we have followed a systematic approach. Inter-contract communication and events are core features of EVM and many bugs may arise from their uses, they were not on the scope of the project therefore the specifications were not included in the model even though they were mandatory as defined in RFC 2119.

Modifiers in Solidity are used to change the behaviour of functions. This mechanism provides a means for structural and behavioral features that apply to different parts of the smart contract to be implemented separately, preserving cohesion and reducing complexity. They create additional features or apply some restriction on functions and can be executed before or after the original function call. The segregation of these properties makes it possible to modularize the code properly and then compose it according to needs. It is not possible to verify modifiers since they are not functions, so all properties related to the behavior of modifiers have been transformed into postconditions added to the functions. In order to create a specification that can be verified for a wide range of smart contracts, we wrote specifications only for the mandatory methods.

### 5.3 ERC20

Proposed in 2015 in order to establish a common interface for fungible tokens, ERC20 is one of the first standards defined for the Ethereum platform, and it is also one of the most popular, with thousands of active tokens deployed on the main Ethereum network and daily transaction volume worth billions of dollars. ERC20 effectively allows smart contracts to act



very similarly to native cryptocurrency like Bitcoin.

It defines member variables: *totalSupply* keeps track of the total number of tokens in circulation, *balanceOf* maps a wallet (i.e. address) to the balance it owns, and *allowance* stores the number of tokens that an address has made available to be spent by another one. It defines public functions: *totalSupply*, *balanceOf* and *allowance* are accessors for the above variables; *transfer* and *transferFrom* can be used to transfer tokens between contracts; and *approve* allows a contract to set an "allowance" for a given address.

Figure 16 presents a specification, derived from the informal description in the standard (VOGELSTELLER; BUTERIN, 2015). In Line 1, we define an invariant requiring that the total number of tokens remain unchanged regardless of the operation carried out by the contract. The functions *totalSupply*, *balanceOf* and *allowance* do not change the state of the smart contract so each function has only one postcondition (lines 8, 6 and 14) to ensure that the expected information will be returned successfully.

The *transfer* function has 3 postconditions; the operation is successful only when the tokens are debited from the source account and credited in the destination account, according to the specifications provided in the ERC20 standard. The postconditions (lines 17 to 18) require that the balances are updated as expected and the postcondition in line 19 will check whether or not the balances of the addresses not involved in the transaction will remain unchanged. The *transferFrom* function automates the transfer process and allows sending a given amount of tokens on behalf of the owner, so we added two more postconditions (lines 25 to 26) in addition to those defined for the *transfer* function to ensure that the tokens available for withdrawal have been successfully debited when a transfer occurs; besides, it is necessary to verify if the money to be withdrawn is less than or equal to the amount allowed. When these functions are successful, the balance of both addresses will be updated within the contract, constituting a state change. The ERC20 standard recommends to emit a *Transfer* event whenever a token transfer has occurred (lines 20 and 28) (VICTOR; LÜDERS, 2019). Finally, on the *approve* function (line 31), we defined one postcondition to make sure the operation is successful always when a credit is available, the specification states that the *Approval* event must be issued (line 32). Table 2 shows the complete list of all ERC20 results we obtained.

## ERC20 Analysis

We use the snippet in Figure 17 - extracted from the Uniswap repository, commit [55ae25](#) - to illustrate the detection of wrong operator errors. When checked by our framework, the

Figure 16 – ERC20 specification

```

2  /** @notice invariant _totalSupply == __verifier_sum_uint(balances) */
   contract ERC20 {
4     mapping (address => uint) balances;
   mapping (address => mapping (address => uint)) allowed;
6     uint public _totalSupply;

8     /** @notice postcondition supply == _totalSupply */
   function totalSupply() public view returns (uint256 supply);

10    /** @notice postcondition balances[_owner] == balance */
   function balanceOf(address _owner) public view returns (uint balance);

12    /** @notice postcondition allowed[_owner][_spender] == remaining */
   function allowance(address _owner, address _spender) public view returns (uint
14     remaining);

16    /** @notice postcondition ((balances[msg.sender] == __verifier_old_uint (balances[msg.
   sender]) - _value && msg.sender != _to) || (balances[msg.sender] ==
   __verifier_old_uint (balances[msg.sender]) && msg.sender == _to) && success) || !
18     success
   * @notice postcondition ((balances[_to] == __verifier_old_uint (balances[_to]) +
   _value && msg.sender != _to) || (balances[_to] == __verifier_old_uint (balances[
   _to]) && msg.sender == _to) && success) || !success
   * @notice postcondition forall (address addr) (addr == msg.sender || addr == _to ||
   __verifier_old_uint(balances[addr]) == balances[addr]) && success || (
   __verifier_old_uint(balances[addr]) == balances[addr]) && ! success
20    * @notice emits Transfer */
   function transfer(address _to, uint _value) public returns (bool success);

22    /** @notice postcondition ((balances[_from] == __verifier_old_uint (balances[_from]) -
   _value && _from != _to) || (balances[_from] == __verifier_old_uint (balances[_from
   ]) && _from == _to) && success) || !success
24    * @notice postcondition ((balances[_to] == __verifier_old_uint (balances[_to]) + _value
   && _from != _to) || (balances[_to] == __verifier_old_uint (balances[_to]) && _from
   == _to) && success) || !success
   * @notice postcondition allowed[_from][msg.sender] == __verifier_old_uint ( allowed[
   _from][msg.sender]) - _value || _from == msg.sender
26    * @notice postcondition allowed[_from][msg.sender] <= __verifier_old_uint(allowed[_from
   ][msg.sender]) || _from == msg.sender
   * @notice postcondition forall (address addr) (addr == _from || addr == _to ||
   __verifier_old_uint(balances[addr]) == balances[addr]) && success || (
   __verifier_old_uint(balances[addr]) == balances[addr]) && ! success
28    * @notice emits Transfer */
   function transferFrom(address _from, address _to, uint256 _value) public returns (bool
   success);

30    /** @notice postcondition (allowed[msg.sender][_spender] == _value && success) || (
   allowed[msg.sender][_spender] == __verifier_old_uint (allowed[msg.sender][
   _spender]) && !success)
32    * @notice emits Approval */
   function approve(address _spender, uint _value) public returns (bool success);

```

third postcondition for the *transferFrom* function presented in the specification in Figure 16 is not satisfied. Note that the allowance amount is not debited if the amount to be transferred is equal to the maximum integer supported by Solidity (i.e. *uint(-1)*). A possible solution would consist of removing the *if* branching, allowing the branch code to always execute.

The code snippet in Figure 18 - DigixDao repository, commit [5aee64](#) - does not conform to its formal specification. The correct allowance for the spender is only returned when it is not greater than the owner's balance. To fix this issue, we need to remove the code related

ERC20							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
0xMonorepo	548fda	2.85s	WOP	DsToken	3c436c	3.77s	No errors
0xMonorepo	6f2cb6	2.84s	No errors	DsToken	733e5c	3.81s	No errors
0xMonorepo	c84be8	2.57s	No errors	DsToken	8b8263	3.08s	No errors
Ambrosus	9fb24b	3.15s	No errors	Klenergy	3d4d62	5.14s	No errors
Ambrosus	b1806b	2.99s	No errors	Klenergy	60263d	1.70s	VRE
Ambrosus	db3ea0	3.74s	No errors	OpenZeppelin	3a5da7	3.59s	No errors
DigixDao	0550e8	5.97s	No errors	OpenZeppelin	<u>43ebb4</u>	3.57s	No errors
DigixDao	1c0c4f	8.82s	No errors	OpenZeppelin	5db741	3.87s	No errors
DigixDao	<u>5aee64</u>	7.60s	NTI	OpenZeppelin	5dfe72	3.96s	No errors
DigixDao	6bddc6	7.74s	No errors	OpenZeppelin	<u>9b3710</u>	3.45s	No errors
DigixDao	845F03	9.17s	No errors	Uniswap	4e4546	3.67s	No errors
DigixDao	aabf24	2.97s	No errors	Uniswap	<u>55ae25</u>	3.43s	WOP
DigixDao	e221ff	9.21s	No errors	Uniswap	e382d7	3.57s	IOU
DigixDao	e320a2	8.89s	No errors	SkinCoin	25db99	0.99s	NTI
DsToken	08412f	4.14s	WOP	SkinCoin	27c298	1.94s	NTI
DsToken	10c964	3.66s	No errors	SkinCoin	ac33d8	3.23s	No errors

Table 2 – ERC20 Results

Figure 17 – Buggy ERC20 transferFrom function

```

1 function transferFrom(address from, address to, uint value) external
  returns (bool success) {
3     if (allowance_[from][msg.sender] != uint(-1)) {
4         allowance_[from][msg.sender] =
5             allowance_[from][msg.sender].sub(value);
6     }
7     _transfer(from, to, value);
8     return true;
9 }

```

to `_balance` (lines 4 to 6 and 8), ensuring that the `_allowance` will be returned regardless of the `_balance` amount.

Figure 18 – Buggy ERC20 allowance function

```

1 function allowance(address _owner, address _spender) public returns (uint256 remaining) {
2     uint256 _allowance = allowed[_owner][_spender];
3     uint256 _balance = balances[_owner];
4     if (_allowance > _balance) {
5         remaining = _balance;
6     } else {
7         remaining = _allowance;
8     }
9     return remaining;
10 }

```

Figures 19 and 20 - extracted from the Openzeppelin repository, commits 9b3710 and 43ebb4, respectively - illustrate a case of safe contract evolution. The code of this contract

has undergone significant changes. The refactoring in question is one of the most common and is known as extract method (function, in Solidity). From commit [9b3710](#) to [43ebb4](#), the new internal function `_transfer` was created, and the extracted code from the original `transfer` function was moved into it. This internal function is then invoked by the function `transfer`. Our framework shows that both of these versions conform to the ERC20 specification and so they can be safely deployed.

Figure 19 – transferFrom function before refactoring

```

1 function transferFrom(address from, address to, uint256 value) public
  returns (bool success) {
3   require(value <= _allowed[from][msg.sender]);
  require(value <= _balances[from]);
5   require(to != address(0));
  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
7   _balances[from] = _balances[from].sub(value);
  _balances[to] = _balances[to].add(value);
9   emit Transfer(from, to, value);
  return true;
11 }

```

Figure 20 – Successful refactoring of the transferFrom function

```

1 function transferFrom(address from, address to, uint256 value) public returns (bool) {
3   require(value <= _allowed[from][msg.sender]);
  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
5   _transfer(from, to, value);
  return true;
7 }

9 function _transfer(address from, address to, uint256 value) internal {
  require(value <= _balances[from]);
11  require(to != address(0));
  _balances[from] = _balances[from].sub(value);
13  _balances[to] = _balances[to].add(value);
  emit Transfer(from, to, value);
15 }

```

## 5.4 ERC3156

Lending and credit are one of the most important financial activities and have been an integral part of human society and it is intricately related to the concept of trust and the promise of repayment (XU; VADGAMA, 2022). Usually the money is held by banks, a financial institution which facilitates the money flow between parties and charges fees for using their services. In recent times, the blockchain technology has significantly impacted centralized finance promoting financial inclusion and solving many problems in inefficient markets.

ERC3156							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
ArgoBytes	c3c92c	4.59s	No errors	Erc3156	512268	3.79s	NSC
ArgoBytes	e524c1	3.19s	No errors	Wrappes	70b977	6.65s	No errors
ArgoBytes	738d47	4.35s	No errors	Wrappes	5f65ac	3.23s	No errors
ArgoBytes	3409ee	2.57s	No errors	Wrappes	6bf6d6	4.79s	No errors
Dss Flash	<u>f2ca83</u>	5.15s	NSC	Weth10	<u>34c42c</u>	5.14s	NSC
Dss Flash	<u>b1e01d</u>	3.37s	NSC	Weth10	<u>b85345</u>	3.85s	NSC
Dss Flash	<u>2e70bb</u>	6.12s	NSC	Weth10	dbe412	3.97s	No errors
Dss Flash	18caa8	4.24s	No errors	Weth10	d2c480	4.25s	No errors
Erc3156	e18bdf	4.01s	NSC	Weth10	4fcc9e	5.39s	No errors

Table 3 – ERC3156 Results

Decentralized finance (DeFi) is a decentralized, censorship-free, low-fee, fully-automated, transparent and permissionless finance ecosystem which eliminates centralized finance since it allows people lend, trade, and borrow through peer-to-peer network without relying on intermediaries such as banks (WANG et al., 2021). DeFi democratizes financial services and relies on the integrity of smart contracts, so any flaw would enable attackers to launch malicious operations to take advantage of the protocol and make profits.

The ERC3156 standard is composed by *ERC3156FlashBorrower* and *ERC3156FlashLender* interfaces and together they provide a standardization for single-asset flash loans. Figure 21 presents a specification for the ERC3156 standard. The postcondition (Line 9) defined for the *maxFlashLoan* function checks if the amount of currency available to be lent is returned correctly; according to the specification defined for this function, the zero value must be returned if the token passed as parameter is not supported. The *flashFee* function must return the fee charged for a given loan (Line 13); when the token is not supported the operation must revert so when the verification process is successful it means that this condition has been met (Line 12). The *flashLoan* function initiates a flash loan and must not modify the token and amount parameters received (Lines 18 to 19). Flash lenders can provide loans of several token types on the same contract so it should be checked whether or not a token is supported (Line 16) as well as the amount available to lend (Line 17). If successful, the function must return true (Line 20). The *onFlashLoan* function receives a flash loan and requires an initiator which should be the same as the one passed as parameter by the lender function (Line 3). Table 3 shows the complete list of all ERC3156 results we obtained.

Figure 21 – ERC3156 specification

```

2  contract ERC3156FlashBorrower {
4      /** @notice postcondition initiator == msg.sender */
      function onFlashLoan(address initiator, address token, uint256 amount, uint256 fee, bytes
          calldata data) external returns (bytes32);
6  }
8  contract ERC3156FlashLender {
10     /** @notice postcondition resp == line && token == address(dai) || resp == 0 */
      function maxFlashLoan(address token) external view returns (uint256 resp);
12
13     /** @notice postcondition token == address(dai)
14     * @notice postcondition resp == (amount * toll)/WAD */
      function flashFee(address token, uint256 amount) external view returns (uint256 resp);
16
17     /** @notice postcondition token == address(dai)
18     * @notice postcondition amount <= line
19     * @notice postcondition __verifier_old_address(token) == token
20     * @notice postcondition __verifier_old_uint(amount) == amount
21     * @notice postcondition resp */
      function flashLoan(IERC3156FlashBorrower receiver, address token, uint256 amount,
          bytes calldata data) external returns (bool resp);
22 }

```

### ERC3156 Analysis

The function in Figure 22 - Dss Flash repository, commits [f2ca83](#), [b1e01d](#) and [2e70bb](#) - does not conform to the requirements defined by the ERC3156 standard. According to the function signature, a boolean value must be returned. After formally verifying the code on the three mentioned commits, it was detected that no explicit return value has been defined by the developer, so the *false* value is returned by default. According to the function's specification, the value *true* must be returned whenever its execution is successful. This represents a serious flaw because from the point of view of a client that invokes the *flashLoan* function, its execution would never be successful, so this iteration could lead to an unexpected behavior.

The code snippet in Figure 23 - Weth10 repository, commits [34c42c](#) and [b85345](#) - illustrates two functions that have an error when checked against its formal specification. The ERC3156 Flash Loans standard can handle multiple tokens, so it is mandatory for these functions to check the address passed by parameter and return the right value or revert when it is necessary.

Figure 22 – Buggy flashLoan function

```

function flashLoan(IERC3156FlashBorrower receiver, address token, uint256 amount, bytes
  calldata data) external override lock returns (bool) {
2   require(token == address(dai), "DssFlash/token-unsupported");
   require(amount <= line, "DssFlash/ceiling-exceeded");

4   uint256 rad = mul(amount, RAY);
6   uint256 fee = mul(amount, toll) / WAD;
   uint256 total = add(amount, fee);

8   vat.suck(address(this), address(this), rad);
10  daiJoin.exit(address(receiver), amount);

12  emit FlashLoan(address(receiver), token, amount, fee);

14  require(receiver.onFlashLoan(msg.sender, token, amount, fee, data) == keccak256("
      ERC3156FlashBorrower.onFlashLoan"), "IERC3156: Callback failed");

16  dai.transferFrom(address(receiver), address(this), total);
18  daiJoin.join(address(this), total);
   vat.heal(rad);
   vat.move(address(this), vow, mul(fee, RAY));
20 }

```

Figure 23 – Buggy flashFee and maxFlashLoan functions

```

function flashFee(address token, uint256 value) external view returns (uint256) {
2   return value.mul(9).div(10000);
   }

4   function maxFlashLoan(address token) external view returns (uint256) {
6       LendingPoolLike.ReserveData memory reserveData = lendingPool.getReserveData(token);
       return IERC20(reserveData.aTokenAddress).balanceOf(address(lendingPool));
8   }

```

## 5.5 ERC1155

The ERC1155 standard was created in order to promote a better integration between the ERC20 and the ERC721, a standard for representing ownership of non-fungible tokens, that is, where each token is unique, so an ERC1155 token can perform the same functions as those of the aforementioned tokens, improving the functionality of both and avoiding implementation errors. It provides an interface for managing any combination of fungible and non-fungible tokens in a single contract efficiently. One of the most important features of the ERC1155 standard is secure transfer, that is, the smart contract that follows this pattern includes a function that checks if the destination address is able to receive the transaction and, if not, reverts it to return control of the tokens to the issuer. In addition, the ERC1155 standard allows batch transfers which can significantly reduce the gas fees paid to the Ethereum network.

The functions *balanceOf* and *balanceOfBatch* returns the balance of specific tokens of the address and a list of addresses specified in the parameter function respectively. The function

Figure 24 – ERC1155 specification

```

contract ERC1155 {
2
    mapping (uint256 => mapping(address => uint256)) private _balances;
4
    mapping (address => mapping(address => bool)) private _operatorApprovals;

6
    /** @notice postcondition _balances[id][account] == balance */
    function balanceOf(address account, uint256 id) public view returns (uint256 balance){}
8

    /** @notice postcondition batchBalances.length == accounts.length
    * @notice postcondition batchBalances.length == ids.length
    * @notice postcondition forall (uint x) !( 0 <= x && x < batchBalances.length ) ||
    *   batchBalances[x] == _balances[ids[x]][accounts[x]] */
10
    function balanceOfBatch( address[] memory accounts, uint256[] memory ids ) public view
    returns (uint256[] memory batchBalances) {}
12

14
    /** @notice postcondition _operatorApprovals[msg.sender][operator] == approved
    * @notice emits ApprovalForAll */
    function setApprovalForAll(address operator, bool approved) public {}
16

18
    /** @notice postcondition _operatorApprovals[account][operator] == approved */
    function isApprovedForAll(address account, address operator) public view returns (
    bool approved) { }
20

22
    /** @notice postcondition to != address(0)
    * @notice postcondition _operatorApprovals[from][msg.sender] || from == msg.sender
    * @notice postcondition __verifier_old_uint ( _balances[id][from] ) >= amount
    * @notice postcondition _balances[id][from] == __verifier_old_uint ( _balances[id][from
24
    ] ) - amount
    * @notice postcondition _balances[id][to] == __verifier_old_uint ( _balances[id][to] )
    + amount
26
    * @notice emits TransferSingle */
    function safeTransferFrom(address from, address to, uint256 id, uint256 amount, bytes
    memory data ) public { }
28

30
    /** @notice postcondition _operatorApprovals[from][msg.sender] || from == msg.sender
    * @notice postcondition to != address(0)
    * @notice emits TransferBatch */
32
    function safeBatchTransferFrom( address from, address to, uint256[] memory ids,
    uint256[] memory amounts, bytes memory data) public {}
}

```

*isApprovedForAll* returns a boolean value informing if an address is allowed to handle the tokens from another address. The *safeTransferFrom* function transfers tokens in a safe way to a valid ERC1155 address. The *transferFrom* function can do batch operations, transferring tokens to several wallets at the same time, reducing transaction costs. The *setApprovalForAll* function gives an address permission to handle another address' tokens.

Figure 24 presents a specification for the ERC1155 standard. The *isApprovedForAll* and *balanceOf* functions only have one postcondition each to ensure that the balance of a given address (line 6) and the approval to manage all tokens from another address (line 18) will be returned correctly. The postcondition defined to *setApprovalForAll* function enforces the approval status should be the same as the one passed as parameter (line 14); an *ApprovalForAll* event should be emitted whenever the approval status is changed (line 15). In the *balanceOfBatch* function, the size of the list of accounts and token ids should be equal to the



ERC3155							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
0xSequence	<a href="#">319740</a>	4.82s	No errors	OpenZeppelin	0db76e	5.59s	No errors
0xSequence	<a href="#">578d46</a>	5.31s	No errors	OpenZeppelin	440b65	6.61s	No errors
0xSequence	<a href="#">99012f</a>	5.59s	No errors	OpenZeppelin	5db741	6.70s	No errors
0xSequence	<a href="#">acfa7c</a>	5.81s	No errors	OpenZeppelin	956d66	8.58s	No errors
Desc-Stock	<a href="#">44464c</a>	4.34s	IOU	Ejin-Erc	30dba0	4.13s	No errors
Desc-Stock	<a href="#">4c5d80</a>	5.18s	IOU	Ejin-Erc	614714	4.49s	No errors
Desc-Stock	<a href="#">96d5b2</a>	4.33s	WOP	Ejin-Erc	bf4d04	4.01s	No errors
Desc-Stock	<a href="#">ae8a13</a>	4.24s	IOU	Ejin-Erc	cc96af	4.57s	No errors
Desc-Stock	<a href="#">bf2c1a</a>	3.60s	IOU	Ejin-Erc	e20fc9	3.77s	No errors

Table 4 – ERC1155 Results

list of balances returned, besides we must also check if the balances are in the list (lines 9 to 11). The postconditions defined for the functions *safeTransferFrom* (lines 21 to 25) and *safeBatchTransferFrom* (lines 29 to 30) are quite similar. In both cases, for the operation to be successful, the token or a batch of them should be debited from the source account and credited in the destination account. Table 4 shows the the complete list of all ERC1155 results we obtained.

### ERC1155 Analysis

Figures 25 and 26 - extracted from the Decentralized-Stock repository, commits [96d5b2](#) and [4c5d80](#), respectively - illustrate a case of safe contract evolution. The code of this contract has undergone significant changes. The refactoring in question was intended to fix a bug introduced in commit [96d5b2](#) where the developer took out the code responsible for checking the size of lists passed as parameter. According to the specification for the ERC1155 standard, this checking is mandatory and without it could lead to unexpected results, since an *out-of-bounds* exception occurs when the *\_ids* list is greater than the *\_owners* list.

Figure 25 – balanceOfBatch function before refactoring

```

1  function balanceOfBatch(address[] calldata _owners, uint256[] calldata _ids) external
   override view returns (uint256[] memory){
2
3      uint256[] memory _balances;
4      for (uint256 i = 0; i < _owners.length; ++i) {
5          require(_owners[i] != address(0));
6          _balances[i] = _balance[_owners[i]][_ids[i]];
7      }
8      return _balances;
9  }

```

Figure 26 – Successful refactoring of the balanceOfBatch function

```

1  function balanceOfBatch(address[] calldata _owners, uint256[] calldata _ids) external
    override view returns (uint256[] memory){
3
    require(_owners.length == _ids.length);
    uint256[] memory _balances = new uint256[](_owners.length);
5    for (uint256 i = 0; i < _owners.length; ++i) {
        require(_owners[i] != address(0));
7        _balances[i] = (_balance[_owners[i]][_ids[i]]);
    }
9    return _balances;
}

```

The snippet in Figure 27 - extracted from Decentralized-Stock repository, commit [96d5b2](#) - illustrates another example of the wrong operator error. A postcondition was not satisfied, because, according to the specification, the size of the `_ids` and `_values` arrays must be equal. So, any call to this function would result in an error or an unexpected behavior. A possible solution would consist of changing the operator `!=` for `==` in the second `require` in Figure 27.

Figure 27 – Buggy ERC1155 safeBatchTransferFrom function

```

function safeBatchTransferFrom(address _from, address _to, uint256[] calldata _ids,
    uint256[] calldata _values, bytes calldata _data) external {
2    require(_to != address(0) && _from != address(0));
    require(_ids.length != _values.length);
4    require(_approv[_from][msg.sender] || _from == msg.sender);

6    for (uint256 i = 0; i < _ids.length; ++i) {
        require(_balance[_from][_ids[i]] >= _values[i]);
8        _balance[_from][_ids[i]] -= _values[i];
        _balance[_to][_ids[i]] += _values[i];
10    }
    emit TransferBatch(msg.sender, _from, _to, _ids, _values);
12    require(_checkOnERC1155BatchReceived(msg.sender, _from, _to, _ids, _values, _data))
        ;
}

```

The snippet in Figure 28 - also extracted from Decentralized-Stock repository, commit [96d5b2](#) - has overflow/underflow error, that is, the arithmetic operation for crediting and debiting reaches the maximum or minimum size of a type. This is a common vulnerability in Ethereum smart contracts because the EVM provides no indication that an overflow/underflow has occurred, that is, the program does throw an exception and executes subsequent code. One approach to solve this problem is to perform the arithmetic operation, check the result, and revert the transaction if an error occurs.

Figure 28 – Buggy ERC1155 safeTransferFrom function

```

1  function safeTransferFrom(address _from, address _to, uint256 _id, uint256 _value, bytes
    calldata _data) external{
2      require(_to != address(0));
3      require(_balance[_from][_id] >= _value);
4      require(_from == msg.sender || _isApproved[_from][msg.sender]);
5      require(_checkOnERC1155Received(msg.sender, _from, _to, _id, _value, _data)==true);
6
7      _balance[_from][_id] = _balance[_from][_id] - _value;
8      _balance[_to][_id] += _value;
9
10     emit TransferSingle(msg.sender, _from, _to, _id, _value);
11 }

```

## 5.6 RESULTS AND DISCUSSION

The results of our evaluation suggest that the kind of verification that we employ in our framework is tractable, as *solc-verify* can efficiently analyse these samples. The fact that errors that could lead to millionaire losses were detected in real-world contracts attests to the necessity of our framework and its practical impact. Smart contracts are increasingly popular, and we believe they will become a key and common element of trusted distributed systems in the future. Therefore, having a safe development process supported by our framework will help to increase the credibility of such a technology and promote its adoption.

Our initial motivation to gather the samples from public github repositories may be a threat to our search strategy. Since we could not analyze private repositories, relevant cases to show strengths and weaknesses of the framework may not have been included. The relatively low number of samples could also be considered as a threat, since it could lead to an unintentionally biased study or less comprehensive than it could have been. Our framework also presents some limitations, since it is not in our scope to verify errors related to inter-contract interactions.

In this work, we do not focus on security properties and trust guarantees of our trusted deployer. Instead, we focus on the functional aspect of our framework. For instance, we do not discuss in detail how the trusted deployer and its infrastructure can be trusted, nor how to establish a secure communication channel with it. We leave a detailed discussion on all these aspects together with the mechanisms and protocols by which they can be implemented for follow-up work.

We focus on contract upgrades that preserve the signature of public functions. Also, we assume contract specifications fix the data structures used in the contract implementation. However, we plan to relax these restrictions in future versions of the framework, allowing the data structures in the contract implementation to be a data refinement of those used in the

specification; we also plan to allow the signature of the implementation to extend that of the specification, provided some notion of behaviour preservation is obeyed when the extended interface is projected into the original one.

Ensuring software correctness has long been the goal in computer science. A smart contract security audit is a methodical way to detect bugs in smart contracts and it is very important to protect resources invested. Usually, auditors will examine the code of smart contracts, produce a report, and provide it to the developers for them to work with. Smart contract audits don't focus only on blockchain security. They also look at efficiency and optimization. Since software involves a construction process, it needs to be guaranteed that from requirements to implementation the software meets the required specifications. Formal methods have become an important tool to solve this problem (GARCIA, 2009).

Unlike manual auditing, formal verification can provide complete coverage with respect to a given requirement. It ensures that each requirement has been mathematically verified. Although *Trusted Deployer* requires users to manually classify all reported vulnerabilities into true positives or false positives, compared to manual audit for smart contracts, *Trusted Deployer* reduces the required effort to inspect the code.

Formal verification is complete with respect to any given requirement. However, additional activities are necessary to ensure that all requirements have been expressed, that is, all admissible behaviors of the software have been specified. This activity states that the completeness of the set of requirements should be demonstrated with respect to the intended function. Since formal methods can't handle the problem of detecting all missing requirements. A manual auditing would be necessary as a complementary activity to the verification process.

## 5.7 LIMITATIONS AND THREATS

According to (RUNESON et al., 2012) the validity of a study denotes the trustworthiness of the results, and to what extent the results are not biased by the researchers' subjective point of view. So, to ensure that the results are consistent with the investigated reality, the researcher must plan the study taking into account four types of threats: construct validity, internal validity, external validity, and reliability. Although we adopted a protocol to gather and analyse the data presented in the last section, this study still has some limitations as discussed below.

Our initial motivation to gather the samples from public github repositories may be a threat

to our search strategy. Since we could not analyze private repositories, relevant cases to show strengths and weaknesses of the framework may not have been included. Our approach was successful to identify and analyse nonconformities that went unnoticed during the development process but the relatively low number of samples could also be considered as a threat, since it could lead to an unintentionally biased study or less comprehensive than they could have been.

Our framework also presents some limitations, since it is not in our scope to verify errors related to the consensus mechanism, EVM bad design and concurrency. The postconditions are bound to the methods that make up the contract interface so it is only possible to specify loop invariants in the body of the implementation contract functions. Although we know that requirements can evolve in such a way that makes the original smart contracts unsuitable for stakeholders' needs. It is important to point out that the study is based on a notion of immutable specifications, therefore, it does not take into account possible changes in the requirements that could lead to changes in the smart contracts interfaces.

## 6 CONCLUSION

In this chapter, we discuss the main contributions, as well as suggestions for future work.

### 6.1 CONTRIBUTIONS

We propose a framework for the safe deployment of smart contracts. Not only does it check that contracts conform to their specification at creation time, but it also guarantees that subsequent code updates are conforming too. Upgrades can be performed even if the implementation has been proven to satisfy the specification initially. A developer might, for instance, want to optimise the resources used by the contract. Furthermore, our *trusted deployer* records information about the contracts that have been verified, and which specification they conform to, so that participants can be certain they are interacting with a contract with the expected behaviour; contracts can be safely executed. None of these capabilities are offered by the Ethereum platform by default nor are available in the literature to the extent provided by the framework proposed in this paper.

We have prototyped our trusted deployer and investigated its applicability - specially its formal verification component - to contracts implementing three widely used Ethereum standards: the ERC20 Token Standard, ERC3156 Flash Loans and ERC1155 Multi Token Standard, with promising results.

This idea of using trusted computing to verify a smart contract before its deployment can be extended to software in general. Particularly, a trusted deployer could be part of a deployment process for reactive systems in general, such as component-based, (micro)service-based systems, or even system-of-systems.

Our framework shifts immutability from the implementation of a contract to its specification, promoting the "code is law" to the "specification is law" paradigm. We believe that this paradigm shift brings a series of improvements. Firstly, developers are required to outline their intent in the form of a (formal) specification, so they can, early in the development process, identify issues with their design. They can and should validate their specification; we consider this problem orthogonal to the framework that we are providing. Secondly, specifications are more abstract and, as a consequence, tend to be more stable than (the corresponding conforming) implementations. A contract can be optimised, for instance, and both the original

and optimised versions must satisfy the same reference specification. Thirdly, even new implementations that involve change of data representation can still be formally verified against the same specification, by using data refinement techniques.

## 6.2 RELATED WORK

There is a glaring need for a safe mechanism to upgrade smart contracts in platforms, such as Ethereum, where contract implementations are immutable once deployed; the many surveys uncovering this fact (HU et al., 2021; TOLMACH et al., 2021; GROCE et al., 2020) and community-proposed design patterns proposing mechanisms to upgrade smart contracts (OPENZEPELIN, 2021; BARROS; GALLAGHER, 2019; LU, 2018; PALLADINO, 2019) attest this necessity. Yet, surprisingly, we could only find three close related approaches (DICKERSON et al., 2018; RODLER et al., 2021b) that try to tackle this problem. The “preliminary work” in (DICKERSON et al., 2018) proposes a methodology based around special contracts that carry a proof that they meet the expected specification. Their on-chain solution requires fundamental changes in the smart contract platforms themselves. They propose the addition of a special instruction to deploy these special proof-carrying contracts, and the adaptation of platform miners, which are responsible for checking and reaching a consensus on the validity of contract executions, to check these proofs. Our framework and the one presented in that work share the same goal: to propose a mechanism by which contracts can be upgraded but only if they meet the expected specification. However, our approach and theirs differ significantly in many aspects. Firstly, while theirs requires a complete change on the rules of the platform - which requires a large distributed network, i.e. the smart contract platform, to synchronise and agree on this change - ours can be implemented, as detailed in this paper, on top of Ethereum’s current capabilities. Moreover, as our framework relies on an off-chain service to ensure that an implementation meets a specification, we can rely on methods that are easier to use, i.e. require less user input, like model checking. The fact that their framework is on-chain makes the use of such verification methods more difficult - the complex computations typically associated with these methods would slow down consensus, likely to a prohibitive level. Hence, they rely on the user to construct a proof that an implementation meets a specification, since checking that a proof is valid tends to be much less complex than constructing it. Finally, there is an inherent difference of maturity between the two works. While theirs introduces abstract ideas with some concrete elements, we provide details on how to implement ours using current technology and

an evaluation based on real-world Solidity samples.

Azzopardi *et al.* (AZZOPARDI; ELLUL; PACE, 2018b) propose the use of runtime verification to ensure that a contract conforms to its specification. Given a Solidity smart contract  $C$  and an automaton-based specification  $S$ , their approach produces an instrumented contract  $I$  that dynamically tracks the behaviour of  $C$  with respect to  $S$ .  $I$ 's behaviour is functionally equivalent to  $C$  when  $S$  is respected. If a violation to  $S$  is detected, however, a reparation strategy (i.e. some user-provided code) is executed instead. This technique can be combined with a proxy to ensure that a monitor contract keeps track of implementation contracts as they are upgraded, ensuring their safe evolution. Unlike our approach, there is an inherent (on-chain) runtime overhead to dynamically keep track of specification conformance.

In (RODLER *et al.*, 2021b), the authors propose a mechanism to upgrade contracts in Ethereum that works at the EVM-bytecode level. Their framework takes vulnerability reports issued by the community as an input, and tries to patch affected deployed contracts automatically using patch templates. It uses previous contract transactions and, optionally user-provided unit tests, to try to establish whether a patch preserves the behaviour of the contract. Ultimately, the patching process may require some manual input. If the deployed contract and the patch disagree on some test, the user must examine this discrepancy and rule on what should be done. Note that this manual intervention is always needed for attacked contracts, as the transaction carrying out the attack - part of the attacked contract's history - should be prevented from happening in the new patched contract.

The main difference between our work and theirs relates to the approach to validate patches. While they simply test patches, we formally verify them against a formal specification. Their approach requires less human intervention, as a specification does not need to be provided - only optionally some unit tests - but it offers no formal guarantees about patches. It could be that a patch passes their validation (i.e. testing with the contract history), without addressing the underlying vulnerability. Our framework, however, is based around a formal notion of "patch validity" that ensures that patches always respect a user-provided specification. Finally, while our approach is proactive in requiring the user to provide a definition for the expected behaviour of a contract, and possibly spotting implementation vulnerabilities while verifying conformance to that specification, theirs is based on vulnerability reports being issued first for then contracts to be rectified.

As soon as smart contract platforms came to be, it became apparent that supporting developers with verification tools was a necessity. A number of analysis tools for EVM byte-



code were created (LUU et al., 2016b; MOSSBERG et al., 2019; LIU et al., 2018; GRISHCHENKO; MAFFEI; SCHNEIDEWIND, 2018; TIKHOMIROV et al., 2018b; TSANKOV et al., 2018b; PERMENEV et al., 2020). They were designed to find specific behaviour patterns witnessing typical bad behaviours. Tools operating on the level of Solidity were also proposed (WANG et al., 2020; HAJDU; JOVANOVIĆ, 2020b; HAJDU; JOVANOVIĆ, 2020a; ANTONINO; ROSCOE, 2021a). These tools tend to focus instead on formally verifying user-provided semantic properties. All of these tools are concerned with the verification aspect but they do not provide any framework to be integrated into a smart contract deployment process.

Design by contract (MEYER, 1992) is a methodology that was originally created for specifying the behaviour of object-oriented programs but was also adopted in other contexts (MEYER, 1988; LEINO, 2010; BARNETT et al., 2005; LEINO, 2008; LEAVENS; BAKER; RUBY, 1999; HAJDU; JOVANOVIĆ, 2020b). This sort of specification is particularly fitting in the case of Solidity smart contracts, especially the format of specification that we propose, as the community already employ a similar format, albeit informal, to describe standard contract interfaces in the form of Ethereum Request for Comments (ERCs); see for example, ERC20 (VOGELSTELLER; BUTERIN, 2015).

Some papers have proposed methodologies to carry out pre-deployment patching/repairing (TORRES; JONKER; STATE, 2021; NGUYEN; PHAM; SUN, 2021; YU et al., 2020). They try to scan a binary for common vulnerabilities and patch the vulnerabilities they find prior to deploying the contract. These papers do not propose a way to update deployed contracts.

### 6.3 FUTURE WORK

A limitation of our current approach is the restrictive notion of evolution for smart contracts: only the implementation of public functions can be upgraded - the persistent state data structures are fixed. However, we are looking into new types of evolution where the data structure of the contract's persistent state can be changed - as well as the interface of the specification, provided the projected behaviour with respect to the original interface is preserved, based on notions of class (LISKOV; WING, 1994) and process (DIHEGO; ANTONINO; SAMPAIO, 2013) inheritance, and interface evolution such as in (DIHEGO; SAMPAIO; OLIVEIRA, 2020).

This work focuses on creating, updating and deploying safe smart contracts and their consequences. It is a rather new and complex subject therefore there are still many themes

---

that can be explored from the study presented here. In this section, we present new ideas that can make our approach more efficient, extensive and systematic. The recommendations for future research with new directions are:

- Throughout the study the differences between informal and formal requirements specification languages were noted. For any project to be successful a high level of adherence between the requirements and the functionality of the developed artifacts is required. In the future, we plan to fulfill the gap between the formal and informal approaches through a systematic mapping between them. Such an approach will help to systematize the process of defining invariants and postconditions and keep a high level of traceability.
- Smart contract development is a dynamic process and requirements changes are a part of it. So understanding the possible implications of making the change is a key aspect of responsible requirements management. As future work, we plan to adopt an impact analysis in order to reduce the risk of missing changes to dependent items and guarantee that postconditions and invariants are adhering to the new requirements. We also plan to consider a more flexible notion of conformance, capturing changes of smart contract specification, data representation (data refinement) and interface evolution as, for example, in (DIHEGO; SAMPAIO; OLIVEIRA, 2020). This will require extending the current state-of-the-art tool support to smart contract verification.

## REFERENCES

- ADHIKARI, C. Secure framework for healthcare data management using ethereum-based blockchain technology. In: *2017 Undergraduate Research and Scholarship Conference*. [S.l.: s.n.], 2017.
- AITZHAN, N. Z.; SVETINOVIC, D. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. In: *IEEE Transactions on Dependable and Secure Computing*. [S.l.: s.n.], 2016. p. 840 – 852.
- ALCHEMY, N. *A short history of smart contract hacks on Ethereum*. 2018. Disponível em: <<https://medium.com/new-alchemy/a-short-history-of-smart-contract-hacks-on-ethereum-1a30020b5fd>>. Acesso em: Oct 29th, 2021.
- ANTONINO, P.; ROSCOE, A. W. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. In: . [S.l.: s.n.], 2020.
- ANTONINO, P.; ROSCOE, A. W. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2021. (SAC '21), p. 1788–1797.
- ANTONINO, P. R.; ROSCOE, A. W. Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. In: *The 36th ACM/SIGAPP Symposium on Applied Computing*. [S.l.: s.n.], 2021. p. 1788–1797.
- ANTONOPOULOS, A. M. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies Capa comum*. [S.l.]: O'Reilly Media, 2014.
- ANTONOPOULOS, A. M.; WOOD, G. *Mastering Ethereum*. 2021. Disponível em: <<https://www.oreilly.com/library/view/mastering-ethereum/9781491971932/ch04.html>>. Acesso em: Feb 15th, 2022.
- ATZEI, N.; BARTOLETTI, M.; CIMOLI, T. A survey of attacks on ethereum smart contracts (sok). In: SPRINGER. *POST 2017*. [S.l.], 2017. p. 164–186.
- AZZOPARDI, S.; ELLUL, J.; PACE, G. J. Monitoring smart contracts: Contractlarva and open challenges beyond. In: *International Conference on Runtime Verification*. [S.l.: s.n.], 2018. p. 113–137.
- AZZOPARDI, S.; ELLUL, J.; PACE, G. J. Monitoring smart contracts: Contractlarva and open challenges beyond. In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11237), p. 113–137.
- BAHGA, A.; MADISETTI, V. K. Blockchain platform for industrial internet of things. In: *Journal of Software Engineering and Application*. [S.l.: s.n.], 2016. v. 9, p. 533–546.
- BARNETT, M.; CHANG, B.-Y. E.; DELINE, R.; JACOBS, B.; LEINO, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In: SPRINGER. *FMCO 2005*. [S.l.], 2005. p. 364–387.

- BARROS, G.; GALLAGHER, P. *EIP-1822: Universal Upgradeable Proxy Standard (UUPS)*. 2019. <<https://eips.ethereum.org/EIPS/eip-1822>>.
- BENIICHE, A. A study of blockchain oracles. In: . [S.l.: s.n.], 2020.
- BIRYUKOV, A.; KHOVRATOVICH, D.; TIKHOMIROV, S. Findel: secure derivative contracts for ethereum. In: FC. *Financial Cryptography and Data Security - FC 2017 International Workshops*. [S.l.], 2017. p. 453–467.
- BUTERIN, V. *Ethereum White Paper*. 2014. <<https://github.com/ethereum/wiki/wiki/White-Paper>>.
- CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: . [S.l.: s.n.], 1999.
- CHEN, T.; LI, X.; LUO, X.; ZHANG, X. Under-optimized smart contracts devour your money. In: IEEE. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2017. p. 442–446.
- CHEN, Y.-C.; CHOU, Y.-P.; CHOU, Y.-C. An image authentication scheme using merkle tree mechanisms. In: *Future Internet*. [S.l.: s.n.], 2019. p. 1–18.
- CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. In: *ACM Computing Surveys*. [S.l.: s.n.], 1996. p. 626–643.
- COWLING, J.; MYERS, D.; LISKOV, B.; RODRIGUES, R.; SHRIRA, L. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In: . [S.l.]: IEEE, 2006.
- DICKERSON, T. D.; GAZZILLO, P.; HERLIHY, M.; SARAPH, V.; KOSKINEN, E. Proof-carrying smart contracts. In: *Financial Cryptography Workshops*. [S.l.: s.n.], 2018.
- DIHEGO, J.; ANTONINO, P. R. G.; SAMPAIO, A. Algebraic laws for process subtyping. In: GROVES, L.; SUN, J. (Ed.). *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings*. Springer, 2013. (Lecture Notes in Computer Science, v. 8144), p. 4–19. Disponível em: <[https://doi.org/10.1007/978-3-642-41202-8\\_2](https://doi.org/10.1007/978-3-642-41202-8_2)>.
- DIHEGO, J.; SAMPAIO, A.; OLIVEIRA, M. A refinement checking based strategy for component-based systems evolution. *J. Syst. Softw.*, v. 167, p. 110598, 2020. Disponível em: <<https://doi.org/10.1016/j.jss.2020.110598>>.
- FEIST, J.; GRIECO, G.; GROCE, A. Slither: A static analysis framework for smart contracts. In: IEEE. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. [S.l.], 2019.
- FRANKENFIELD, J. *Nonce*. 2021. Disponível em: <<https://www.investopedia.com/terms/n/nonce.asp>>. Acesso em: Feb 28th, 2022.
- GARCIA, G. A. Formal verification and testing of software architectural models. In: . [S.l.: s.n.], 2009.
- GRISHCHENKO, I.; MAFFEI, M.; SCHNEIDEWIND, C. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, 2018.

GROCE, A.; FEIST, J.; GRIECO, G.; COLBURN, M. What are the actual flaws in important smart contracts (and how can we find them)? In: BONNEAU, J.; HENINGER, N. (Ed.). *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020. p. 634–653.

HAHN, A.; SINGH, R.; LIU, C.-C.; CHEN, S. Smart contract-based campus demonstration of decentralized transactive energy auctions. In: IEEE. *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference*. [S.l.], 2017. p. 1–5.

HAJDU, Á.; JOVANOVIĆ, D. Smt-friendly formalization of the solidity memory model. In: *ESOP 2020*. [S.l.]: Springer, 2020. p. 224–250.

HAJDU, Á.; JOVANOVIĆ, D. solc-verify: A modular verifier for solidity smart contracts. In: *VSTTE*. [S.l.]: Springer, 2020. p. 161–179.

HAJDU Ákos; JOVANOVIĆ, D.; CIOCARLIE, G. Formal specification and verification of solidity contracts with events. In: . [S.l.: s.n.], 2020.

HILDENBRANDT, E.; SAXENA, M.; RODRIGUES, N.; ZHU, X.; DAIAN, P.; GUTH, D.; MOORE, B.; PARK, D.; ZHANG, Y.; STEFANESCU, A. et al. Kevm: A complete formal semantics of the ethereum virtual machine. In: IEEE. *CSF 2018*. [S.l.], 2018. p. 204–217.

HU, B.; ZHANG, Z.; LIU, J.; LIU, Y.; YIN, J.; LU, R.; LIN, X. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, v. 2, n. 2, p. 100179, 2021. ISSN 2666-3899. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S2666389920302439>>.

HUISMAN, M.; GUROV, D.; MALKIS, A. Formal methods: From academia to industrial practice. a travel guide. In: . [S.l.: s.n.], 2020.

JAMESON, H. *Introduction to Ethereum Improvement Proposals (EIPs)*. 2022. <<https://ethereum.org/en/eips/>>.

KRASNER, G. E.; POPE, S. A description of the model-view-controller user interface paradigm in the smalltalk80 system. In: . [S.l.: s.n.], 1988.

KUMAR, D. G.; SUBBARAO, C. D. V. Peer – to – peer computing: Architectures, applications and challenges. In: *International Journal of Recent Technology and Engineering*. [S.l.: s.n.], 2019.

LEAVENS, G. T.; BAKER, A. L.; RUBY, C. Jml: A notation for detailed design. In: \_\_\_\_\_. *Behavioral Specifications of Businesses and Systems*. Boston, MA: Springer US, 1999. p. 175–188. ISBN 978-1-4615-5229-1. Disponível em: <[https://doi.org/10.1007/978-1-4615-5229-1\\_12](https://doi.org/10.1007/978-1-4615-5229-1_12)>.

LEE, J.; NIKITIN, K.; SETTY, S. Replicated state machines without replicated execution. In: . [S.l.: s.n.], 2020. (IEEE).

LEINO, K. R. M. This is boogie 2. *manuscript KRML*, Citeseer, v. 178, n. 131, p. 9, 2008.

LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In: CLARKE, E. M.; VORONKOV, A. (Ed.). *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 348–370. ISBN 978-3-642-17511-4.

LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 16, n. 6, p. 1811–1841, nov. 1994. ISSN 0164-0925. Disponível em: <<https://doi.org/10.1145/197320.197383>>.

LIU, C.; LIU, H.; CAO, Z.; CHEN, Z.; CHEN, B.; ROSCOE, B. Reguard: finding reentrancy bugs in smart contracts. In: ACM. *ICSE 2018*. [S.l.], 2018. p. 65–68.

LU, A. *Solidity DelegateProxy Contracts*. 2018. <<https://blog.gnosis.pm/solidity-delegateproxy-contracts-e09957d0f201>>.

LUU, L.; CHU, D.-H.; OLICKEL, H.; SAXENA, P.; HOBOR, A. Making smart contracts smarter. In: CCS '16. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. [S.l.], 2016. p. 254–269.

LUU, L.; CHU, D.-H.; OLICKEL, H.; SAXENA, P.; HOBOR, A. Making smart contracts smarter. In: ACM. *CCS 2016*. [S.l.], 2016. p. 254–269.

MADIR, J. *Introducing Blockchain-Based Smart Contracts – A Roadmap For Lawmakers*. 2018. <<https://sites.law.duke.edu/thefinregblog/2018/11/13/introducing-blockchain-based-smart-contracts-a-roadmap-for-lawmakers/>>.

MCCORRY, P.; SHAHANDASHTI, S. F.; HAO, F. A smart contract for boardroom voting with maximum voter privacy. In: KIAYIAS A. (EDS). *Financial Cryptography and Data Security. FC 2017. Lecture Notes in Computer Science*. [S.l.], 2017. v. 10322, p. 357–375.

MCSHANE, G. *What Is a 51% Attack?* 2021. Disponível em: <<https://www.coindesk.com/learn/what-is-a-51-attack/>>. Acesso em: Feb 28th, 2022.

MEYER, B. *Object-Oriented Software Construction*. 1st. ed. USA: Prentice-Hall, Inc., 1988. ISBN 0136290493.

MEYER, B. Applying 'design by contract'. *Computer*, v. 25, n. 10, p. 40–51, 1992.

MICHELIN, R. A. A lightweight blockchain data model for the internet of things. In: . [S.l.: s.n.], 2019.

MISSION, H. A. Applying formal verification techniques to embedded software in uav design. In: . [S.l.: s.n.], 2019.

MOSSBERG, M.; MANZANO, F.; HENNENFENT, E.; GROCE, A.; GRIECO, G.; FEIST, J.; BRUNSON, T.; DINABURG, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: IEEE. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2019. p. 1186–1189.

MUSUMECI, M. *PRIVATE AND PUBLIC KEYS ON ETHEREUM*. 2018. Disponível em: <<https://www.massmux.com/private-and-public-keys-on-ethereum/>>. Acesso em: Feb 8th, 2022.

NADOLINSKI, E.; SPAGNUOLO, F. *Proxy Patterns*. 2018. <<https://blog.openzeppelin.com/proxy-patterns/>>.

NAKAMOTO, S. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009. Disponível em: <<http://bitcoin.org/bitcoin.pdf>>.

- NGUYEN, T. D.; PHAM, L. H.; SUN, J. Sguard: Towards fixing vulnerable smart contracts automatically. In: *2021 IEEE Symposium on Security and Privacy (SP)*. [S.l.: s.n.], 2021. p. 1215–1229.
- NIKOLIC, I.; KOLLURI, A.; SERGEY, I.; SAXENA, P.; HOBOR, A. Finding the greedy, prodigal, and suicidal contracts at scale. In: . [S.l.: s.n.], 2018.
- NOTHEISEN, B.; GÖDDE, M.; WEINHARDT, C. Trading stocks on blocks - engineering decentralized markets. In: HEVNER A. (EDS). *Designing the Digital Transformation. DESRIST 2017. Lecture Notes in Computer Science*. [S.l.], 2017.
- OLIVEIRA, S.; SOARES, F.; FLACH, G.; JOHANN, M.; REIS, R. Building a bitcoin miner on an fpga. In: *XXVII SIM - South Symposium on Microelectronics*. [S.l.: s.n.], 2013.
- OPENZEPPELIN. *Proxy Upgrade Pattern*. 2021. <<https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies>>.
- PALLADINO, S. *EIP-1967: Standard Proxy Storage Slots*. 2019. <<https://eips.ethereum.org/EIPS/eip-1967>>.
- PERMENEV, A.; DIMITROV, D.; TSANKOV, P.; DRACHSLER-COHEN, D.; VECHEV, M. Verx: Safety verification of smart contracts. In: *S&P 2020*. [S.l.: s.n.], 2020. p. 18–20.
- PORAT, A.; PRATAP, A.; SHAH, P.; ADKAR, V. Blockchain consensus : An analysis of proof-of-work and its applications. In: . [S.l.: s.n.], 2017.
- RAJ, R. *What is Solidity?* 2021. Disponível em: <<https://intellipaat.com/blog/tutorial/blockchain-tutorial/what-is-solidity/>>. Acesso em: Mar 13th, 2022.
- RIBERA, E. G. Design and implementation of a proof-of-stake consensus algorithm for blockchain. In: . [S.l.: s.n.], 2018.
- RODLER, M.; LI, W.; KARAME, G. O.; DAVI, L. Evmpatch: timely and automated patching of ethereum smart contracts. In: 30TH USENIX SECURITY SYMPOSIUM. *USENIX Security 2021. USENIX Association*. [S.l.], 2021.
- RODLER, M.; LI, W.; KARAME, G. O.; DAVI, L. Evmpatch: Timely and automated patching of ethereum smart contracts. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021. p. 1289–1306. ISBN 978-1-939133-24-3. Disponível em: <<https://www.usenix.org/conference/usenixsecurity21/presentation/rodler>>.
- RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. *Case Study Research in Software Engineering: Guidelines and Examples*. [S.l.]: Wiley Publishing, 2012.
- SHORISH, J. Blockchain state machine representation. In: . [S.l.: s.n.], 2018.
- SIEGEL, D. *Understanding The DAO Attack*. 2016. <<https://www.coindesk.com/understanding-dao-hack-journalists>> accessed on 22 July 2021.
- SOUZA, E. *Teoria Do Não-conhecimento*. 2015.
- SUNDEVALL, E.; NYSTRÖM, M.; KARLSSON, D.; ENELING, M.; CHEN, R.; ÖRMAN, H. Applying representational state transfer (rest) architecture to archetype-based electronic health record systems. In: . [S.l.: s.n.], 2013.

SWAN, M. Blockchain thinking: The brain as a dac (decentralized autonomous organization). In: *Texas Bitcoin Conference*. [S.l.: s.n.], 2015.

SZABO, N. S. *Smart Contracts: Building Blocks for Digital Markets*. 1996. Disponível em: <[https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart\\_contracts\\_2.html](https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html)>. Acesso em: Oct 29th, 2021.

TIKHOMIROV, S.; VOSKRESENSKAYA, E.; IVANITSKIY, I.; TAKHAVIEV, R.; MARCHENKO, E.; ALEXANDROV, Y. Smartcheck: Static analysis of ethereum smart contracts. In: WETSEB'18: WETSEB'18:IEEE/ACM. *1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. [S.l.], 2018.

TIKHOMIROV, S.; VOSKRESENSKAYA, E.; IVANITSKIY, I.; TAKHAVIEV, R.; MARCHENKO, E.; ALEXANDROV, Y. Smartcheck: Static analysis of ethereum smart contracts. In: IEEE. *WETSEB 2018*. [S.l.], 2018. p. 9–16.

TOLMACH, P.; LI, Y.; LIN, S.-W.; LIU, Y.; LI, Z. A survey of smart contract formal specification and verification. Association for Computing Machinery, New York, NY, USA, v. 54, n. 7, 2021. ISSN 0360-0300. Disponível em: <<https://doi.org/10.1145/3464421>>.

TORRE, D.; SEANG, S. Proof of work and proof of stake consensus protocols: a blockchain application for local complementary currencies. In: . [S.l.: s.n.], 2019.

TORRES, C. F.; JONKER, H.; STATE, R. Elysium: Automagically healing vulnerable smart contracts using context-aware patching. *CoRR*, abs/2108.10071, 2021. Disponível em: <<https://arxiv.org/abs/2108.10071>>.

TSANKOV, P.; DAN, A.; DRACHSLER-COHEN, D.; GERVAIS, A.; BÜNZLI, F. Securify: Practical security analysis of smart contracts. In: CCS '18. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. [S.l.], 2018. p. 67–82.

TSANKOV, P.; DAN, A.; DRACHSLER-COHEN, D.; GERVAIS, A.; BUENZLI, F.; VECHEV, M. Securify: Practical security analysis of smart contracts. In: ACM. *CCS 2018*. [S.l.], 2018. p. 67–82.

VASSANTLAL, R.; ALCHIERI, E.; FERREIRA, B.; BESSANI, A. From byzantine replication to blockchain: Consensus is only the beginning. In: . [S.l.]: IEEE, 2020.

VICTOR, F.; LÜDERS, B. K. Measuring ethereum-based erc20 token networks. In: *Financial Cryptography and Data Security: 23rd International Conference*. [S.l.: s.n.], 2019. p. 113–129.

VOGELSTELLER, F.; BUTERIN, V. *EIP-20: Token Standard*. 2015. <<https://eips.ethereum.org/EIPS/eip-20>>.

VOLLMER, J. *The Biggest Hacker Whodunnit of the Summer*. 2016. <<https://www.vice.com/en/article/pgkzqm/the-biggest-hacker-whodunnit-of-the-summer>> accessed on 22 July 2021.

WANG, D.; WU, S.; LIN, Z.; WU, L.; YUAN, X.; ZHOU, Y.; WANG, H.; REN, K. Towards a first step to understand flash loan and its applications in defi ecosystem. In: *International Workshop on Security in Blockchain and Cloud Computing 2021*. [S.l.: s.n.], 2021. p. 23–28.



- WANG, R.; YANG, K.; YANG, L. Designing hub-and-spoke network with uncertain travel times: a new hybrid methodology. In: *Journal of Uncertain Systems*. [S.l.: s.n.], 2017. p. 243–256.
- WANG, S.; YUAN, Y.; WANG, X.; LI, J.; QIN, R.; WANG, F.-Y. An overview of smart contract: Architecture, applications, and future trends. In: *IEEE Intelligent Vehicles Symposium (IV)*. [S.l.: s.n.], 2018. p. 108–113.
- WANG, Y.; LAHIRI, S. K.; CHEN, S.; PAN, R.; DILLIG, I.; BORN, C.; NASEER, I.; FERLES, K. Formal verification of workflow policies for smart contracts in azure blockchain. In: *VSTTE*. [S.l.: s.n.], 2020. p. 87–106.
- WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. 2014.
- WÜST, K.; MATETIC, S.; EGLI, S.; KOSTIAINEN, K.; CAPKUN, S. Ace: Asynchronous and concurrent execution of complex smart contracts. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. [S.l.: s.n.], 2020. (CCS '20), p. 587–600.
- XU, J.; VADGAMA, N. From banks to defi: the evolution of the lending market. In: SPRINGER. [S.l.], 2022. p. 113–129.
- YERMACK, D. Corporate governance and blockchains. In: *Review of Finance*. [S.l.: s.n.], 2017. p. 7–31.
- YU, X. L.; AL-BATAINEH, O.; LO, D.; ROYCHOUDHURY, A. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, set. 2020. ISSN 1049-331X. Disponível em: <<https://doi.org/10.1145/3402450>>.
- ZHENG, Z.; XIE, S.; DAI, H.-N.; CHEN, W.; CHEN, X.; WENG, J.; IMRAN, M. Future generation computer systems. In: . [S.l.: s.n.], 2020. p. 475–491.
- ZHOU, E.; HUA, S.; PI, B.; SUN, J.; NOMURA, Y.; YAMASHITA, K.; KURIHARA, H. Security assurance for smart contract. In: IEEE. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. [S.l.], 2018. p. 1–5.