Luca Olivieri

Blockchain Software Verification and Optimization

Ph.D. Thesis

Università degli Studi di Verona
Dipartimento di Informatica

Blockchain Software Verification and Optimization

Ph.D. Thesis

Candidate
Luca Olivieri

Supervisor
Prof. Nicola Fausto Spoto

Co-supervisor
Ph.D. Elisa Burato

Thesis Referees
Prof. Étienne Payet

Prof. Samir Genaim

University of Verona, Department of Computer Science

Doctoral Program in Computer Science

Cycle XXXV

S.S.D. INF/01

Università degli Studi di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134, Verona, Italy

I, Luca Olivieri, declare that this thesis titled, *"Blockchain Software Verification and Optimization"* and the work presented in it are my own. I confirm that: (i) This work was done wholly or mainly while in candidature for a research degree at this University. (ii) Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated. (iii) Where I have consulted the published work of others, this is always clearly attributed. (iv) Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work. (v) I have acknowledged all main sources of help. (vi) Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Verona, December 6th, 2022

ACKNOWLEDGMENTS

I would first like to thank and express my sincere gratitude to my advisors Prof. Nicola Fausto Spoto and Ph.D. Elisa Burato both for supporting and encouraging me on this non-trivial journey. These thanks are extended to the people of Corvallis's team for following me over these years and for their valuable advice. I would like to thank also the research groups and researchers with whom I collaborated, with a special mention for the Software and System Verification group at Ca' Foscari University of Venice and Celtique project-team at University of Rennes 1. Last but not least, my thanks go to my family, loved ones, and all the people who have always supported me both in good times and in bad times. Thanks to believe in me and allowing me to follow my dreams and my passions!

ABSTRACT

In the last decade, blockchain technology has undergone a strong evolution. The maturity reached and the consolidation obtained have aroused the interest of companies and businesses, transforming it into a possible response to various industrial needs. However, the lack of standards and tools for the development and maintenance of blockchain software leaves open challenges and various possibilities for improvements. The goal of this thesis is to tackle some of the challenges proposed by blockchain technology, to design and implement analysis, processes, and architectures that may be applied in the real world. In particular, two topics are addressed: the *verification of the blockchain software* and the *code optimization of smart contracts*.

As regards the verification, the thesis focuses on the original developments of tools and analyses able to detect statically, i.e. without code execution, issues related to *non-determinism*, *untrusted cross-contracts invocation*, and *numerical overflow/underflow*. Moreover, an approach based on *on-chain verification* is investigated, to proactively involve the blockchain in verifying the code before and after its deployment.

For the optimization side, the thesis describes an optimization process for the *code translation from Solidity language to Takamaka*, also proposing an *efficient algorithm to compute snapshots* for fungible and non-fungible tokens.

The results of this thesis are an important first step towards improving blockchain software development, empirically demonstrating the applicability of the proposed approaches and their involvement also in the industrial field.

CONTENTS

1 Introduction . . . 1
- 1.1 Research Objectives and Contributions . . . 2
- 1.2 Thesis Structure . . . 2

Part I Background

2 Blockchain Overview . . . 9
- 2.1 Blockchain Structure . . . 9
  - 2.1.1 Consensus Mechanisms . . . 11
  - 2.1.2 The Choice of the Blockchain Network . . . 12
- 2.2 Evolution of the Technology and Historical Outline . . . 13
- 2.3 Conclusions . . . 14

3 Blockchain Software . . . 17
- 3.1 What is Blockchain Software? . . . 17
- 3.2 Programming Languages and Blockchain Development . . . 19
- 3.3 A Taxonomy of General-Purpose Languages For Smart Contracts 21
- 3.4 Limitations and Challenges . . . 22
  - 3.4.1 Full and Restricted Languages . . . 23
  - 3.4.2 Meta-programming Languages . . . 24
  - 3.4.3 Limitations of the Tool Belts . . . 26
- 3.5 Related Work . . . 27
- 3.6 Conclusions . . . 28

Part II Blockchain Software Verification

4 Bug Fixing and Software Verification in Blockchain 31
4.1 Blockchain and Traditional Software Verification 31
4.2 Verification Techniques 32
4.3 Static Analysis by Abstract Interpretation 33
4.3.1 Classification of Alarms 34
4.3.2 Soundness, Abstractions, and Precision 35
4.4 How to Build a Static Analyzer From Scratch 36
4.5 Conclusions 38

5 A Parametric Static Analyzer for Go 39
5.1 The Go Language 39
5.2 An Overview of GoLiSA 40
5.2.1 Parsing and CFG Construction 41
5.3 Related Work 42
5.4 Conclusions 44

6 A Parametric Static Analyzer for Tezos Smart Contracts 45
6.1 The Michelson Language 45
6.2 An Overview of MichelsonLiSA 48
6.2.1 Parsing and CFG Construction 49
6.2.2 Intermediate Representation in SSA Form 49
6.3 Related Work 54
6.4 Conclusions 55

7 Ensure Determinism in Blockchain Software 57
7.1 Non-Determinism in Blockchain 58
7.1.1 Sources of Non-Determinism 59
7.1.2 Sinks of Non-Determinism 60
7.2 Non-Determinism in Go 61
7.3 An Information Flow Analysis Approach 65
7.3.1 An Overview on Information Flow 65
7.3.2 Non-interference 67
7.3.3 Taint Analysis 67
7.4 Detection of Non-Determinism with GoLiSA 68
7.4.1 Detection of Sources and Sinks with GoLiSA 69
7.5 Related Work 72
7.6 Experimental Evaluation 72
7.6.1 Quantitative Study: Hyperledger Fabric Benchmark 73
7.6.2 Qualitative Study 75
7.6.3 Limits 76
7.7 The Industrial Case of Study: Commercio.network 77
7.7.1 The Blockchain Application of Commercio.network 78

7.7.2 Limits of the Cosmos SDK Toolbelt 78
7.7.3 Code Evaluation 80
7.8 Conclusions 84

8 Untrusted Cross-Contract Invocation Detection 85
8.1 Untrusted Cross-Contract Invoking 85
8.2 Detection by Taint Analysis 86
8.2.1 Detection of Sources and Sinks with GoLiSA and MichelsonLiSA 87
8.3 Related Work 88
8.4 Experimental Evaluation 89
8.4.1 Experimental Result Discussion 90
8.5 Conclusions 95

9 Numerical Issues Detection 97
9.1 Numerical Overflow/Underflow 97
9.2 Detection by Numerical Abstractions 98
9.2.1 Implementation in GoLiSA 100
9.3 Related Work 102
9.4 Experimental Evaluation 102
9.4.1 Case Study 104
9.5 Conclusions 105

10 On-Chain Verification of Smart Contracts 107
10.1 On-Chain Code Verification 107
10.1.1 The Architecture over Tendermint 108
10.2 Implementation 112
10.3 Experimental Evaluation 115
10.4 Evolution of Code Verification 118
10.4.1 Governance for Tool Upgrade 120
10.5 Related Work 121
10.6 Conclusions 122

Part III Blockchain Software Optimization

11 Code Optimizations in Blockchain 125
11.1 Blockchain Optimizations 125
11.2 Optimize Code Migration and Translation 126
11.3 Case of Study: Blockchain Token Standards 126
11.3.1 ERC-20 and its OpenZeppelin Implementation 127
11.3.2 ERC-721 and its OpenZeppelin Implementation 130
11.4 Conclusions 132

12 Optimization of Token Standards 137
12.1 From Solidity to Takamaka 137
12.1.1 Snapshot ERC-20 in Takamaka 142
12.2 An Efficient Algorithm for Snapshots 142
12.3 Related Work 147
12.4 Performance Evaluation 148
12.5 Conclusions 149

Part IV Conclusion

13 Final Conclusions and Future Directions 153

References 157

LIST OF ACRONYMS

API	Application Programming Interface
AST	Abstract Syntax Tree
BFT	Byzantine Fault Tolerant
BOS	Blockchain-Oriented Software
CFG	Control Flow Graph
CLI	Command-Line Interface
DApp	Decentralized Application
DSL	Domain Specific Language
ERC	Ethereum Request for Comment
EVM	Ethereum Virtual Machine
GPL	General Purpose Language
HF	Hyperledger Fabric
IR	Intermediate Representation
JVM	Java Virtual Machine
LoC	Line of Code
PKI	Public Key Infrastructure
PoW	Proof-of-Work
PoS	Proof-of-Stake
RPC	Remote Procedure Call
SQL	Structured Query Language
UCCI	Untrusted Cross Contract Invocation

LIST OF FIGURES

2.1 Structure of a generic blockchain. The construction of the structure can undergo forks such as what is coming in the blocks after the Block N+1. 10

3.1 Issues related to meta-programming using SmartPy. 26

4.1 The left column contains the real program points that hold the property of interest in a program, while the right column contains the real ones that do not hold the property. The first row contains the program points reported by the analysis as holders of property, while the second row contains the ones considered by the analysis as non-holders. 34
4.2 Approximation schema. 35
4.3 LiSA overall architecture. 36

5.1 GoLiSA overall execution. 41
5.2 Example of shadowing in Go. (a) A re-declaration of `v` in the block. (b) An assignment to `v` in the block. (c) An example with multiple blocks. 43

6.1 A Michelson smart contract that adds two input integers and stores the sum in the blockchain. 46
6.2 An example of the execution of the smart contract in Figure 6.1. . 46
6.3 MichelsonLiSA architecture. 48
6.4 Generic implementation of the `SUB` symbolic expression of LiSA. `MichelsonSub` is a binary expression with components `left` and `right`. It is both a stack consumer and a stack producer. The template method `binarySemantics` implements its semantics, parametrically *wrt.* the abstract domain. 50
6.5 A Michelson smart contract and its translation into SSA form. . . . 51

6.6 Example of transformation into SSA form. 52
6.7 Michelson code using a `DIG n` instruction and its SSA form representation. 52
6.8 Example of transformation of a conditional into SSA form, with a junction point. The ϕ-function is written as `phi`. 53
6.9 Michelson code that uses a `DIP n` instruction and its corresponding stack execution. Round brackets highlight the protected area of the stack. 54

7.1 Cosmos SDK code affected by CVE-2021-41135. 61
7.2 Example of safe use of time APIs in blockchain software. 61
7.3 Example of issue of non-determinism with time APIs in blockchain software. 61
7.4 Example of issue of non-determinism related to response in blockchain software. 63
7.5 ABCI methods and consensus flow [107]. 64
7.6 Main store of Cosmos SDK. 65
7.7 Example of (a) explicit, (b) implicit, and (c) side channel flows. . . 66
7.8 Method `registrarBoleto` of *boleto* contract. 76
7.9 Commercio.network architecture. 79
7.10 *Bug #1*. A snippet of the `AssignMembership` method not ensuring determinism in the `commerciokyc` module. 82
7.11 *Bug #2*. A snippet of the `BurnCCC` method not ensuring determinism in the `commerciomint` module. 83

8.1 A simplified smart contract for Hyperledger Fabric containing an UCCI. 86
8.2 Simplified code from *chaincode_union_loan*. 92
8.3 Smart contract *expruqYPRHnQyNih8sK1vhNLRBLx37VeuZ3T58SWax-Pj5WwbCQJb2V.tz*. 93
8.4 Smart contract exprthPm93Nt4TBdDSd9LVG829YcgbK9VKE4TRDXt-ZiU8Fv7gFEBod.tz 94

9.1 Go fragment affected by integer overflow. 98
9.2 A set of points (a), and its best approximation in the interval (b), octagon (c), and polyhedron (d) abstract domains [138]. 100
9.3 (a) CFG with interval analysis result of Figure 9.1. (b) Overflow checker report file for the exit node. 101
9.4 (a) A simplified chaincode of `invoke`. (b) Patch for Lines 15–16. . .104

10.1 Tendermint Core and a Tendermint application, with their respective databases. . . . 109
10.2 High-level architecture of an application running on Tendermint Core and performing on-chain verification. . . . 110
10.3 Sequence diagram for code verification and installation in the blockchain. . . . 110
10.4 The evolution of the application state during a sequence of requests. . . . 111
10.5 Some of the 26 on-chain verifications currently performed by Takamaka. . . . 114
10.6 The on-chain check for a correct use of `caller()`. . . . 116
10.7 Sequence diagram for lazy code re-verification. . . . 119
10.8 Schema for new verification rule upgrade. . . . 120

11.1 A portion of OpenZeppelin's ERC-20 implementation in Solidity [152]. . . . 133
11.2 A portion of OpenZeppelin's Solidity ERC-20 contract with snapshots [153]. . . . 134
11.3 A simplified portion of OpenZeppelin's ERC-721 implementation in Solidity [151]. . . . 135

12.1 A portion of our ERC-20 implementation in Takamaka [99]. . . . 138
12.2 A portion of our ERC-721 implementation in Takamaka [100]. . . . 141
12.3 The pseudocode of an alternative implementation of snapshots in Solidity, that its compiler does not accept. . . . 142
12.4 The `snapshot` method added to the code in Figure 12.1. . . . 143
12.5 The UML class diagram of the `IERC20View` and `IERC20` interfaces, implemented by the `ERC20` class. The `IERC20View` interface is a very abstract view of a ledger: it has methods for read-only access and for creating snapshots. . . . 144
12.6 A red/black tree with immutable nodes, with snapshots in $O(1)$. . . 146
12.7 The `snapshot` method added to the code in Figure 12.2. . . . 147
12.8 Result of test simulating ten days of interaction with proposed ERC-20 contract implementations. . . . 149

LIST OF TABLES

2.1 Comparison between Proof-of-Work and Proof-of-Stake. 12

3.1 Target languages for some popular blockchains. 21
3.2 Blockchain classification based on the proposed taxonomy. 22

6.1 Domain-specific operations of Michelson. 47

7.1 Overview of non-deterministic behaviors in Go. Most of the APIs contained in these packages lead to non-deterministic behaviors, but some can be considered safe for determinism. 59
7.2 Main sinks for blockchain software written in Go. The *Critical point* column describes the sensitive point of the method where non-determinism may lead to different executions in different nodes (i.e. in parameters, returned value, or the full statement). . . 63
7.3 Evaluation of analysis for non-determinism detection. 74
7.4 Coverage percentages of different static analyzers. 75
7.5 Warnings triggered by the analyzers on the 46 files. 75
7.6 Analysis evaluation. 80

8.1 HF methods of interest for the detection of UCCIs. 88
8.2 MichelsonLiSA statements of interest for the detection of UCCIs. . 88
8.3 UCCI analysis evaluation for GoLiSA. 90
8.4 UCCI analysis evaluation for MichelsonLiSA. 90

9.1 Results of overflow checker. 103
9.2 Numerical comparison between common results. 103

PREFACE

The present thesis is the result of my research work developed during the three years of my Ph.D. program. Since mine was an industrial Ph.D., the research activities carried out were in agreement with the University of Verona and the company Corvallis s.r.l of Padua. This allowed me to meet both new business and academic realities as well as to enrich myself with new experiences and knowledge. I spent most of the time working on blockchain issues and secondarily on other problems related to privacy and security vulnerability in traditional software. Hence, the thesis focuses only on the core studies related to blockchain technology. Part of the contents of this thesis are published in international peer-reviewed conferences and journals. They have been developed under the supervision of Fausto Spoto, associate professor at the University of Verona, and the Ph.D. Elisa Burato, industrial advisor at Corvallis s.r.l.

Chapter 1

INTRODUCTION

In 2008, the publication of the manuscript *"Bitcoin: A Peer-to-Peer Electronic Cash System"* [142] starts the blockchain era. The author proposes a new paradigm to process financial transactions in a decentralized way, based on a consensus mechanism able to avoid intermediaries, third parties, and issues related to digital cash such as double-spending [13, Chapt. 1]. The core underlying this paradigm is precisely the blockchain, a collection of concepts and technologies deriving mainly from the fields of cryptography and distributed systems [13, Chapt. 9]. Blockchain technology has undergone a sudden evolution thanks to its growing popularity in different areas: from financial transactions [3] to insurance refunds [113], supply chain management [172], anti-piracy campaigns [207], self-sovereign identity [140], etc.

In recent years, the increased knowledge and awareness of the blockchain potential has aroused the interest of companies and businesses, transforming it into a possible response to various industrial needs [128]. Thus, the number of companies using blockchain technology continues to grow.

However, the engineering and tools related to the development and integration of software solutions based on blockchain technology progressed at a lower pace [29, 162]. The development process is far from being standardized in this context. There are currently no proven guidelines for implementing software suitable for blockchain solutions. Traditional software paradigms, such as Software Development Life Cycle (SDLC), fail to guarantee adequate processes for the blockchain development requirements [127]. Therefore, several of these processes are not optimal nor fully automated. Hence, resources such as human effort and time are increased while programming errors are not minimized, worsening the security and quality of the code.

1.1 Research Objectives and Contributions

The purpose of this thesis is to provide solutions for verification and optimization issues related to blockchain software. The analysis and development of these topics are motivated by the lack of significant analyses, processes, and architectures able to face the challenges proposed by blockchain technology in the state of practice.

The first main contribution is the development of two tools to define static analyses for the verification of Go and Michelson languages. These tools are based on the abstract interpretation theory in order to pursue a *sound* approach and to prove the absence of malicious behaviors in the code. Given the rapid evolution of blockchain technology, the state of practice lacks tools capable of guaranteeing the safety and quality of blockchain software.

In this direction, we propose analyses related to non-trivial issues: *non-determinism*, *untrusted cross-contract invocations*, and *numerical overflows*. For the detection of these problems, we analyzed the program semantics without limiting the analyses to syntactic checks only, as is the case in most blockchain framework analyzers instead. In this way, it is possible to design more precise analyzes that allow one to reduce false positives in comparison with other tools used for the detection of the same issue. Moreover, to the best of our knowledge, some of these analyses are novel implementations for Go and Michelson.

Another contribution is related the *on-chain verification*. We describe an alternative paradigm for blockchain verification, where the nodes of the blockchain verify the code being deployed, in order to guarante that all code executed in the blockchain has been successfully verified over time.

Finally, regarding code optimization, we propose snapshot algorithm optimizations to reduce the gas and time costs of executions of token standards within the JVM.

1.2 Thesis Structure

The first part of the thesis provides background on blockchain technology, paying attention to the terminology and defining the concepts discussed in the rest of the thesis:

- Chapter 2 introduces the basic foundations needed to understand the work and the notations adopted in the rest of the thesis, providing a gentle introduction to blockchain technology.
- Chapter 3 investigates the meaning of blockchain software. Moreover, it offers an overview of the programming languages involved in blockchain soft-

ware development, focusing on their diffusion and proposing a **taxonomy of general-purpose languages** about smart contracts, decentralized applications, and underlining the related challenges and problems.

The second part of this thesis deals with the verification of blockchain software:

- Chapter 4 describes the importance of software verification in blockchain context, with attention to code immutability and its decentralized distribution. It introduces static analysis by abstract interpretation as a means of formal verifying software. Moreover, it summarizes the basic concepts related to LiSA, a library to facilitate the development of verification tools on which we relied for the realization and implementation of the analyzers proposed in this thesis.
- Chapter 5 presents GoLiSA, a static analyzer for the Go language. We designed and developed a tool based on abstract interpretation theory able to support several industrial blockchain frameworks and analyze real-world code. To the best of our knowledge, it is the first **analyzer based on LiSA applied to the industrial context**. Moreover, it supports the most popular blockchain frameworks for Go and it is able to perform semantic analyses on real-world applications.
- Chapter 6 describes the design and implementation of MichelsonLiSA, a static analyzer for Tezos smart contracts written in the Michelson language. The analyzer implements an **intermediate representation based on the static single-assignment form and a symbolic stack**, to manage the peculiarities of the memory model of Tezos, and domain-specific instructions.
- Chapter 7 presents a method for the **detection of issues related to non-determinism in blockchain software based on information flow analyses**. The proposed analyses allow one to significantly reduce the false positives generated compared to state-of-the-art tools. This chapter also contains a GoLiSA evaluation and an industrial case study related to Commercio.network.
- Chapter 8 deals with the issues of *untrusted cross-contract invocations*. It also reports our **approach based on taint analysis for the detection of untrusted cross-contract calls**.
- Chapter 9 introduces an on-going work related to the detection of numerical issues, investigating and comparing some numerical abstract domains. Experimental results empirically show the **applicability of specific numerical domains on software blockchains**, which otherwise may be impractical in other software contexts.
- Chapter 10 proposes a definition of ***on-chain* code verification paradigm**, i.e. an approach that involves the blockchain nodes in the code verification. Furthermore, a **lazy re-verification approach** is described to cope with the evolution of code verification rules. Then, the chapter describes an actual

implementation of a blockchain with on-chain verification based on the Tendermint and Takamaka frameworks, including also **governance for tool upgrade management**.

The third part of this thesis describes blockchain software optimizations. In particular, it focuses on optimizing standards for tokens by focusing on translations from one programming language to another.

– Chapter 11 introduces the benefits of code optimization in blockchain and introduces two standards for fungible and non-fungible tokens.
– Chapter 12 presents a process for a **literal translation from Solidity to Takamaka**, then describes **novel implementations for making snapshots of tokens**, based on tree maps, that is possible in Java, but not in Solidity.

The fourth part concludes the thesis:

– Chapter 13 summarizes the work done in the thesis and investigates future research directions.

Publications

Part of the results presented in the thesis have been already published or are under publication in:

– [147] Luca Olivieri, Fausto Spoto, and Fabio Tagliaferro. *On-Chain Smart Contract Verification over Tendermint.* 5^{th} Wokshop on Trusted Smart Contracts (WTSC'21), pages 333–347, Springer, 2021 .
– [51] Marco Crosara, Luca Olivieri, Fausto Spoto, and Fabio Tagliaferro. *Re-engineering ERC-20 Smart Contracts with Efficient Snapshots for the Java Virtual Machine.* 3^{rd} International Conference on Blockchain Computing and Applications (BCCA'21), pages 187–194, IEEE, 2021.
– [148] Luca Olivieri, Fabio Tagliaferro Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. *Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report.* 11^{th} ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'22), pages 23–29, ACM, 2022.
– [50] Marco Crosara, Luca Olivieri, Fausto Spoto, and Fabio Tagliaferro. *Fungible and non-fungible tokens with snapshots in Java.* Cluster Computing, Springer, 2022.
– Luca Olivieri, Thomas Jensen, Luca Negrini, Fausto Spoto. *MichelsonLiSA: A Static Analyzer for Tezos.* Accepted paper at 4^{th} Workshop on Blockchain theoRy and ApplicatIoNs (BRAIN'23), 2023.

– Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, Fausto Spoto. *Information Flow Analysis for Detecting Non-Determinism in Blockchain.* Accepted paper at European Conference on Object-Oriented Programming (ECOOP'23), 2023.

Other side studies are published in:

– [74] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. *BackFlow: Backward context-sensitive flow reconstruction of taint analysis results.* 21^{st} International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'20), pages 23–43, Springer, 2020.
– [30] Marco Bozzetti, Luca Olivieri, and Fausto Spoto. *Cybersecurity Impacts of the Covid-19 Pandemic in Italy.* Italian Conference on Cybersecurity 2021 (ITASEC'21), pages 145–155, CEUR Workshop Proceedings, 2021.
– [75] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. *Static Privacy Analysis by Flow Reconstruction of Tainted Data.* International Journal of Software Engineering and Knowledge Engineering, 31(7):973–1016, 2021.

Collaborations

The research done in this thesis is joint work with several teams. Professor Fausto Spoto and his team at the University of Verona introduced and supported me in the blockchain research field. GoLiSA and related analyses were developed in collaboration with Agostino Cortesi, Pietro Ferrara, Luca Negrini, and Vincenzo Arceri (SSV team of the Ca' Foscari University of Venice, Italy). Moreover, the tool was tested on real-world code by the company Commercio.network. The design and implementation choices of MichelsonLiSA have been extensively discussed with Thomas Jensen, Thomas Genet, and Delphine Demange (Celtique project-team of the University of Rennes 1, France).

Part I

Background

Chapter 2

BLOCKCHAIN OVERVIEW

This chapter provides an overview of the main concepts related to blockchain technology used throughout the thesis. In particular, we introduce the components of the blockchain data structure, how consensus mechanisms work, which ones are the most popular, and the difference among blockchain networks. The chapter concludes by describing how blockchain technology is evolving over time.

2.1 Blockchain Structure

A blockchain is an abstract shared data structure that is *immutable*, *distributed*, and *decentralized*. Figure 2.1 shows its structure in a general shape. It is literally a chain of blocks. The information is regrouped and collected in blocks, each of which contains a certain bounded amount of data records. When a new block is added, it is concatenated with the previous one, thus creating a chain of linked blocks.

To understand how this technology works it is important to know the three main components contained in each block:

- *hash code of block*: an alphanumeric value with a fixed length that uniquely identifies the block. If any of the data related to the block is changed, then the block's hash code also changes.
- *previous hash code*: a reference to the hash code of the previous block.
- *timestamp*: a field specifies the moment in which the block was created and helps to maintain the chronological order of the chain. Typically, it is an incremental numerical value such as a counter.

These three components combined together make the blocks of blockchain tamper-proof. Indeed, modifying a block would also change the hash and times-

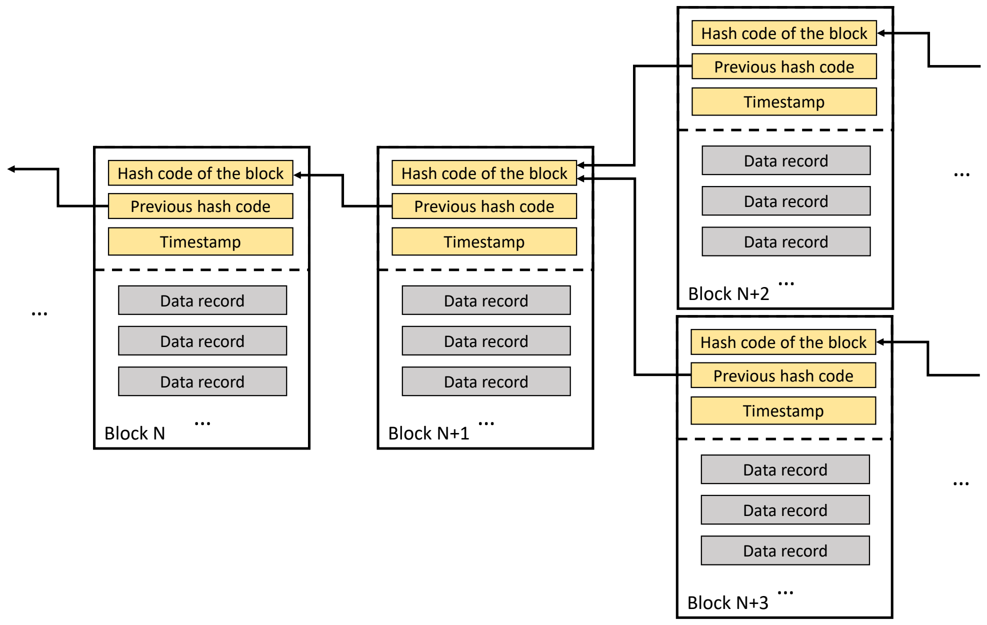


Figure 2.1: Structure of a generic blockchain. The construction of the structure can undergo forks such as what is coming in the blocks after the Block N+1.

tamp of that block. As a result, this leads to a mismatch between the hash codes stored in the blocks to which it is linked, making it immediately apparent that the chain has been altered.

However, the only data structure of the blockchain is not enough to guarantee the security and reliability properties. Hence, it is shared in a peer-to-peer network, a.k.a. blockchain network, where it is possible to add new data through transaction requests, in order to achieve:

- *Distributability of the data*: each peer of the network can keep a copy (full or partial) of the blockchain and approve transaction requests through a consensus mechanism.
- *Decentralization of the data*: the network peers are located in different geographic areas, avoiding single points of failure.
- *Immutability of the data*: the anti-tampering properties, together with the distributability and decentralization, make the data become *"immutable"* (or hardly tamperable).

2.1.1 Consensus Mechanisms

In distributed contexts, it is common to find cryptographic infrastructure algorithms such as *PKI*s (*Public Key Infrastructures*) used for the secure exchange of information. In addition, about distributed data, it is fundamental to achieve a consensus among network peers to avoid inconsistency and to decide what data can be validly added and what data cannot be stored among peers.

The main technological innovation brought by blockchain networks compared to traditional ones is the introduction of an incentive system that allows peers to act collectively in order to guarantee the integrity and security of the network. Blockchain is based on the principle of *trustless*, in which, no one must necessarily trust third parties or individual peers. Trustless does not mean completely eliminating the trust, but rather distributing it in a type of economy that encourages certain behaviors, and punishes the unfair ones [121]. In this way, it added a social component (i.e. a component not related to a computer algorithm, but related to a human peer) that allows one to solve any stalemates and conflicts related to the trust.

The consensus mechanism with rewards and disincentives is the backbone of blockchain technology because it ensures the validity and authenticity of the data stored in blockchain. There are several fault-tolerant mechanisms [1, Chapt. 11] that can be exploited by consensus-based systems, in order to reach a consensus on a single state of a network among distributed peers. Nowadays, the main challenge is to improve the trustness, efficiency, and effectiveness of consensus mechanisms [209] [14, Chapt. 14]. Currently, the most popular paradigms for the blockchain context are *Proof-of-Work* and *Proof-of-Stake*.

Proof-of-Work

The Proof-of-Work (PoW) incentivizes the peers called *miner*s to compete with each other in processing transaction requests, receiving a reward in return (a.k.a. *mining*) [13, Chapt. 10] [14, Chapt. 14]. The competitions consist in solving hard computational problems[1] to validate a new block. At the end of each competition, the winner has the right to add the block to the blockchain and get a reward to incentivize the continuation of the work. In PoW, the punishment consists of the cost of energy required to participate in mining, because computing an hard problem is extremely energy expensive. If participants do not follow the rules and earn the reward, they risk the funds they have already spent on electricity to mine, thus forcing the participants to behave honestly out of self-interest.

[1] Typically, the challenge is to compute an NP-complete problem, where it is easy to check the result is correct, but computationally expensive to calculate the result in the first place. The difficulty of the challenge can also be parameterized by changing over time [19].

Proof-of-Work	***Proof-of-Stake***
The probability of validating a block depends on the computational work performed by a miner	The probability of validating a transaction depends on the stake held by a validator
The first miner that solves the block's challenge gets the reward	There is no block reward, validators earn transaction fees
Miners tend to increase their computational power	Validators tend to increase their stake amount

Table 2.1: Comparison between Proof-of-Work and Proof-of-Stake.

Proof-of-Stake

The Proof-of-Stake (PoS) incentivizes a subset of peers called *validator*s, who have the task of validating and creating new blocks. A peer to become a validator must freeze an economic asset that takes the name of *stake*, such as a certain amount of currencies [14, Chapt. 14]. The consensus algorithm randomly gives the possibility to some validators to validate a block. After an individual check, these validators compare the result with each other and a majority agree on a common result, then earn the transaction fees. Generally, the higher the stake value, the more likely it is that a validator will be selected. Conversely, if a validator submits incorrect results or fraudulent transactions, it will be punished, losing part or all of its stake.

Proof-of-Work vs Proof-of-Stake

Table 2.1 summarizes the main differences in these consensus mechanisms. In recent years, for ecological reasons and to reduce resource consumption [176], the trend is to adopt a PoS in blockchains implemented from scratch and in many other cases to migrate the consensus from PoW blockchain to PoS, such as the migration of Ethereum [78].

2.1.2 The Choice of the Blockchain Network

A critical aspect of enterprise decision-making certainly lies in the network. Blockchain networks are mainly divided into two types of paradigms: *permissionless* and *permissioned*.

Permissionless blockchains (a.k.a. *public* blockchains) provide open networks, have no reference property or actor, and are designed not to be controlled and managed. The peers can join the network without previous authorizations and they can be directly involved in the consensus and data validation process.

Typically, the characterization of these networks is to have a high decentralization, full transparency of transactions, and no central authorities. This implies that these networks provide greater security in terms of points of failure and in terms of consensus as it will be difficult to corrupt most of the networks as they grow. Notable examples are Bitcoin [13,142], Ethereum [14,33] and Tezos [6,85] blockchains.

Permissioned blockchains (a.k.a. *corporate* or *private* blockchains) provide closed networks composed of known peers, such as members of a consortium, which interact and participate together or partially in consensus and data validation. Typically, decentralization is limited in the sense that it is distributed across a restricted number of parties rather than an unknown and potentially unlimited number of participants, as in permissionless blockchains. The key characteristic of these networks is to have the lack of a central authority, which is replaced by a private decentralized group, that has network administrator privileges. In this way, it is possible to achieve greater control of the network, improve blockchain performances, and apply patches and fixes faster than in a permissionless blockchain. Notable examples that allow implementing this type of network are Hyperledger Fabric [12,102] and Tendermint [31,190] blockchains.

Both network paradigms allow for similar value propositions. However, their differences make them more suitable for some use-cases and less suitable for others. Permissionless blockchains tend to be used in contexts with a strong financial component or that require highly decentralized blockchains such as cryptocurrency exchanges, digital assets, crowdfunding, donations, and decentralized autonomous organizations. Instead, permissioned blockchains are favored for applications that depend on confidential data such as supply chain provenance tracking, claims settlement, and identity verification.

2.2 Evolution of the Technology and Historical Outline

The idea behind blockchain technology has been strongly inspired by Haber et. al. [21, 92]. The problem to be faced by the authors was how to certify when a document was created or lasted modified. In [92], as solution, they theorized the first primordial description of cryptographically data structure that chained together (i.e. a *chain*) the hash values of a digital document and time-stamping service, ensuring the immutability of data. Haber et. al [21] introduce Merkle tree [132] within the data structure, merging many unnoteworthy time-stamping events into one noteworthy event, like merging data within a single *block*.

The first application of blockchain technology as it is known today is to be attributed to the implementation of Bitcoin [13, 142], in 2008. The platform allows exchanging digital currency called *bitcoin* through a peer-to-peer network, using a blockchain-based on PoW consensus as a distributed ledger where to store and track transactions. A Turing-incomplete low-level language specifies Bitcoin's transactions. It can be seen as a limited scripting language for smart contracts, i.e. contracts implemented through a programming language and run as distributed software.

In 2013, the second killer application of blockchain is born, namely *Ethereum* [14, 33]. It introduced a Turing-complete bytecode for smart contracts, developing decentralized applications [14, Chapt. 12]. Turing-completeness allows one to express the power of modern programming languages, thus increasing the potential and the use of smart contracts. Ethereum smart contracts can be programmed in high-level languages such as Solidity, the most popular one, and run on the Ethereum Virtual Machine (EVM). Also, it introduces the concept of *gas* [14, Chapt. 1], a parametric transaction fee paid to execute a smart contract within the blockchain network and avoid non-termination. About the consensus mechanism, Ethereum uses PoW but is currently switching to PoS [78].

The third technological evolution still underway begins in the following years. Blockchains, such as *Tendermint Core* [31, 117] (recently rebranded as *Ignite* [106]) and Polkadot [159], expands the concept of blockchain network to real ecosystems populated by different blockchain networks capable of interacting with each other, implementing common inter-blockchain communication protocols [164]. Another interesting contribution of *Tendermint Core* is to provide a generic and customizable infrastructure, leaving the notion of transaction unspecified. The infrastructure is split into three layers (application, consensus, and networking) and a Byzantine Fault Tolerant (BFT) middleware separates the application logic from the consensus and networking layers. This allows one to develop blockchain applications written in any programming language (that supports Remote Procedure Calls), and replicate them on many machines [32]. About the transaction notion, programmers can develop an application layer specifying programmatically which transactions exist and which are their semantics, redefining them according to the use-cases.

2.3 Conclusions

In this chapter, we explained how blockchain technology works and has evolved. Furthermore, to create trustless networks for blockchains, it is necessary to apply both software and economic techniques to guarantee the network's security through consensus mechanisms based on incentives and disincentives. Moreover, in addition to being a data collection, the blockchain is also a technology capable of running software uploaded by the participants and users of the blockchain network, thus significantly increasing its versatility and potential. In the next Chapter, the issue of blockchain software is addressed.

Chapter 3

BLOCKCHAIN SOFTWARE

Many people use the term *blockchain software* for different things. The first that comes to mind for blockchain software is smart contracts, i.e. programmable and executable code within the blockchain. However, this is only a small part of it and there are several software definitions that may take the name of blockchain software. For this reason, it is necessary to initially define the various types of software involved in blockchain technology on their own terms. In addition, some of these misuse terms help to confuse the reader. This chapter defines the various types of software involved in blockchain technology in their own terms.

3.1 What is Blockchain Software?

Blockchain can be thought of as a complex ecosystem, where the software is the component used to build the system and to implement interactions with it. In this context, the software can be categorized in *blockchain software* and *blockchain-oriented software*. As recalled by the names, both categories involve blockchain technology. The main differences are where the software is located and executed, as well as having a different purpose.

Blockchain software includes all the code present within the blockchain, i.e. the implementation code of the blockchain itself and the one contained in the database of blocks. Even if there is not always a clear distinction the code related to blockchain implementation can generally be divided into five layers:

- the *Hardware layer* contains the software to work at a low-level with the hardware of devices. The blockchain network consists of different devices supported by heterogeneous hardware. Typically, the code of this layer can be involved to virtualize the hardware and making the blockchain software platform independent. Otherwise, it is exploited to optimize the performance of an ad-hoc hardware architecture such as in the case of Internet-of-Things devices [131].

- the *Data layer* contains the implementation of a database of blocks with its primitives. Here the data and block structure, how to add the data, and any additional information to be included in the data collection are programmatically defined.
- the *Network layer* deals with the communications of the blockchain network, such as the P2P protocols, etc. It allows one to connect the various peers, handle transactions, and propagate information across the network.
- the *Consensus layer* implements the logic of the chosen consensus mechanism in order to validate or reject the data to store in the blockchain agreeing with the blockchain network.
- the *Application layer* contains the code to manage the content of transactions, propose the updating of the database of blocks with new data, and perform additional operations.

Smart Contract means different things depending on the context. As reported in [14], the term was coined by Nick Szabo and defined as *a set of promises, specified in digital form, including protocols within which the parties perform on the other promises*. However, its original meaning is blurred given the genericity of the software that can run within modern blockchains. Hence, we use the term *smart contracts* to refer to immutable computer programs that run within a blockchain. A smart contract is also considered part of blockchain software. Indeed, it needs to be deployed in the blockchain before being executed, i.e. stored in the database of blocks. Then, the contract can only be run when called by a transaction request. Typically, the execution is performed by a smart contract framework located in the application layer.

Blockchain-Oriented Software (BOS), according to Porru et al. [162], includes all software working with an implementation of a blockchain. That is software that interacts directly or indirectly with the blockchain but is located and executed outside the blockchain. Here, for example, the software can range from generic applications that use blockchain only as data storage, to applications that actively interact with smart contracts, e.g. wallets, crypto-currency, asset exchangers, etc.

Decentralized Application (DApp) refers to an application that is executed by multiple users over a decentralized network, such as the blockchain network. This definition broadly includes both blockchain software and the BOS definitions. Indeed, users are not necessarily peers of the blockchain network. In general, the decentralized application has an external interface with which it can communicate and receive information. For instance, in blockchain, popular DApps are decentralized autonomous organizations [93] and decentralized games [135, 136].

3.2 Programming Languages and Blockchain Development

The popularity and widespread diffusion of a programming language have fundamental implications in the economic context. In general, it means reducing costs and saving time, because fluent developers can quickly apply their expertise to a different domain. Several programming languages are involved in the blockchain context, both DSLs and GPLs.

However, the adoption of GPLs must be considered with care. The familiarity of a programmer with a language is not sufficient to justify the adoption of GPLs in the programming of blockchain software. Indeed, it is a weak assumption to argue that it is an advantage if developers write software in the language they already know. Nowadays, applications are implemented with multi-language solutions and senior developers are proactive in learning new languages. Moreover, according to Deursen et al. [60], DSLs are typically able to ensure some guarantee (restrictions, expressiveness, abstractions, etc.) on a specific problem domain that might not be reached using the full features of a GPL.

In blockchain technology, the payoff is minimal in terms of cost if the language exposes the company to severe economic consequences, such as the immutable deployment of a vulnerable contract that manages cryptocurrencies or financial transactions. Indeed, although only the code of the smart contracts is immutable, the remaining blockchain software, i.e the code of all layers, is also difficult to patch because it is still part of a distributed and decentralized environment formed by a network of potentially untrusted peers. Therefore, every blockchain software update could be critical.

The choice of the language should be based on the project to be implemented or on the type of functionality that the language makes available to extend, maintain and evolve the code repository. According to Deursen et al. [60], the adoption of DSLs leads to benefits and disadvantages.

A critical point is surely the difficulty of balancing between DSLs and GPLs constructs and guarantees. Let us consider Solidity for the Ethereum blockchain, one of the most known DSLs in the blockchain context for smart contracts. The purpose of Solidity was to create a DSL that could be safer and more easily verifiable than a GPL. However, its naive design exposed it to multiple vulnerabilities [15], which were exploited to perform fraudulent actions such as in the case of the DAO attack [161]. Solidity is a high-level language with syntax and semantics that are very close to programming languages such as Java, C++, and Go. However, Solidity is not specialized enough and also not high-level (abstracted) enough. Furthermore, its specialization can be easily replicated in GPLs, such as using Java [51, 188]. Hence, if DSLs and GPLs are similar, it is easier for developers to choose the latter. In that sense, a widespread GPL is often supported by wider communities, which consequently leads to many studies, research, and development of many tools (debugging, monitoring, analyzers, etc.), libraries, software utilities, and IDEs. Nevertheless, as we will see in Section 3.4, this leads to other challenges. Therefore, it is not surprising that various industrial blockchain solutions, such as Hyperledger Fabric [103], Cosmos SDK [144], and Tendemint [31] use mainly the GPLs.

Another key concept to highlight is that there is nothing similar to a standard DSL for programming smart contracts and DApps. For instance, in the database context, there are several Relational Data Base Management Systems, but most of them use the same Structured Query Language (SQL) as the standard language for database development. The various development frameworks can contain many variants of SQL (such as MySQL, PostgreSQL, etc.), which implement additional methods, instructions, and macros to facilitate the developer, yet maintain the common features of the language. However, by providing the same common instruction set of SQL, they make it easy to migrate the code to other systems, with minor changes. In industrial realities, the code has to be reused or the code has to be migrated from one system to another. The same occurs in the BOS, i.e. software working with an implementation of a blockchain [162]. Indeed, the target blockchain could change in favor of another. This can happen for many reasons, for example changing the visibility context of the blockchain such as from a private or corporate to a public one, or vice versa. The visibility change can drastically impact the performance of a blockchain network. For instance, frameworks like Hyperledger Fabric, designed for private or corporate solutions, are generally better performing than public ones such as Ethereum [17]. But the formers require fewer guarantees and fewer constraints in terms of network reliability, availability, and peer trust. In the absence of standards, GPLs allow one to facilitate the portability and re-usability of code. As shown in Table 3.2, there are many frameworks targeting different blockchains that are covered by the same GPL. Although each framework has certain differences, the core logic of smart contracts is typically implemented in the same way by using the same programming language, or at most, it requires only small tricks and fixes to be re-used in other frameworks.

In the enterprise field, the trend that leads to the use of GPLs for the development of blockchain software is therefore not attributable to a simple answer. Currently, the mix between the still low maturity of blockchain DSLs, the lack of standards, the scarcity of supporting tools, and the need for custom requirements bring the cost/benefit ratio towards GPLs.

Blockchain	**Target Languages**
Algorand [36]	Transaction Execution Approval Language (TEAL)
Cosmos [118]	Any language
EOSIO [66]	WebAssembly
Ethereum [14]	Ethereum Bytecode
Hotmoka [96]	Java
Hyperledger Fabric [103]	Go, Java, Javascript
IOTA [160]	WebAssembly
Lisk [125]	JavaScript
Neo [143]	Neo Execution Format (NEF)
Polkadot [159]	WebAssembly
Solana [204]	Solana BPF Bytecode
Tezos [6]	Michelson language

Table 3.1: Target languages for some popular blockchains.

3.3 A Taxonomy of General-Purpose Languages For Smart Contracts

Regarding smart contracts and DApps implementations, GPLs are exploited in different ways depending on the blockchain.

It is necessary to consider the language in which the code is written by developers and which will be the language that gets executed in a distributed manner within the blockchain. Indeed, many programming languages can be used at a high-level, but blockchains could support only a few target languages for executing programs on the network (Table 3.1).

In order to reason about that, it is possible to classify the various GPLs involved in the development of smart contracts and DApps into three macro categories:

- *Full language*: the code is written in a GPL without restrictions and the blockchain uses the same GPL as the target language during the code execution.
- *Restricted language*: the code is written using a restricted subset of a GPL and it is the same restricted target language used by the blockchain during the code execution.
- *Meta-programming language*: the code is written in a language that generates a program in another language. Basically, this happens when a code is written in a GPL and after a framework or a compiler translates/compiles it to another language used by the blockchain as the target language.

Table 3.2 shows a classification of GPLs in the blockchain context based on these macro categories. For instance, the GPLs involved in the development of

Blockchain	**Framework/SDK**	**Languages**	**Typology**
Algorand	PyTeal	Python	Meta-programming
Cosmos	Tendermint Core	Any language	Full language
	Cosmos SDK	Go	Full language
	CosmWasm	Rust	Meta-programming
EOSIO	EOSIO Contract Development Toolkit	C++	Meta-programming
	EOSIO SDK	Javascript, Swift, Java	Meta-programming
Ethereum	Ethereum SDKs	Dart, Delphi, C#, Go, Java, JavaScript, Python, Ruby, Rust	Meta-programming
Hotmoka	Takamaka	Java	Restricted language
Hyperledger Fabric	Hyperledger Fabric SDKs	Go, Java, Javascript	Full language
IOTA	IOTA	Rust, Go	Meta-programming
Lisk	Lisk SDK	JavaScript	Full language
Neo	Neo SDK	Python, C#, Go, TypeScript, Java	Meta-programming
Polkadot	Ink!	Rust	Meta-programming
Solana	Solana SDK	Rust, C, C++	Meta-programming
Tezos	SmartPy	Python, OCaml, TypeScript	Meta-programming
	LIGO	OCaml, Javascript, Pascal, ReasonML	Meta-programming

Table 3.2: Blockchain classification based on the proposed taxonomy.

smart contracts and DApps in *Hyperledger Fabric* are classified as full languages, because they allow one to write and execute code by using the same language, without any restriction or modifications. While for *Hotmoka*, the GPL is Java but *Takamaka* allows one to use only a restricted subset of its features and instructions, in order to guarantee some properties to the code execution [187]. Next, other blockchains such as *Tezos* allow one to write code in a high-level GPL and run a compiled version of it in another target language supported by the blockchain. There are also cases like *Polkadot* with *Ink!*, that use a subset of Rust, i.e. a GPL that compiles into another target language, i.e. WebAssembly. In terms of behavior, this is much closer to the meta-programming language than to a restricted language.

3.4 Limitations and Challenges

Despite GPLs being more mature in terms of development and support, the main problem is that they were not initially designed for the development of smart contracts and DApps, exposing developers to several limitations and challenges. This section proposes some food for thought related to the given taxonomy.

3.4.1 Full and Restricted Languages

The classification of using *full* and *restricted* languages arises from two different schools of thought. The first one argues that it is the developer that should be in charge of the quality and safety of the code and, in turn, the implications it may have while using the language at its full expressiveness. Instead, the second one applies restrictions to the full language in such a way that the developer is protected from unexpected behaviors or known vulnerabilities that may arise when using the full language. However, these restrictions could limit development by denying useful implementations. Full languages are typically involved in controlled environments such as permissioned blockchains, where only identified users can perform only specific actions granted to them by the blockchain administrators. Instead, restricted languages can be involved also in permissionless blockchains, where peers can develop and deploy code without asking for any permission since the restricted language forbids a priori specific features of the full language and it is more likely to prevent the blockchain compromise.

New Issues Related to Full Languages

The use of a full language in the frontier of smart contracts and DApps development leads to the emergence of new and challenging issues, which usually do not affect DSLs because they have already been taken into consideration in their design phase. For instance, let us consider the problem of non-determinism. Intuitively, a non-deterministic value involved during the update of the global state of the blockchain leads to a consensus issue. Specifically, the involved transaction fails because the blockchain validators are unable to agree on a common value that should be updated in the blockchain. This is the case of code invocations that return the current time of the local machine, since this value may vary between the different blockchain validators, leading to a consensus failure. For instance, both *Ignite* and *Hotmoka* support the GPL Java, where several APIs can be non-deterministic. *Hotmoka* allows the usage just of a restricted subset of Java libraries, forbidding the use of features not suitable for smart contracts [187], such as random value functions, disk writing, reading, internet connections, etc. These restrictions are applied a priori and developers will not be able to lift them. However, as argued in [148], non-determinism is unsafe in the blockchain context only if it is *global*, that is if it can affect the global state of the blockchain. This means that certain use of non-determinism is still safe even in blockchain software development. For instance, let us suppose that something must be logged on a blockchain node in your local machine, while the software is running. In the case of *Hotmoka* smart contracts, this will not be possible since the use of local time is not allowed due to the imposed restrictions, while it would be possible in the case of *Cosmos*, where developers are allowed to use the full expressiveness of the Go language. However, in *Ignite* and also in *Cosmos* DApps, developers have to worry about verifying, in a similar case, that the time plotted in the log does not end up in the global state of the blockchain. Tools using formal verification techniques can be used to detect this kind of problem.

Code Re-engineering

Given the flexibility and maturity of GPLs, it is possible to re-engineer the code and standards already implemented using DSLs to obtain technological advantages and develop a more efficient code in terms of performance and gas consumption in other blockchains. A trend in the blockchain is to apply standards from platform to platform, easing the design challenges with trusted and widely-used specifications. In particular, in [51], we show how re-engineering, in an efficient way for the Java Virtual Machine, an implementation of the ERC-20 standard with snapshot for fungible tokens from Solidity to Takamaka (*restricted* language based on Java) using a data structure based on tree maps. However, this approach is typically possible only using *restricted* or *full* languages, because, in *meta-programming*, the high-level optimizations have no effect unless supported by the target language.

3.4.2 Meta-programming Languages

Meta-programming is widely used in several blockchains because typically it allows one to program in several popular high-level languages and then compile or translate them into a single target language, generally at low-level.

Information Loss

The switch from a high to a low-level language can imply a loss of information, leading to difficulties in understanding, reverse engineering, analyzing, and verifying blockchain software. The high-level languages for their nature tend to abstract semantics through compact instructions, types, annotations, etc. Instead, low-level languages have a restricted set of instructions and must explicit all the operations to perform during the execution, losing expressiveness and increasing the code verbosity. For instance, this is a well-known problem in *WebAssembly*, because the recovery of high-level function types from WebAssembly binaries are challenging [122].

Translation and Semantic Issues

As already reported above, meta-programming involves at least two languages, a source, and a target language. If the languages are different, then translation problems may occur, as the semantics of certain operations may not be translatable from one language to another. In this scenario, an interesting case of study is SmartPy and the Michelson language. SmartPy is a framework to develop Tezos smart contracts in Python. While it supports many Python APIs, there is no direct compilation to Michelson from them. Hence, within the compilation into Michelson, to overcome this problem, these functions are resolved [183] and the results are hardcoded in the Michelson compiled program. Let us explain the problem by means of the example reported in Figure 3.1a, which reports a smart contract example written using SmartPy v0.11.1. The smart contract allows to be initialized with a numerical parameter `myParameter1` using the function `__init__` at Line 6, and after can change that value using the function `myEntryPoint` at Line 11. However, the `myEntryPoint` function relies on `random.randint` at Line 13, a standard Python API that cannot be translated in Michelson, since it does not support instructions to generate random numerical values. In this case, the SmartPy framework allows the compilation of this Python contract without any warning message. Figure 3.1b shows the Michelson code retrieved after the compilation of the Python smart contract reported in Figure 3.1a. It is worth noting that `random.randint` is resolved within the compilation and its evaluation, in this case, the value `7`, is hard-coded in the Michelson compilation output, at Line 8. The problem here is that, when running the Michelson code in the blockchain, it will not add a random value, as expected by the Python program, but it will always add the constant value `7`. It is worth noting the compiled Michelson program may differ from one compilation step to another one; the value to which a random function used in the Python code is resolved by the compiler and may vary at each compilation step. Moreover, this makes complex reverse code engineering because pieces of Python information are lost during the compilation process.

Target Language Issues

Another issue concerning meta-programming is related to the fact that the target language might not be a DSL designed for the blockchain context. Hence, it does not necessarily place guarantees on the execution of the code, which is not checked during the meta-programming phase and can lead to issues, bugs, and vulnerabilities. Let us consider the case of *WebAssembly*. As reported in the documentation [202], it is designed to enable high-performance applications on the Web. It was later adopted to run on different blockchain platforms for the following key factors: a high-performance execution, a compact representation, and platform independence. Although it has several positive sides to being adopted in the blockchain context, the language is still exposed to potential risks, such as non-determinism [200] or numerical overflow [135, 136]. Another pitfall of meta-programming includes the fact that a certain program that is compiled

```
import smartpy as sp

import random
# A class of contracts
class MyContract(sp.Contract):
def __init__(self,
    myParameter1):
self.init(myParameter1=
    myParameter1)

# An entry point, i.e. a
    message receiver
# (contracts react to messages
    )
@sp.entry_point
def myEntryPoint(self):
self.data.myParameter1 +=
    random.randint(0,10)
```

(a) Python code

```
parameter (unit %myEntryPoint)
    ;
storage int;
code
{
  CDR;          # @storage
  # == myEntryPoint ==
  # self.data.myParameter1 +=
      7 # @storage
  PUSH int 7; # int : @storage
  ADD;          # int
  NIL operation; # list
      operation : int
  PAIR;         # pair (list
      operation) int
};
```

(b) Michelson Code

Figure 3.1: Issues related to meta-programming using SmartPy.

from a GPL to the target language may not preserve the same semantics. For instance, suppose to use the Python3 language for the meta-programming and WebAssembly or Ethereum Bytecode as the target language. Python3 allows one to represent integers potentially in an unlimited range [79]. Instead, WebAssembly and Ethereum Bytecode support a priori bounded integers [77, 201]. This means that the translation or compilation of some instructions related to integer values could fail, or in the worst case the compilation could be successful but the execution of arithmetic operations could be subject to vulnerabilities, as happened in the case of the EOSIO blockchain [135, 136], which was affected by such an integer overflow vulnerability.

3.4.3 Limitations of the Tool Belts

According to Destefanis et al. [59], compared to traditional software, smart contracts and blockchain software engineering is not yet sufficiently developed. Contracts and DApps rely on a non-standard software life-cycle [127], because of the main peculiarities of the blockchain (immutability, decentralization, and distributability) which make bug fixing and code patching more difficult. Moreover, although the GPLs have wide toolbelts, often these tools are designed for general use and do not include specific features for the blockchain context. In addition, the use of these tools in the blockchain context without supporting its peculiar features may lead to critical implications. Let us consider the verification phase of a software life-cycle. If a verification tool does not properly model blockchain software, it cannot best detect issues related to the blockchain context, leading to incomplete or incorrect checks. According to Section 7.7.2, the risk of using unsuitable tools is to give a false sense of trust to the developers, which could let their guard down. At best this will be detected in later stages, however creating delays and increasing the costs for the fix. In the worst case, the deployment in blockchain could occur and the vulnerability could be exploited maliciously without any possibility of a fix.

3.5 Related Work

The aim of this chapter is to classify and discuss languages for blockchain software development. To the best of our knowledge, there are very few academic studies that classify blockchain programming languages, especially with regard to GPLs. Foschini at el. [76] provide an overview of GPLs involved in the development of *chaincodes* (also known as smart contracts) for Hyperledger Fabric, from the point of view of performance. Varela-Vaca at el. [196] propose a mapping study and a general snapshot of the languages for smart contracts, emphasizing the importance of grey literature (i.e. white papers, reports, documentation, working papers, etc.).

Some research papers investigated and surveyed DSLs, which we present in the following, without explicitly referencing GPLs. Halder et al. [2] investigate state-of-art DSLs introduced in literature since 2015, also providing a comparative analysis based on their application target, development stage, and offered features. In [155], the authors provide a survey on programming languages used for smart contract development, focusing on three DSLs, namely Solidity, Pact, and Liquidity, focusing on their usability and security aspects. Seijas at el. [177] overview scripting languages used in existing cryptocurrencies, focusing on the ones of Bitcoin, Nxt, and Ethereum. Zou et al. [210] analyze the current state and potential challenges developers are facing in developing smart contracts on blockchains, highlighting the limitations of DSLs such as Solidity.

Even if it does not explicitly investigate programming languages used to build blockchain software, the research community had proposed several studies and surveys about the security of blockchain. In the following, we report some related work on this topic. In [91], the authors discuss different and present technologies embedded in blockchains, such as consensus algorithms, public key cryptography, and hash functions used in the blockchain, with a focus on their security aspects, providing a survey about the types of attacks that had affected blockchains, the state-of-art analysis tools that have been proposed during the years for those attacks, and qualitative comparison between these tools, based on the number of vulnerabilities detected. Similarly, the authors of [82] surveyed the blockchain technologies until the end of 2019, presenting also the more popular blockchain context applications. Even [24] proposes a survey paper about security aspects of blockchain, such as blockchain availability and integrity, but the survey focuses on the use of blockchain within information systems. As far as information systems are concerned, [123] reviews blockchain security in this context at three different stages, at the process, data, and infrastructure levels, drawing future research directions on blockchain security, with a particular focus on business and industrial-related issues.

3.6 Conclusions

This chapter sheds more light on the definition of blockchain software, classifying it and adding a taxonomy of the programming languages involved in the development of smart contracts. In addition, it discusses the strengths and weaknesses of each category of smart contract languages.

In the rest of the thesis, we focus only on blockchain software related to application layers, smart contracts, and DApps. In particular, we deal with the blockchain platforms and frameworks Hyperledger Fabric, Tendermint Core, Cosmos SDK, Tezos, and Takamaka. Chapters 5, 7, 8, 9 propose a verification tool and analyses for blockchains written in Go. Chapters 6, 8 deal with the Michelson language. Chapters 10, 12, respectively, describe a verification architecture and optimization processes applied to Java smart contracts. While, Chapters 11, 12 deal with token optimizations and translations from Solidity language.

Part II

Blockchain Software Verification

Chapter 4

BUG FIXING AND SOFTWARE VERIFICATION IN BLOCKCHAIN

Blockchain software is executed by a network of mutually distrusting peers, without any external trusted authority. For this reason, the implementations must be secure against attacks and bugs which lead to data tampering, system instability, and unexpected execution behaviors. In this thesis, we focus only on the verification of blockchain software at the **application layer** (Section 3.1), because it is the one that generally requires more custom development and also includes the smart contract part. In this chapter, we describe the differences between the verification of blockchain code and the traditional one, focusing on static analysis techniques that allow one to analyze the code before its execution, i.e. before deploying it on the blockchain. We also introduce LiSA, a framework to build verification tools from scratch.

4.1 Blockchain and Traditional Software Verification

Traditional software might not be written by taking complete care of quality and safety, but it can still be assessed and improved later. However, this is not practicable in the blockchain, which is distributed with immutable data. For this reason, it is recommended to apply software verification by design. Moreover, this factor gets more problematic by the lack of best practices and standard architectures [29, 127, 162].

In the case of *permissionless* blockchain, when a bug is found and faults happen because of it, the caused problems are generally *immutable* and cannot be fixed. This immutability is achieved through a consensus mechanism that makes it difficult, or impossible, to withdraw transactions from the blockchain. In practice, the network majority should agree on rolling back to the state before an incident, effectively rewriting the history of the blockchain. However, the more a blockchain network increases in popularity, the more it is tricky to undertake such a turnaround. Part of the network may be interested in ignoring to resolve the consequences of a fault and continuing as if nothing happened, leading to an independent blockchain, also known as a *hard fork*. For instance, this happened in Ethereum blockchain because of the DAO attack [161].

In the case of *permissioned* blockchains, it is possible to patch a buggy code through network governance. Typically, there is a limited subset of peers, with the power to propose a plan for halting, modifying, and restarting the blockchain with updated software, carefully migrating the state of the previous version. This kind of blockchain solution is often adopted in the industrial field, especially for enterprise or consortium blockchains. However, enforcing an update leads to a stop of services and data management problems. For instance, a blockchain such as Cosmos to mitigate these problems offers an automatic process to apply blockchain upgrades, improving the synergy between the on-chain module `upgrade`, responsible for halting the chain, and an off-chain daemon capable of installing a new binary of the node software at the right time and autonomously restarting the node.

In any case, it is of substantial importance to detect any kind of problem as soon as possible and, above all, before the software is used by the peers of a blockchain network.

4.2 Verification Techniques

Code verification is a process that is applicable since the beginning of implementation and reduces the issues of the final software product. Programming bugs are unavoidable and it is not advisable to rely only on the skill of the programmer who is prone to errors (e.g., lack of information, knowledge, distractions, etc.). In blockchain, it becomes more significant meaning because it applies before the deployment in blockchain and therefore before the code is immutable or difficult to patch.

According to Chess et al. [38], the most used approach to find bugs is *dynamic testing* which executes the software and compares the output and the expected results. However, this type of dynamic analysis has several drawbacks. Creating test cases is not trivial activity and can require a lot of effort since developers need to compute the expected results on each input case. This is also associated with *unit testing* and goes to verify small portions of code (i.e. the *units*), with normal and special inputs that could generate errors. According to Rival et al. [169, Chapt. 1.4.1], testing can observe only a finite set of finite program executions. For instance, it is not feasible to test a program that takes an arbitrary natural value as input for each input value, because natural values are infinite and the test would not terminate. Hence, this solution can be used only to show the presence of bugs, but never to show their absence [53, Part I, Chapt. 3]. In addition, dynamic analysis is applicable from an advanced stage of development as it needs to be executed, increasing the cost of bug fixing in case of multiple bugs.

A complementary approach to dynamic analysis is *static analysis*. It automatically verifies the properties of computer programs before their execution [169]. This solution reduces the cost of bug fixing for developers, giving them the chance to fix bugs and code smell in an early stage [38].

For software coverage and to prove the presence or absence of a property, such as a bug, it is necessary to use formal methods based on mathematical frameworks [41, 47, 115].

4.3 Static Analysis by Abstract Interpretation

This section introduces the basic concepts of static analysis by abstract interpretation, as well as common notations used in the rest of this thesis. More in-depth introductions to abstract interpretation have been written by Cousot [158] and Rival et al. [169].

According to Cousot [158, Chapt. 1.2], abstract interpretation [47, 48] is a unifying theory of formal methods that proposes a general methodology for proving the correctness of computing systems. In addition to the formalization of systems, it also allows one to discuss the guarantees they provide, such as *soundness* and *completeness* (Section 4.3.1).

Abstract interpretation is based on approximations. In static program analysis, it is used to approximate concrete behaviors of a program (*concrete semantics*), by an abstract version of them (*abstract semantics*). It also formalizes the intuition that semantics are more or less precise depending on the abstraction level. The idea behind abstract interpretation is that reasoning on the abstract properties implies some reasoning on the concrete ones. The abstraction is a necessary step to perform analyses that are able to detect properties in the concrete world. Indeed, as proved by Rice's theorem [168], it is undecidable to reason about non-trivial program properties on concrete semantics. Therefore, reasoning by abstraction allows one to acquire decidability while sacrificing the precision of observed concrete objects.

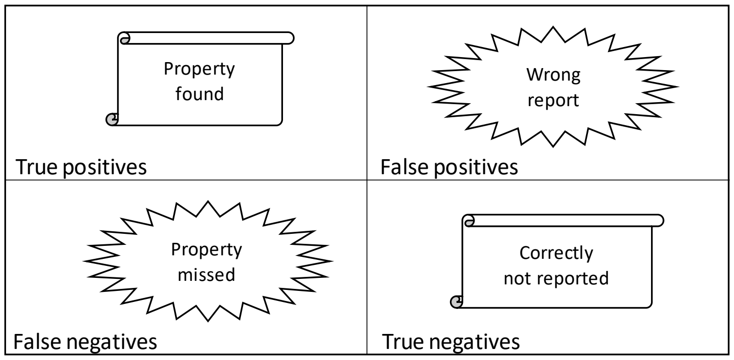


Figure 4.1: The left column contains the real program points that hold the property of interest in a program, while the right column contains the real ones that do not hold the property. The first row contains the program points reported by the analysis as holders of property, while the second row contains the ones considered by the analysis as non-holders.

4.3.1 Classification of Alarms

Analyzers based on static analysis issue alarms at program points where the property chosen for the analysis might occur at runtime, such as when a security vulnerability might be exploited, a specific bug occurs during the program execution, etc. However, the analysis results must be checked to confirm whether they are correct or not in reality. Figure 4.1 proposes a classification of possible behaviors:

- *true positives*: the property is held in reality and the analysis reports the property (the analysis detects correctly the property)
- *false negatives*: the property is held in the reality, but the analysis does not report the property (the analysis miss to detect the property)
- *false positives*: the property is not held in the reality, but the analysis reports the property (the analysis produces a wrong report)
- *true negatives*: the property is not held in reality and the analyzer does not report the property (the analysis correctly does not detect the property).

Ideally, a verification tool should feature true positives and true negatives only. However, this is not computationally possible in general [168]. This implies that a tool has to sacrifice one of the two goals.

A static analyzer is *sound* with respect to a program and a property of interest when it considers all possible program executions and is thus able to give definite guarantees on the property of interest. Thus, if it does not issue any alarm, the property is guaranteed to hold on to any possible execution. In other

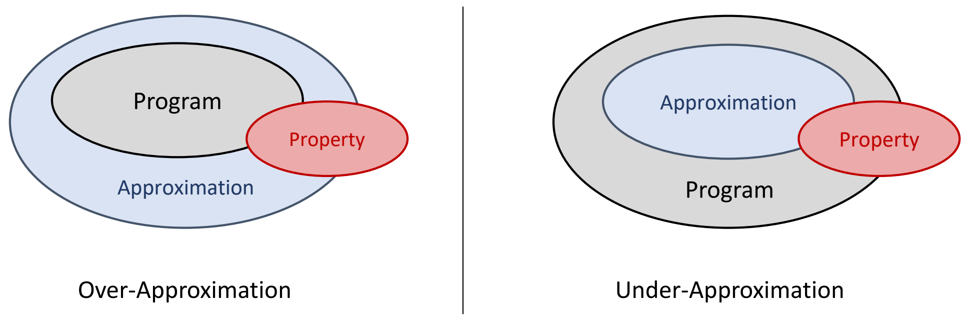


Figure 4.2: Approximation schema.

words, **sound analyzers have *no false negatives***. Instead, a static analyzer is *complete* with respect to a program and a property of interest, when it considers the subset of all program executions where it is able to prove the presence of the property. Hence, if at least one alarm is issued, the property is guaranteed to be detected in at least one execution. That is, **complete analyzers have *no false positives***.

In terms of abstract interpretation, over-approximations can guarantee to keep the analyses *sound* while under-approximations can be used to achieve *complete* analyses (Figure 4.2).

In this thesis, **we are interested in pursuing a *sound* approach and the implementation of the analyzers proposed in this thesis are all sound**. According to B. Meyer [133], it is generally better to use *sound* yet *incomplete* techinques, since false negatives can lead to critical bugs whose mitigation might be impracticable in some contexts (e.g. blockchain).

4.3.2 Soundness, Abstractions, and Precision

Theoretically, developing a sound tool is straightforward. All that is required is for the tool to trigger an alarm for each program point. In this way it always guarantees to verify the property of interest for the analysis, avoiding false negatives. However, in practice, users will perceive the tool as useless because they still have to check all the code to figure out which alarm is a true positive or a false positive. For this reason, a sound tool should reduce the number of false positives as much as possible. Regarding the tools based on abstract interpretation, their analyses can be more or less precise depending on the abstractions. Typically, the higher the level of abstraction, the more over-approximation will be produced which will lead to more false positives. However, abstractions trade precision for computational complexity. Hence, it is necessary to find a trade-off between performance and the number of false positives to develop a valuable tool for the users.

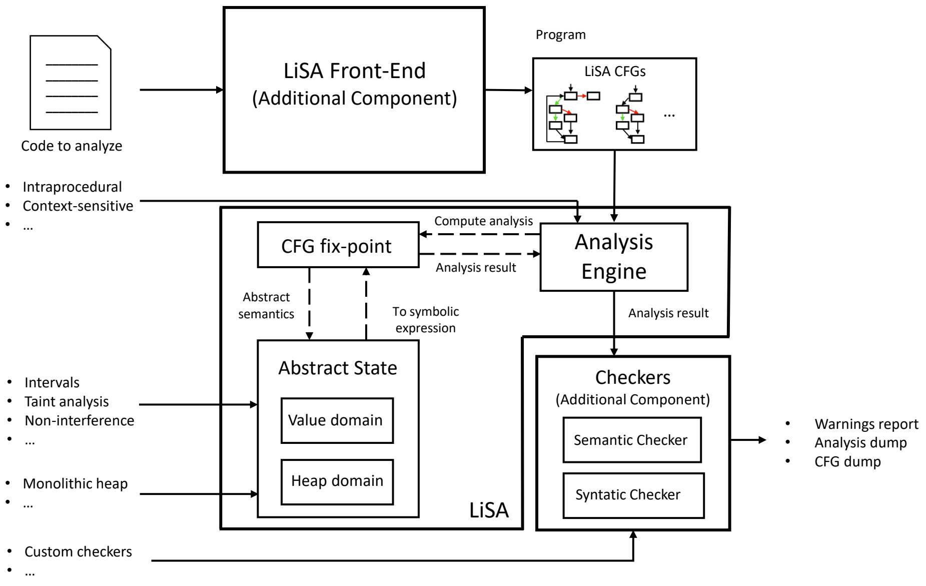


Figure 4.3: LiSA overall architecture.

4.4 How to Build a Static Analyzer From Scratch

Static analysis based on formal methods requires a non-trivial theoretical background and development skills. Before being able to design and implement a new analysis, it is often necessary to have an infrastructure capable of providing the basic features (parser, control flow graphs (CFG) representation [5], fixpoint algorithms, etc.). Hence, the development of even a toy static analyzer from scratch requires a lot of effort.

In this thesis, the implementations of analyzers proposed have relied on LiSA (Library for Static Analysis) [72]. LiSA is an open-source platform developed in Java with the purpose to reduce the technological gap and favor the creation and implementation of static analyzers based on the abstract interpretation theory. In this section, we recall the main concepts and the architecture of LiSA. The information provided is strongly referenced by the official Lisa's documentation [184] and by the teaching experience report of Ferrara et al. [72]. Instead, we discuss the challenges faced during the implementations of two analyzers based on LiSA for the analysis of programs written in Go and Michelson language, respectively in Chapter 5 and in Chapter 6.

The main components provided by LiSA are the following: (i) an internal and extensible CFG representation, (ii) an algorithm for the fixpoint computation on CFG representations, (iii) a common analysis framework for the development of new abstract domains, and (iv) interfaces for the development of common analyses, such as non-relational, data flow analysis [178], type analysis [47], etc.

LiSA is agnostic from the target languages to analyze, thus each component is designed to be as versatile and flexible as possible. Figure 4.3 describes the general architecture of LiSA. As input, LiSA expects a program model which contains a set of LiSA CFGs and the analysis settings. They are generated by a *front-end* which usually generates them from source code. On each CFG, LiSA applies the typical worklist fixpoint algorithm on CFG nodes. At the end of analysis computation, LiSA produces entry and exit abstract states for each node of the CFGs, which can be exploited for instance to generate warnings using *checker*s or to dump analysis information.

Given the general nature of LiSA, the *front-end*s are additional components to manage the translation from a program P written in a programming language L to LiSA CFGs. The first step of a LiSA *front-end* is to extract each syntax element from P and verify if the code is well-written. In this phase, the main components involved are the lexer and the parser, which are built starting from a *grammar* of L. Given P, the lexer spots the *lexemes*, i.e. words in the source code, and returns the corresponding tokens. Then, the parser takes them as input and builds an *abstract syntax tree* (AST) that reflects the grammar. In case of failure, the analyzer aborts the execution with a syntax error [158, Chapter 5]. The AST is a structure that is used by several analyzers. However, it has limited expressiveness. For instance, it cannot represent program details such as paths or intraprocedural views of subroutines. Anyway, the AST is also a good starting point to build a CFG and add the missing expressiveness. At this point, a CFG builder can visit the syntax trees and converts the syntax elements into a CFG representation. Each node corresponds to a statement implementation, which expresses its custom semantic, i.e. the semantic relating to L, through *symbolic expressions* [72] in order to be understandable after by LiSA. The *symbolic expressions* can be considered as an internal language of LiSA to generalize the semantics of a node. While, the connections between one statement and another are indicated through edges. Ideally, the LiSA CFGs express the *syntax* of the program of interest, while the LiSA symbolic expressions express the semantics of CFGs, specifying the meaning of statements for each CFG node. At the end of CFGs computation, an instance of the program will be created.

LiSA provides two interfaces for the implementation of syntactic and semantic checkers, respectively. Intuitively, a syntactic checker performs checks only purely based on the syntax of a program (e.g., a function/method with a certain name and parameters is declared in the program). Instead, for semantic is able to exploit both the syntactic structure of the program and the semantic information produced with the fixpoint iteration.

4.5 Conclusions

Verification is a fundamental phase of software development. In blockchain software, the code is hardly patchable and bugs can become potentially immutable and exploitable by any attacker. Through the use of formal methods, it is possible to prove the absence of malicious properties in the code, while it is possible to detect these properties without executing the code with the static analysis, thus in the early stages of development and reduce the costs and timing of bug fixing during development. However, the implementation of tools based on formal methods is not trivial and requires technical skill and often considerable effort. For this reason, it is advisable to use extensible frameworks or tools, which facilitate their development, such as LiSA.

In the next chapters, we propose tools and analyses based on static approaches and the abstract interpretation theory. In particular, chapters 5 and 6 present two LiSA-based analyzers for Go and Michelson languages, respectively. While chapters 7, 8, 9 propose analyses implemented in LiSA-based analyzers able to scale on real-world software.

Chapter 5

A PARAMETRIC STATIC ANALYZER FOR GO

This chapter presents GoLiSA, an open-source tool developed in Java for the code analysis of programs written in Go. Its primary goal is to reduce the technological gap and facilitate the creation and implementation of highly customizable static analyses for Go, pursuing a *sound* approach [158, Chapt. 1.2]. Some contents of this chapter have been published in [148]. The implementation of GoLiSA is available at [8]. To the best of our knowledge, this is the first analyzer based on LiSA applied to the industrial context and also the first analyzer for Go that supports analyzing different blockchain frameworks.

5.1 The Go Language

Go [63] is a general-purpose programming language, conceptualized at Google, whose aim is to speed up software development in a simple, reliable, and efficient way. It is inspired by various programming languages such as C and Java, but with some specific differences related to memory safety, garbage collection, structural typing, and concurrency. As reported in the official survey [28], Go is particularly suited for developing software like API/RPC services, runnable/interactive programs (CLIs), data processing, and web services. This success has generated many Go frameworks for heterogeneous application contexts.

In the blockchain context, Go has been widely adopted for the creation of several platforms and is also used as the language for the development of smart contracts and DApps. Listed below are the most popular blockchain frameworks.

Cosmos [118] is a framework to build ecosystems of independent interconnected blockchains, which support both public and permissioned Proof-of-Stake (PoS) networks. The blockchain nodes are built using *Tendermint Core* [31], recently rebranded as *Ignite* [106], a BFT middleware that separates the application logic from the consensus and networking layers. This allows one to develop blockchain applications written in any programming language that supports remote procedure calls, and replicate them on many machines [32]. However, the application development, although it supports many different languages, is mainly oriented toward Go. Indeed, the recommended development involves the use of *Cosmos SDK*, an open-source framework written in Go that abstracts away the machinery needed to set up a network running through the consensus mechanism. Calls to *Tendermint Core* are managed at high-level while supporting the interconnection with the IBC (Inter-Blockchain Communication) protocol. The programming style of Cosmos SDK follows the object-capability model where the security of subcomponents is imperative, especially those belonging to the core library. *Cosmos SDK* is not a smart contract framework but a real framework for DApps supporting different functionalities through highly customizable modules, that can also manage smart contracts such as *CosmosWasm* [45].

Hyperledger Fabric [103] is a permissioned blockchain framework, designed to be adopted in the industrial context. It is supported by The Linux Foundation and other contributors such as IBM, Cisco, and Intel. In this blockchain, smart contracts and DApps are called *chaincode* and can be implemented using the full set of instructions and features of popular GPLs such as Go, Node.js, and Java. In most cases, chaincode interacts only with the world state database component of the ledger, and not with the transaction log [102]. Go is currently the most popular language on GitHub related to *chaincode*[1], as *chaincodes* written in Go are the most performant [76].

5.2 An Overview of GoLiSA

We developed GoLiSA based on LiSA [72] to support the standard components of abstract interpretation. It is the first LiSA-based project which, in addition to its scholarly purpose, is also involved in the analysis of industrial code and the detection of real-world issues (Section 7.7).

It models the Go language, its memory model, the implementations and representation of Go instructions with their semantics, supports the various Go

[1] Querying the keyword `chaincode` on GitHub (`https://github.com/search?q=chaincode`) results in almost 2000 repositories and half of them are written in Go, as of 04/2022.

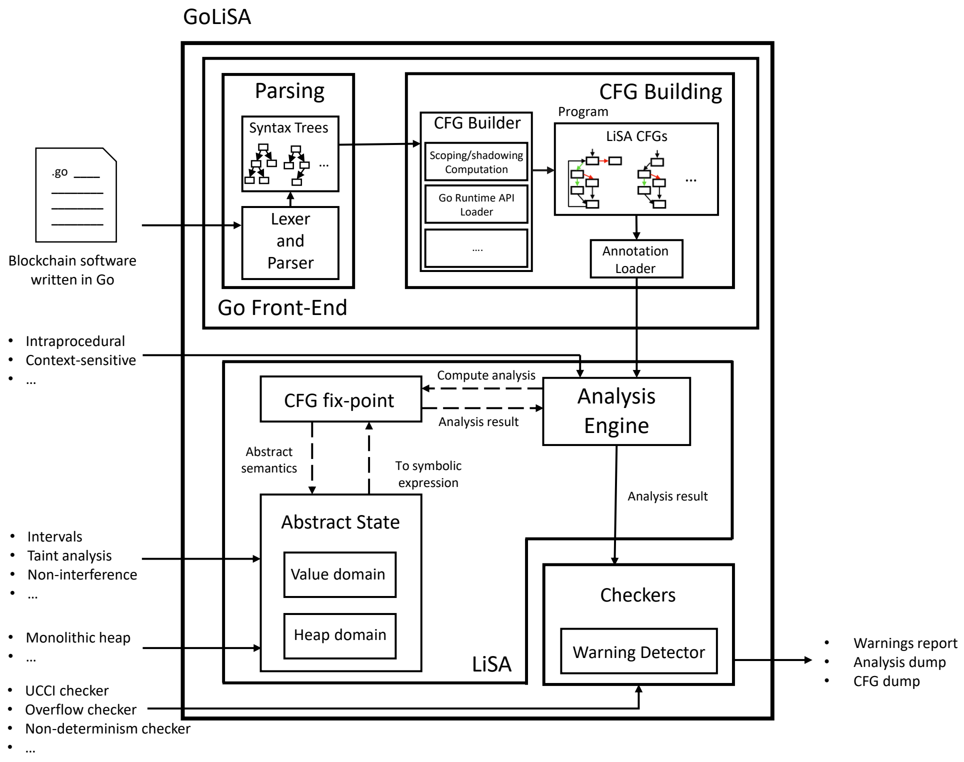


Figure 5.1: GoLiSA overall execution.

frameworks, provides interfaces and classes to build analysis specific to Go, and creates Go program instances enriched with additional information for the analyses. Indeed, as already reported in Section 4.4, LiSA is a generic framework for abstract interpretation and is agnostic from the analyzed language. Then, LiSA does not handle the peculiarities of Go, and for this reason, it needs GoLiSA components to model it.

5.2.1 Parsing and CFG Construction

Figure 5.1 shows the overall high-level architecture of GoLiSA. The execution flow starts with the *Go front-end*, which takes a Go program as input. It translates Go applications into a set of LiSA CFGs, later inspected and analyzed by LiSA. First, it extracts each syntax element from the file and verifies if it is syntactically correct. In this phase, the main components involved are a lexer and a parser, developed from the grammar of Go, as provided by ANTLR4 [157]. Given a Go program, if it is syntactically correct, the parser produces an AST of the program of interest. From that, GoLiSA builds a set of CFGs, whose nodes implement the source statements in terms of an analysis-independent intermediate language of *LiSA symbolic expressions*, that LiSA understands.

The generation of the CFGs takes also into account language-specific issues, such as, in the case of Go, the possibility of variable shadowing[2]. A variable is said to be *shadowing* another variable if it *overrides* the variable in a more specific scope. In Go, a variable can be re-defined within each new block and restores its value once closed the block. The operations, including the assignments, will only affect the last visible redefinition of that variable. The value of the other definitions will be restored as soon as the block of the nested redefinition is closed. Figure 5.2 shows some examples of *shadowing* in Go. In particular, the operator`:=` defines a variable declaration with an assignment, while `=` defines a variable assignment only. The language-specific aspects are of fundamental importance for Go code analyses. In the absence of this variable information, a conservative analysis, to be *sound*, over-approximates the propagation results, leading to the generation of false positive warnings. GoLiSA during the building of the program collects all this information and passes it for each statement to LiSA

Going on with the execution flow, at this point, CFGs with additional information are passed to an interprocedural analysis that uses a call graph to resolve calls and compute their results. LiSA provides a set of interprocedural fixpoint algorithms over the CFGs, that interprets the abstract semantics of the symbolic expressions by using an abstract state, including a heap domain (to model the dynamic memory of the program) and a value domain (to track the abstract values of variables and dynamic abstract locations). The fixpoint computes a sound over-approximation of the CFG node's semantics of the symbolic expressions, according to the specific logic of the node's semantics. Once a fixpoint is computed, it can be used by a set of checkers to verify the properties of interest and possibly trigger warnings.

5.3 Related Work

The range of analysis tools for Go is mainly oriented toward generic issues and code smells detection [94,141,185]. In most cases, these are *linters*[3] or enterprise tools, therefore subject to limited analysis potential or extension difficulty for custom analyses, respectively. Moreover, although the generic nature of these tools can be useful in most implementations written in Go, they can fail and generate critical problems if applied to specific contexts. Indeed, even if to a much more limited extent, it is possible to find specialized tools in particular

[2] https://stackoverflow.com/questions/38697987/golang-function-contains-anonymous-scope
[3] https://en.wikipedia.org/wiki/Lint_(software)

```
v := 0      // v <- 0
{
  v := 1    // v <- 1
  foo(v)    // v <- 1
}
foo(v)      // v <- 0
```

(a)

```
v := 0      // v <- 0
{
  v = 1     // v <- 1
  foo(v)    // v <- 1
}
foo(v)      // v <- 1
```

(b)

```
v := 0        // v <- 0
{
  v := 1      // v <- 1
  {
    v = 2     // v <- 2
    foo(v)    // v <- 2
  }
  foo(v)      // v <- 2
}
foo(v)        // v <- 0
```

(c)

Figure 5.2: Example of shadowing in Go. (a) A re-declaration of `v` in the block. (b) An assignment to `v` in the block. (c) An example with multiple blocks.

contexts. For instance, Gobra [203] is a modular, deductive program verifier based on separation logic that proves memory safety, crash safety, data-race freedom, and user-provided specifications. Its implementation translates an annotated Go program into the Viper intermediate verification language and uses an existing SMT-based verification backend to compute and discharge proof obligations. While, in [34], Chabbi et al. propose strategies based on dynamic analysis for detecting data races on industrial code.

About blockchain, in recent years, new verification tools have been developed that allow the modeling of smart contracts and their properties [193]. Unfortunately, concerning Go, aside from GoLiSA, only a few tools are specialized in the blockchain field. Moreover, they are tailored exclusively to specific smart contract frameworks and cannot analyze other blockchain frameworks supporting Go. Chaincode Analyzer [119] is a static analyzer proposed by Fujitsu in the Hyperledger Labs. Revive^CC [180] is an extension of the revive analyzer [134], specific for chaincode. Both extract the abstract syntax tree from the source code and perform their checks on it. The checks performed by these tools are almost exclusively syntactic with the addition of minor reasonings about types. According to [126], also their tool is similar to Chaincode Analyzer. The main differences are that Chaincode Analyzer covers a minor set of issues, while, their tool is more accurate, i.e. it syntactically checks the signature functions instead of just triggering a warning on black-listed imports. Instead, HFContractFuzzer [62] applies go-fuzz [87] to discover a vulnerability on Go smart contracts for HF. ZEUS [112] is a framework for the verification of smart contracts written for Ethereum and Hyperledger Fabric. It converts source code into LLVM bytecode and exploits abstract interpretation and symbolic models to detect vulnerabilities. However, Kalra et al. [112] does not explain in detail the analysis to make a comparison and the tool is not available. Moreover, there are no works nor benchmarks about ZEUS for Go, although it would seem to support it.

5.4 Conclusions

Go is a versatile programming language applied to different contexts. However, this exposes it to multiple vulnerabilities when applied to blockchain scenarios, being a GPL and then not conceived by design for the blockchain context. In this chapter, we have presented a LiSA-based tool capable of verifying software implemented using popular blockchain frameworks in Go. In particular, it allows performing non-trivial semantic analyses that many of the state-of-practice tools for blockchain do not have. About the analyses treated in the thesis, Chapters 7, 8 and 9 describe the detection of issues respectively related to *non-determinism*, *untrusted cross-contract invocation* and *numerical overflow/underflow* using GoLiSA.

Chapter 6

A PARAMETRIC STATIC ANALYZER FOR TEZOS SMART CONTRACTS

This chapter introduces MichelsonLiSA, a prototype of a static analyzer based on abstract interpretation for the verification of Tezos smart contracts written in the Michelson language, which is the low-level language of the Tezos blockchain. The implementation of MichelsonLiSA is available at [11]. To the best of our knowledge, this is the first analyzer based on LiSA with an intermediate representation based on the static single-assignment form and a symbolic stack. Some contents of this chapter are under publication at Workshop on Blockchain theoRy and ApplicatIoNs (BRAIN'23).

6.1 The Michelson Language

The Tezos community has made several tools for implementing smart contracts at high-level, supporting popular programming languages such as Python, OCaml, and Javascript, among others [124,182]. However, despite this large language support, the only native smart contract language of the Tezos blockchain is the low-level Michelson language: high-level code is translated into Michelson and the latter is deployed in the blockchain. Michelson is a domain-specific language, statically typed, and whose only variables are the stack elements: there are no fields and no global variables. It supports low-level instructions only, enough to implement Turing-complete smart contracts, whose structure is specified by three components:

```
parameter (pair int int) ; # Parameter declaration: two integers
storage int ; # Storage declaration
code { # Code declaration

  CAR ; # Push the input parameter to the stack and discard the
      current storage value
  UNPAIR ; # Pop the input pair from the stack, split it into two
      integers and push them on the stack instead
  ADD ; # Pop the two integers and push their sum instead

  NIL operation ; # Push an empty list of operations required to end
       the contract

  PAIR ; # Build the final stack: a pair consisting of a list of
      operations and the value to keep in storage (in this case, the
       result of addition)
}
```

Figure 6.1: A Michelson smart contract that adds two input integers and stores the sum in the blockchain.

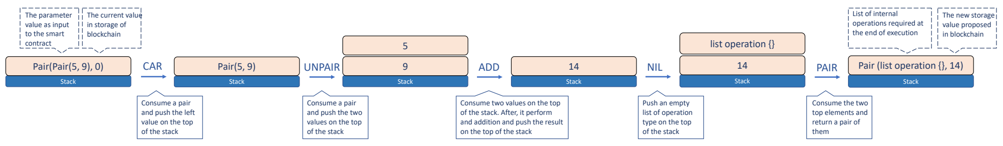

The parameter value as input to the smart contract
The current value in storage of blockchain
Pair(Pair(5, 9), 0)
Stack
CAR
Consume a pair and push the left value on the top of the stack
Pair(5, 9)
Stack
UNPAIR
Consume a pair and push the two values on the top of the stack
5
9
Stack
ADD
Consume two values on the top of the stack. After, it perform and addition and push the result on the top of the stack
14
Stack
NIL
Push an empty list of operation type on the top of the stack
list operation {}
14
Stack
PAIR
Consume the two top elements and return a pair of them
List of internal operations required at the end of execution
The new storage value proposed in blockchain
Pair (list operation {}, 14)
Stack

Figure 6.2: An example of the execution of the smart contract in Figure 6.1.

Parameter declaration: explicitly typed input.
Storage declaration: explicitly typed blockchain store locations.
Code declaration: sequence of instructions, similar to bytecode.

Technically, the input is a single value that specifies the required code execution. However, the use of aggregate types, such as `pair` and `or`, allows one to provide more than a single input value to a contract, as Figure 6.1 shows.

The execution of a Michelson contract is stack-based: instructions pop and/or push stack elements. In the Tezos blockchain, a smart contract execution request (*invocation*) specifies the address of the smart contract in blockchain and its input[1]. The execution starts from a stack whose only element is the pair of the input and the current value of the storage of the contract. Figure 6.2 shows an example: for the execution of the contract in Figure 6.1 with input `Pair(5,9)` and assuming that the current value of the storage of the contract is the integer `0`, the initial stack contains a singleton value `Pair(Pair(5,9),0)`. Note that the user provides the input, while the blockchain protocol retrieves the storage value from the blockchain state. The first instruction in this example, `CAR`, splits the pair and projects it on its first component `Pair(5,9)` (the input),

[1] https://tezos.gitlab.io/michelson-reference/#execution

Instruction	***Description***
`ADDRESS`	pop a contract value and push the address of that contract
`AMOUNT`	push the amount of the current transaction
`BALANCE`	push the current amount of mutez of the executing contract
`CHAIN_ID`	push the chain identifier
`CONTRACT`	replace the top of the stack after cast to a contract type
`CREATE_CONTRACT`	push a contract creation operation
`IMPLICIT_ACCOUNT`	push the address of a new implicit account
`LEVEL`	push the current block level
`NOW`	push block timestamp
`SELF`	push the current contract
`SELF_ADDRESS`	push the address of the current contract
`SENDER`	push the contract that started the current internal transaction
`SET_DELEGATE`	push a delegation operation
`SOURCE`	push the contract that initiated the current transaction
`TOTAL_VOTING_POWER`	push the total voting power of all contracts
`TRANSFER_TOKENS`	push a transaction operation
`VOTING_POWER`	push the voting power of a contract

Table 6.1: Domain-specific operations of Michelson.

which pushes on the stack instead: the current storage value is discarded. The subsequent `UNPAIR` instruction decomposes `Pair(5,9)` into its two components `5` and `9` which pushes on the stack instead. The `ADD` instruction computes their sum (`14`), that pushes on the stack instead. At this point, the core execution of the smart contract is done. In Tezos, to successfully terminate the execution of the contract, it is always necessary to return a pair containing a list of operations (internal values pushed by instructions such as `TRANSFER_TOKENS`, `SET_DELEGATE` and `CREATE_CONTRACT`) and a value typed as reported in the storage declaration. In the example, the `NIL` instruction pushes an empty list of operations (operations are optional, but the list is always required) to perform at the end of the execution and the final `PAIR` instruction boxes the list and the result into a pair: the latter is the result of the execution. The blockchain protocol will take that second component (`14`) and store it in the storage of the contract, for future use.

Michelson consists of around 100 instructions[2]: for stack manipulation (`PUSH`, `DROP`, `SWAP`, ...); for creation and management of high-level data structures (`MAP`, `UPDATE`, `SIZE`, ...); for arithmetic (`SUM`, `SUB`, `AND`, ...); for control flow (`IF`, `LOOP`, ...) and for blockchain-specific operations (see Tab.6.1).

[2] https://tezos.gitlab.io/michelson-reference

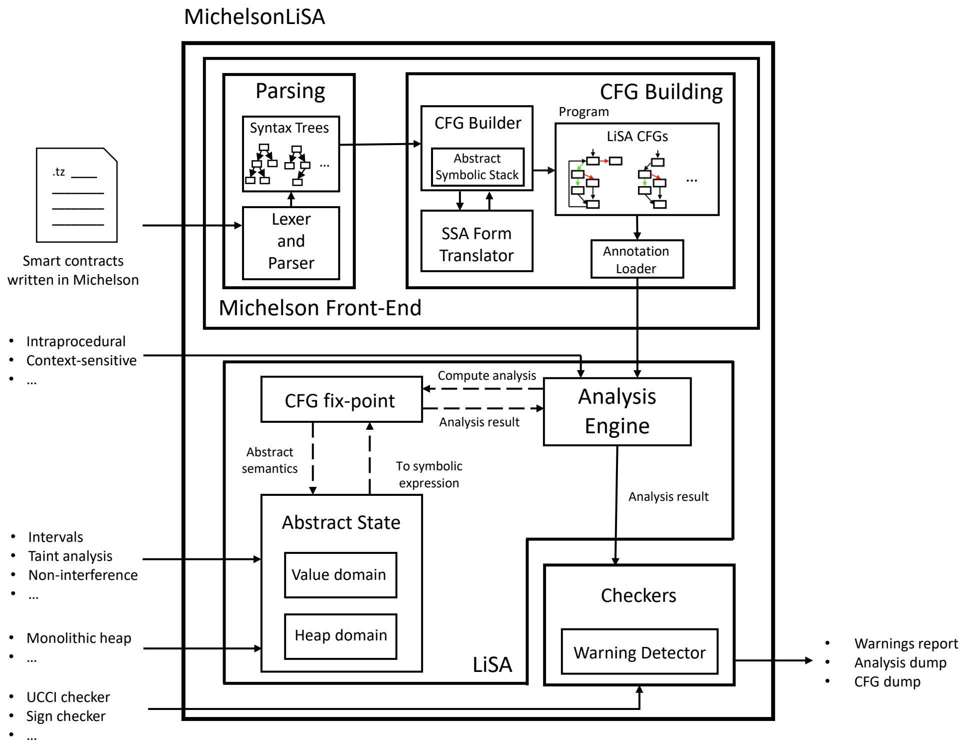


Figure 6.3: MichelsonLiSA architecture.

6.2 An Overview of MichelsonLiSA

MichelsonLiSA is a static analyzer based on abstract interpretation [47, 48], for Tezos smart contracts written in the Michelson language. It relies on LiSA [72], a library that provides a complete infrastructure for the development of static analyzers. In particular, LiSA implements several standard components of abstract interpretation-based analyzers [158], such as an extensible CFG representation, a common analysis framework for the development of new static analyses, several built-in standard static analyses (such as type analysis [46], information flow analyses [47, Chapt. 47], etc.) and fixpoint algorithms on LiSA CFGs. To analyze smart contracts with LiSA, MichelsonLiSA needs to design and implement additional components, that manage the translation from the Michelson source code to the intermediate representation (IR) supported by LiSA. The next section presents these components and the challenges faced to support the Michelson language. Figure 6.3 shows an overview of MichelsonLiSA.

6.2.1 Parsing and CFG Construction

The first step of the Michelson *front-end*, in order to verify and analyze a smart contract, is to define the language syntax, i.e, a specification of how the code must be written. The syntax of the Michelson language is specified by a grammar[3]. However, that grammar lacks some syntactic sugar (such as annotations, use of brackets, smart contract structure, macros, etc.) widely used in real-world Tezos contracts. Hence, we enriched that grammar and implemented it[4] in the ANTLR v4 format. ANTLR [157] is a popular tool that, starting from a grammar, builds a *lexer* and a *parser* for the grammar. Then, the CFG building phase starts after the parsing of the Michelson source code into AST. The CFG builder translates the code into an IR based on Static Single Assignment (SSA) form [52, 170] and builds the LiSA CFGs.

MichelsonLiSA analyses the SSA code once it is put inside a CFG, that expresses the control structure of the code. Each node contains *symbolic expressions*, that is, expressions in the internal language of LiSA, used to implement the semantics of the statements, in a language-independent way. Namely, such symbolic expressions are low-level instructions that can be used to compile many source languages. The abstract semantics of a CFG is defined as a fixpoint, that will be reached in a finite number of iterations if the *abstract domain* has finite height [70]. The abstract states computed during that fixpoint computation are a sound over-approximation of the semantics of the symbolic expressions, that a checker can use to issue warnings. For instance, the `v21 = SUB(v10, v11)` SSA instruction in Figure 6.5b is translated into a symbolic binary expression that performs subtraction, whose semantics is implemented in Figure 6.4, parametrically *wrt.* the abstract state.

6.2.2 Intermediate Representation in SSA Form

Michelson is a low-level, stack-based language. According to Demange et al. [55], the use of a stack makes it difficult to apply standard static analysis techniques. Therefore, an IR is necessary to provide an efficient model for the analysis, in terms of transformation time and produced code. LiSA is designed to handle a generic program language but is currently variable-oriented. For this reason, we translate the stack-based representation into a variable-based IR, by using the SSA form. The translation maps each Michelson instruction[5] into a list of MichelsonLiSA instructions, by using new fresh variables. It tracks, abstractly,

[3] https://tezos.gitlab.io/active/michelson.html#full-grammar

[4] https://github.com/lisa-analyzer/michelson-lisa/tree/master/michelson-lisa/src/main/antlr

[5] https://tezos.gitlab.io/active/michelson.html#core-instructions

```java
public class MichelsonSub extends BinaryExpression
implements StackConsumer, StackProducer {

  public MichelsonSub(CFG cfg, String sourceFile, int line, int col,
  Expression left, Expression right) {

    super(cfg, new SourceCodeLocation(sourceFile, line, col), "SUB",
    computeType(left.getStaticType(),right.getStaticType()), left,
        right);
  }

  @Override
  protected <A extends AbstractState<A,H,V,T>,
  H extends HeapDomain<H>,
  V extends ValueDomain<V>,
  T extends TypeDomain<T>>
  AnalysisState<A,H,V,T> binarySemantics(
  InterproceduralAnalysis<A,H,V,T> interprocedural,
  AnalysisState<A,H,V,T> state,
  SymbolicExpression left, SymbolicExpression right,
  StatementStore<A,H,V,T> expressions) throws SemanticException {

    return state.smallStepSemantics(
    new BinaryExpression(getStaticType(), left, right,
    NumericNonOverflowingSub.INSTANCE, getLocation()),
    this);
  }
}
```

Figure 6.4: Generic implementation of the `SUB` symbolic expression of LiSA. `MichelsonSub` is a binary expression with components `left` and `right`. It is both a stack consumer and a stack producer. The template method `binarySemantics` implements its semantics, parametrically *wrt.* the abstract domain.

the values on the stack through a symbolic stack of such variables[6]. In this way, it tracks the stack elements by using symbolic names only, and not their exact values. Figure 6.5 shows the translation of a Michelson contract into SSA and the corresponding symbolic stack. Given a parameter, the contract performs an addition if the first component of the input pair is larger than the second one; otherwise, it performs a subtraction. At the end of the contract, the result is encapsulated in a pair, consisting of an empty list of internal operations and the new value for the storage data.

Instructions that push values on the stack are translated into variable assignments, with fresh variables standing for stack elements, each assigned exactly once. Instructions that pop from the stack are translated into MichelsonLiSA instructions taking those variables as parameters. Some instructions can be both

[6] https://github.com/lisa-analyzer/michelson-lisa/tree/master/michelson-lisa/src/main/java/it/unive/michelsonlisa/frontend/visitors/MichelsonStack.java

```
parameter (pair int int);
storage int;
code {
  CAR;
  DUP;
  UNPAIR;
  COMPARE;
  GT;
  IF
  {  # True branch
    UNPAIR;
    ADD;
  }
  { # False branch
    UNPAIR;
    SUB;
  }

  NIL operation;
  PAIR;
}
```

(a) Michelson code

```
v0 = parameter_storage();
v1 = CAR(v0);
v2 = DUP(v1);
v3 = get_left(v2);
v4 = get_right(v2);
v5 = COMPARE(v3, v4);
v6 = GT(v5);
IF(v6)
{ # True branch
  v7 = get_left(v1);
  v8 = get_right(v1);
  v9 = ADD(v7, v8);
}
{ # False branch
  v10 = get_left(v1);
  v11 = get_right(v1);
  v12 = SUB(v10, v11);
}
v13 = phi(v9, v12);

v14 = NIL(operation);
v15 = PAIR(v14, v13);
```

(b) SSA Form representation

Figure 6.5: A Michelson smart contract and its translation into SSA form.

producers and consumers. Figure 6.6 shows an example of translation in SSA for some common instructions. `PUSH <type> <data>` pushes a constant of the declared type: it is translated with a fresh new variable that gets assigned a constant of a declared type. `SUB` consumes its two operands from the stack and pushes their difference instead: it is translated as a function that receives the operands as arguments and yields their difference. `DROP` pops and discards the top of the stack: it is translated with a function with no return value. `PAIR` consumes the two topmost stack elements and packs them into a pair that pushes on the stack instead: it is translated as a function with two arguments that yields the pair. `UNPAIR` pops a pair, splits it, and pushes its two components instead: it is translated with two functions, that select the two components and store them into fresh new variables.

Some Michelson stack-modifying instructions perform relatively complex stack operations. Namely, `SWAP` exchanges the topmost two elements of the stack; `DIG n` shifts the stack element at depth `n` into the top of the stack, while `DUG n` does the converse. All these instructions can be translated into SSA. Figure 6.7 shows an example of translation for `DIG n`.

Michelson includes instructions for conditionals, such as `IF`, and for iteration, such as `LOOP`, both leading to branches and junction points. For junctions, SSA reconciles distinct values of the same variable, arising along different paths, through ϕ-functions [52]. The idea is to translate instructions separately along

```
0: PUSH
    int
    23;
1: PUSH
    int
    13;
2: SUB;
3: DROP;
4: PUSH
    int
    23;
5: PUSH
    int
    13;
6: PAIR;
7: UNPAIR;
8:
```

(a) Michelson code

```
0: []
1: [v1]
2: [v1,v2]
3: [v3]
4: []
5: [v4]
6: [v4,v5]
7: [v6]
8: [v7, v8
    ]
```

(b) Symbolic stack

```
0: v1 = PUSH(int, 23)
    ;
1: v2 = PUSH(int, 13)
    ;
2: v3 = SUB(v1,v2);
3: DROP(v3);
4: v4 = PUSH(int, 23)
    ;
5: v5 = PUSH(int, 13)
    ;
6: v6 = PAIR(v4,v5);
7: v7 = get_left(v6);
v8 = get_right(v6);
8:
```

(c) SSA form

Figure 6.6: Example of transformation into SSA form.

```
0: PUSH nat 5;
1: PUSH nat 3;
2: PUSH nat 2;
3: DIG 2;
4: DROP;
5:
```

(a) Michelson code

```
0: []
1: [5]
2: [5, 3]
3: [5, 3, 2]
4: [3, 2, 5]
5: [3, 2]
```

(b) Execution stack

```
0: v1 = PUSH(nat, 5);
1: v2 = PUSH(nat, 3);
2: v3 = PUSH(nat, 2);
3: DIG(2);
4: DROP(v1)
5:
```

(c) SSA form

```
0: []
1: [v1]
2: [v1, v2]
3: [v1, v2, v3]
4: [v2, v3, v1]
5: [v2, v3]
```

(d) Symbolic stack

Figure 6.7: Michelson code using a `DIG n` instruction and its SSA form representation.

each path, using disjoint sets of variables, and merge, at the junction point, into a new unique fresh variable, the distinct variables standing, along distinct paths, for the same stack element. Figure 6.8 shows an example.

Michelson has *stack-protecting* instructions, such as `DIP n`, that temporarily freeze the topmost `n` elements of the stack, keeping them unaffected during the execution of a specified group of subsequent instructions. Figure 6.9a shows a snippet of code that uses `DIP 2` at Line 3. There, the stack holds `[5, 3, 4]` (from bottom to top), as reported in Figure 6.9b. `DIP 2` freezes its topmost

```
0: IF
1: { #
     True
     branch
  2:
       PUSH
        int
        -1;
  3: }
4: { #
     False
     branch
  5:
       PUSH
        int
        7;
  6: }
7:
```

(a) Michelson code

```
0: stack   [v0]
1: stack   []
2: stack1  []
3: stack1  [v1]
4: stack1  [v1]
5: stack2  [v2]
6: stack1  [v1],
    stack2 [v2]
7: stack   [v3]
```

(b) Symbolic stack

```
0: IF(v0)
1: { # True
    branch
  2: v1 = PUSH(
      int, -1);
  3: }
4: { # False
    branch
  5:   v2 = PUSH
      (int, 7);
  6: } v3 = phi(
      v1, v2) #
      Junction
      point
7:
```

(c) SSA form

Figure 6.8: Example of transformation of a conditional into SSA form, with a junction point. The ϕ-function is written as `phi`.

two elements (`3` and `4`) during the execution of the instructions specified inside the curly braces. Namely, `PUSH nat 1` pushes `1` immediately below the frozen elements, instead of on top of the stack, leading to the stack `[5, 1, 3, 4]`. Similarly, `ADD` pops the two topmost, unprotected stack elements `5` and `1` and pushes their addition immediately below the frozen elements. This behavior is reflected in the SSA translation (Figure 6.9c): `PUSH nat 1` becomes `v1 = PUSH(nat, 5)`, with `v1` pushed on top of the symbolic stack (Figure 6.9d). Similarly for the two subsequent `PUSH` instructions. At Line 3, the symbolic stack will be `[v1, v2, v3]` and `v2` and `v3` will become protected. Consequently, at Line 4, the `PUSH` instruction is translated into `v4 = PUSH(nat, 1)`, with `v4` placed below the protected area of the symbolic stack, which becomes now `[v1, v4, v2, v3]`. The subsequent `ADD` instruction will operate on the unprotected elements `v1` and `v4` and gets translated into `v5 = ADD(v1, v4)`, with `v5` pushed immediately below the protected values.

Michelson smart contracts interact with the context of Tezos where they execute. For instance, at the beginning of their execution, the stack holds a pair of the input value and the current storage value. This must be made explicit in the SSA translation, as in Figure 6.5, with `v0 = parameter_storage()`. Instrumentation is needed for data structures as well. Namely, Michelson supports high-level data structures (sets, lists, maps, optionals) and has specific instructions to operate on them, such as `ITER`, `LOOP_LEFT` and `IF_CONST`. These typically push additional elements on the stack. For instance, `ITER` consumes a collection

```
0: PUSH nat 5;
1: PUSH nat 3;
2: PUSH nat 4;
3: DIP 2 {
  4:  PUSH nat 1;
  5:  ADD;
  6: }
7:
```

(a) Michelson code

```
0: []
1: [5]
2: [5, 3]
3: [5, 3, 4]
4: [5, (3, 4)]
5: [5, 1, (3, 4)]
6: [6, (3, 4)]
7: [6, 3, 4]
```

(b) Execution stack

```
0: v1 = PUSH(nat, 5);
1: v2 = PUSH(nat, 3);
2: v3 = PUSH(nat, 4);
3: DIP(2) {
  4:   v4 = PUSH(nat,
      1);
  5:   v5 = ADD(v1, v4)
      ;
  6: }
7:
```

(c) SSA form

```
0: []
1: [v1]
2: [v1, v2]
3: [v1, v2, v3]
4: [v1, (v2, v3)]
5: [v1, v4, (v2, v3)]
6: [v5, (v2, v3)]
7: [v5, v2, v3]
```

(d) Symbolic stack

Figure 6.9: Michelson code that uses a `DIP n` instruction and its corresponding stack execution. Round brackets highlight the protected area of the stack.

from the stack and applies a set of instructions to each of its elements. These gets simulated, in SSA, by using assignments to additional variables.

6.3 Related Work

The formal verification of smart contracts is a crucial issue in the blockchain context. Several tools are developed in this regard [193]. However, only a few of them are involved in verifying the Michelson language. Bernardo et al. propose Mi-Cho-Coq [25, 26], a Coq framework for verifying the functional correctness of Michelson contracts. They also introduce an intermediate language called Albert [27], which provides a high-level stack abstraction based on linearly typed records that can be exploited by Mi-Cho-Coq. In [166], Reis et al. describe an intermediate representation called Tezla that abstracts the stack usage through the usage of a store and can be combined with SoftCheck analyzer to perform data-flow analyses. Arrojado et al. propose a proofer in Why3 for the deductive verification of the Michelson contract called WhylSon [95]. Instead, Bau et al. describe a static analyzer for Michelson [20] based on abstract interpretation and implemented within MOPSA [137].

Our transform function for the intermediate representation is similar to BC2BIR [55], which transforms Java bytecode into variable assignaments, including exception flows, and is based on a symbolic stack execution. However, BC2BIR is variable-based without being in SSA form. Indeed, it does not guarantee SSA of variables in linear code, it does not consider ϕ-functions and variables are assigned several times at predecessors of junction points.

6.4 Conclusions

Michelson is the low-level language for Tezos blockchain. Its peculiarities of DSL and its memory management based solely on the stack, make it a challenging language for traditional verification tools. Our solution proposes a tool based on LiSA, which computes the program model using the SSA form and an abstract symbolic stack for the intermediate representation. As far as we know, this is the first proposed SSA-based intermediate representation for LiSA and may be used in the future as a starting point for supporting other languages. Several analyses are introduced in the next chapters. In particular, Chapter 8 deals with the detection of *untrusted cross-contract invocation*s by using MichelsonLiSA.

Chapter 7

ENSURE DETERMINISM IN BLOCKCHAIN SOFTWARE

A mandatory feature for blockchain software is determinism. When determinism is not met it can lead to serious implications in the blockchain network while compromising the software development, release, and patching processes.

Typically, DSLs for blockchain software ensure determinism by simply avoiding non-deterministic language features, as happens for Michelson language.

However, this is not the case with GPLs, such as Go. They are adopted in the industrial context for developing blockchain solutions (Section 3.2), but they allow one to implement software containing non-deterministic behaviors, being these programming languages not originally designed for blockchain.

By the way, our perspective is that not all non-deterministic behaviors are critical for blockchain. Indeed, only those that affect the state or the response of the blockchain can cause problems, i.e. observable and shared by the network, as other uses are only visible by the single peer that executes the application and not by others.

This chapter describes the problem in detail, proposing a flow-based approach to detect non-deterministic vulnerabilities which could compromise the blockchain. This led to the implementation of an analysis for GoLiSA and the experimental results show that GoLiSA is able to detect all vulnerabilities related to non-determinism on a significant set of real-world programs written in Go, with better results than other open-source analyzers. To the best of our knowledge, this is the first application of information flow analysis for the detection of issues related to non-determinism in blockchain software. Some contents of this chapter are published in [148] and also they are under publication at European Conference on Object-Oriented Programming (ECOOP'23).

7.1 Non-Determinism in Blockchain

The software deployed into a blockchain is distributed and decentralized in different peers of the blockchain network. The consensus mechanism is the component that checks the results of blockchain software, and the state of peers involved in the consensus and allows or not to update the global state of the blockchain. If a certain threshold of commonly committed states among peers is reached, the common state is used to update the global state, i.e. consensus is reached, otherwise it is discarded, i.e. consensus is not reached, avoiding updating the state differently among the nodes of the blockchain network.

In this scenario, the non-deterministic response of the blockchain network may lead to a discard of updates because they could produce different states among the blockchain network participants. Hence, deterministic execution is required for software that runs in a blockchain, since it guarantees that, starting from a common state, the same result is reached with the same response in any distinct blockchain node, avoiding inconsistency among peers and consensus failures.

Moreover, these issues related to non-determinism are subtle and do not necessarily implicate an attack from an unknown enemy. In fact, the developer is often the worse enemy of himself. Indeed, this can also happen in a network where all the participants are trusted and it is unrelated to the majority attack (51% attack) [206], in which most of the network is considered compromised by malicious peers. Indeed, according to [64], non-determinism is *"most often the result of a mistake on the part of the programmer"*. Non-determinism can have a critical impact on the peers and the network:

- *Deny of Service.* The software of a smart contract that contains this issue could cause the failure of all transaction requests, making the contract inaccessible. Furthermore, in the worst-case scenario, if the problem affects the entire application layer, it could totally deny transaction requests across the entire network, making the blockchain unable to perform write or execute operations. In addition, as already reported in Section 4, depending on the type of blockchain, patching could take a long time and also lead to a blockchain fork.
- *Economic loss.* The blockchain uses the economic factor as a deterrent, to discourage inappropriate behavior. It is managed by a consensus algorithm, which is not generally able to recognize that the problem is due to an unwanted non-determinism problem. Therefore the penalties will still be applied to the components of the blockchain network. The transaction fee is a mechanism normally used both to reward validators to mitigate DoS attacks and to discourage the overflow of transaction requests. However, users lose transaction fees for failed transactions due to non-determinism issues. While,

Level	***Category***	***Package***	***Statements/Methods***
Framework and *Language*	Map iteration	-	`range` on map
	Parallelization and concurrency	-	`go` (go routine), `<-` (channel)
	Random value generation APIs	`math/rand`, `crypto/rand`	*
Environment	File system APIs	`io`, `embed`, `archive`, `compress`	*
	OS APIs	`os`, `syscall`, `internal`, `time`	*
	Database APIs	`database`	*
	Internet APIs	`net`	*

Table 7.1: Overview of non-deterministic behaviors in Go. Most of the APIs contained in these packages lead to non-deterministic behaviors, but some can be considered safe for determinism.

in a PoS blockchain, the validator nodes can be indicated by the consensus algorithm as being cheaters due to the different (non-deterministic) results. Therefore, they could lose partially or fully the amount of stake.

– *Exclusion from validation.* In addition to the economic and stake loss, and in the worst case even the validator nomination can be lost and the peer is marked as malicious or banned from the blockchain network. Consensus algorithms are not able to understand the difference between intentionally cheating behavior and unintentional behavior caused by issues of non-determinism. Hence, they can apply even the most severe penalties.

7.1.1 Sources of Non-Determinism

The sources of non-determinism can be categorized into two main families. The first one is related to the blockchain framework and the programming language adopted to develop the software. This family comprises a set of constructors or APIs allowed by the framework that may break the consensus during the execution of smart contracts or DApps. Examples are non-deterministic iterations, parallelization or concurrency statements, and random value generators[1].

The second family is trickier since it involves statements related to the underlying environment, such as file systems, operating systems, databases, and Internet connections. For instance:

[1] Random numbers are allowed in smart contracts [35], but they are strictly related to the semantics of the methods that generate them. Usually, standard methods of general-purpose languages to generate random values do not satisfy such semantics, i.e. they generate random values that compromise the consensus algorithm.

– *File systems*: if a program needs to handle a certain file on a node, the same file may not exist in all nodes involved in the consensus protocol — at each validation round the nodes could be different and the file might have been deleted, edited, moved, or any kind of operation might fail due to insufficient space on the disk;
– *Operating systems*: the blockchain protocol may operate on different operating systems (OS) and some language APIs could return different results if called from a different OS, for instance when low-level calls occur during contract execution — common APIs such as time and date methods could return different values if nodes are not synchronized;
– *Databases*: database records could be deleted or edited, and the database state could be different in the nodes or the access could be denied during the consensus phase;
– *Internet connections*: some Internet addresses could be unreachable from some nodes of the network, or the messages exchanged during the connection could be different.

7.1.2 Sinks of Non-Determinism

The sinks of non-determinism, i.e. the statements that are sensitive to non-deterministic behaviors, can be categorized into two families. The first one is related to the blockchain state. This family includes a set of constructors or APIs with the ability to modify the common state of the blockchain and therefore involved in the consensus mechanism. The second one is related to the response of blockchain networks. The execution of code within the blockchain does not necessarily change the state of the blockchain (e.g., functions that simply read a value). However, the execution may lead to non-deterministic transaction responses, compromising the consensus of the network.

```go
func (g Grant) ValidateBasic() error {
  if g.Expiration.Unix() < time.Now().Unix() {
    return sdkerrors.Wrap(ErrInvalidExpirationTime, "Time can't be in
        the past")
  }
  // [...]
}
```

Figure 7.1: Cosmos SDK code affected by CVE-2021-41135.

```go
func transfer(from, to Address,  value int64, stub *shim.
    ChaincodeStub) {
  start := time.Now()
  //... transfer operations that takes some milliseconds ...
  elapsed := time.Now().Sub(start)
  log.Println("Time elapsed for the transfer operations: ", elapsed)
}
```

Figure 7.2: Example of safe use of time APIs in blockchain software.

```go
func transfer(from, to Address, value int64, stub *shim.
    ChaincodeStub) {
  t := time.Now()
  //... transfer operations ...
  err := shim.PutState("transaction-time", t)
  //... other operations ...
}
```

Figure 7.3: Example of issue of non-determinism with time APIs in blockchain software.

7.2 Non-Determinism in Go

Unlike DSLs such as Michelson, where there are only deterministic language features, Go is a GPL that contains methods and instructions of general use and is not specific to the blockchain context. Nevertheless, GPLs provide several components that can explicitly lead to non-determinisms, such as (pseudo-)random values generators or external computations. Furthermore, even some methods that are explicitly sequential and deterministic pose a threat when executed on different nodes, such as the `time.Now()` call from Figure 7.1. Despite these threats, popular blockchain frameworks such as HF and Cosmos SDK do not enforce particular restrictions on the usage of non-deterministic methods and components.

A first solution that comes to mind is to restrict the usage of some APIs known to introduce non-deterministic behaviors or, conversely, only allow a subset of the language that is known to be deterministic [187]. While this can fix the problem, it also forbids the usage of development tools that could be useful in non-harmful contexts. Consider the Go snippets reported in Figure 7.2 and Figure 7.3. Both fragments rely on the `time` API for retrieving a timestamp from the system. In general, the effects of the `time` APIs execution are subjective to the node and might lead to non-deterministic behaviors in blockchain, because each node of the network could have different system settings (e.g., time, date, time zones) or execute the code at slightly different instants. Figure 7.2 shows a safe use of the `time` APIs, as the timestamp is only used for logging with no consequences on the blockchain state or the execution result. Instead, Figure 7.3 reports a problematic usage of non-determinism, since the timestamp is stored in the blockchain using `PutState`, a specific function of the HF framework that updates the common state. Since timestamps could differ on the various nodes, or the nodes might have different time settings, this might potentially lead to inconsistent states or results at each execution, thus causing the transaction to fail.[2]

In the rest of this section, we identify, for each blockchain framework presented in Section 5.1, the *sources* of non-determinism, and the blockchain state modifiers and the response builders (that is, statements that make a transaction succeed or fail, as in the case of Figure 7.4), namely *sinks*. This will prepare the ground for the core contribution of this chapter: a static flow-based approach for detecting critical usage of non-determinism in blockchain software reported in Section 7.4.

Regarding the sources, Table 7.1 summarizes the Go instructions and libraries that we considered as causes of potential non-determinism. For the sake of simplicity, the table reports instructions and packages omitting the signatures of every single method. Only a few methods within those packages lead to non-deterministic behaviors. For instance, the package `time` contains different methods to handle dates and times. Most of them are not potentially risky and common in smart contracts and DApps. However, getting the current time of the OS (i.e. methods `Since`, `Now`, `Until`) is highly dangerous because the execution of the contract may differ from a peer machine to another one, depending on when a certain node executes the contract. The full list of sources is available at [7]. Regarding sinks, Table 7.2 summarizes the main instructions and components, grouped by framework, that are sensitive to non-deterministic behaviors.

Hyperledger Fabric v2.4. In HF, the world state may be changed using specific APIs. In the HF framework for Go, the interface `ChaincodeStubInterface` is the main component involved in the access and modification of the blockchain state. Table 7.2 reports the current elements involved in the data-write pro-

[2] In this case, the developer should have used the method `GetTxTimestamp` instead of `time.Now`, that is provided by HF framework for returning a safe timestamp shared by the nodes.

```
func (s *SmartContract) transaction(APIstub shim.
    ChaincodeStubInterface) sc.Response {

  if rand.Int() % 2 == 0 {
    return shim.Error("Fail")
  } else {
    return shim.Success(nil)
  }
}
```

Figure 7.4: Example of issue of non-determinism related to response in blockchain software.

Framework	*Package*	*Type/Interface*	*Statements/Methods*	*Critical point*
HyperLedger Fabric	`shim`	`ChaincodeStubInterface`	`PutState`	parameters
			`DelState`	parameters
			`PutPrivateData`	parameters
			`DelPrivateData`	parameters
			`Success`	statement
			`Error`	statement
Tendermint Core	`abci/types`	`Application`	`ResponseBeginBlock`	instance returned
			`ResponseDeliverTx`	instance returned
			`ResponseEndBlock`	instance returned
			`ResponseCommit`	instance returned
			`ResponseCheckTx`	instance returned
Cosmos SDK	`types`	`KVStore`	`Set`	parameters
			`Delete`	parameters
	`kv`, `dbadapter`, `gaskv`, `iavl`, `listenkv`, `prefix`, `tracekv`,	`Store`	`Set`	parameters
			`Delete`	parameters
	`types/errors`		`ABCIError`	statement
			`Redact`	statement
			`ResponseDeliverTx`	statement
			`ResponseCheckTx`	statement
			`WithType`	statement
			`Wrap`	statement
			`Wrapf`	statement

Table 7.2: Main sinks for blockchain software written in Go. The *Critical point* column describes the sensitive point of the method where non-determinism may lead to different executions in different nodes (i.e. in parameters, returned value, or the full statement).

posal. According to the execution model, the semantics of these elements do not affect the blockchain state until the transaction is validated and successfully committed. Hence, if these components provide different results (e.g. changes to the shared state with different values) due to non-determinism, the consensus mechanism will not validate the transaction proposal and no new state will be committed. Regarding the response statements, HF provides the `Success` and `Error` methods to get respectively successful and failed transaction responses.

Tendermint Core v0.35. Tendermint Core is a middleware that allows developers to create custom blockchain application layers as long as specific methods declared in the Application BlockChain Interface (ABCI) are implemented. Figure 7.5 reports a flow diagram of the consensus process used to validate and store a transaction using the ABCI methods. Although the logic of these methods is

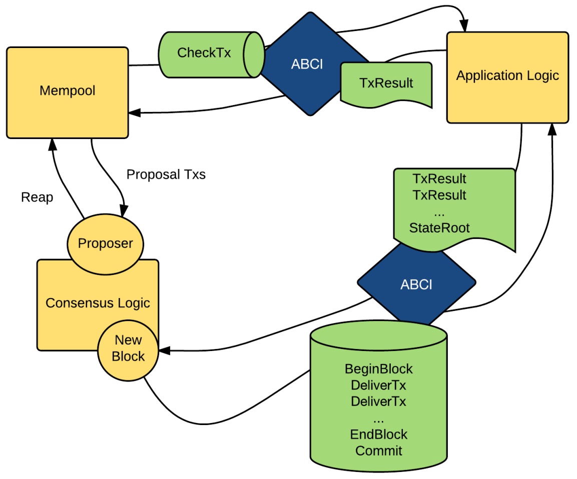


Figure 7.5: ABCI methods and consensus flow [107].

different, the structure is similar. All of them receive as input a request object and return a response object (`ResponseBeginBlock`, `ResponseDeliverTx`, `ResponseEndBlock`, `ResponseCommit`) that must be deterministic. As reported in the official documentation of Tendermint [191], among these methods, only `BeginBlock`, `DeliverTx`, `EndBlock`, and `Commit` must be strictly deterministic to force determinism of the code execution over the consensus connection.

Cosmos SDK v0.35. In Cosmos SDK applications, both the application and the blockchain state are handled through the *store* component[3], a set of key-value pairs used to store and retrieve data. However, the nature of this component can also be multistore, i.e. a store of stores, as shown in Figure 7.6. In this way, it is possible to encapsulate multiple states enabling the modularity of the Cosmos SDK. The idea is that each module may declare and manage its own subset of the state with specific keys, typically held by Cosmos SDK abstractions called *keepers*. The store components are defined by `Store` types, which are declared in several packages such as `kv`, `tracekv`, `gaskv`, and `ival`. Each of these definitions also implements the `KVStore` interface, which provides common APIs to access and modify the state of blockchain using the methods such as `Set` and `Del`. Regarding the response statements, Cosmos proposes several methods in the package `types/errors` to return failed transaction responses such as `ABCIError`, `Wrap`, and `ResponseDeliverTx`.

[3] https://docs.cosmos.network/master/core/store.html

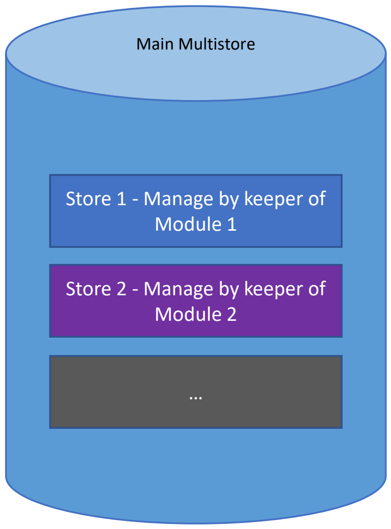

Main Multistore
Store 1 - Manage by keeper of Module 1
Store 2 - Manage by keeper of Module 2
...

Figure 7.6: Main store of Cosmos SDK.

7.3 An Information Flow Analysis Approach

In this section, we introduce and discuss our purpose for detecting non-deterministic behaviors in blockchain software. In particular, we consider a non-deterministic behavior as *critical* only if a non-deterministic value can affect the blockchain state, either directly (i.e. being stored inside the state) or indirectly (e.g., guarding the execution of state updates). Any other usage of non-determinism is considered safe, as it does not affect the blockchain state or response. As such, when mentioning non-determinism in the remainder of the chapter, we refer to this critical kind. We rely on information flow analysis for detecting when values originating from sources of non-determinism affect the state of the blockchain. We only focus on static analyses, that soundly over-approximate all possible behaviors of target programs and can thus give guarantees about the absence of non-deterministic behaviors. We instantiate two types of analyses: a taint analysis, able to capture the so-called *explicit flows*, and a non-interference analysis, that can also detect *implicit flows*.

7.3.1 An Overview on Information Flow

Information flow analyses [56,171] address the problem of understanding how information flows within the program during its execution. According to [57], information flows from an object x to object y, whenever information stored in x

```
var x, y
y := x
```

(a)

```
var y, x
if x = true
    then
y := 7
else
y := 13
```

(b)

```
var y, x
if x = 1 then
(* do some
    time-
    consuming
    work *)
y := 0
```

(c)

Figure 7.7: Example of (a) explicit, (b) implicit, and (c) side channel flows.

is transferred to, or used to derivate information transferred to, object y. Hence, these analyses are involved to detect program executions where information *flows* from one partition to the other. Flows can be classified into three types:

- *explicit flow*: when the information in object x flows is transferred explicitly to object y (Figure 7.7a);
- *implicit flow*: when an the information of object y implicit depends on information of object x (Figure 7.7b);
- *side channel*: when some observable property of the execution, for instance, the amount of computational resources used, depends on the information in object x (Figure 7.7c).

Traditionally, the objects holding values that one wants to track along program executions are called *source*, while the locations where information coming from sources should not flow are described with the term *sink*. This general terminology may be used to map different properties. For instance, if we are interested in ensuring the *integrity* of secret variables, information flow analyses can be instantiated by using public variables as sources and private ones as sinks, respectively. In this way, it is possible to detect when a possibly corrupted value provided by a malicious attacker could be stored in variables whose content is supposed to be safe. Otherwise, if we are interested in ensuring the *confidentiality* of secret variables, then the same analyses can be recast with private variables acting as sources and public ones as sinks, thus looking at flows in the opposite direction. In this case, the analysis goal is to detect all possible private data disclosures to external entities.

For non-deterministic behaviors in blockchain environments, we are interested in ensuring the *integrity* with respect to non-deterministic behaviors. Hence, information flow analyses may be exploited to detect when non-deterministic values flow into critical components for blockchain environments. As such, we mark the components that are initialized to non-deterministic values as *sources*, while the critical ones as *sinks*.

In this kind of analyses, we are interested only in explicit and implicit flows, in order to identify non-deterministic information that is either used to update the blockchain's state or a transaction's result, or that governs their execution flow. The reason is that side channels are typically studied to detect secret information leaking through, for instance, execution time, thus violating the confidentiality of that information instead of its integrity.

In the following, we describe two well-established information flow analyses that we applied for the detection of non-determinism issues.

7.3.2 Non-interference

Non-interference [83,84] captures the intuition that if computations over private information are independent of public information, then no leakage of the former can happen. This notion is often instantiated in language-based security where the space of variables of a program (denoted by P) are partitioned into *low* (denoted by $\mathbb{L}$, referred to as private or secret), and *high* (denoted by $\mathbb{H}$, referred to as public or available to anyone). The non-interference property is satisfied if changes in the high variables do not affect the observable (i.e. public) values of low variables in the program:

$$\forall v_L \in \mathbb{L}, \forall v_H, v'_H \in \mathbb{H}.\ \mathsf{P}(v_L, v_H) = \mathsf{P}(v_L, v'_H)$$

Given the non-interference, it is possible to design an information flow analysis able to find instances of explicit and implicit flows between low and high partitions. For each program point, the analysis computes a mapping from variables to the information level they hold (low or high), while also keeping track of an execution state depending on the information level of boolean conditions that guard the program point. Whenever an assignment to a variable in $\mathbb{H}$ either assigns a low value (that is, an expression involving variables in $\mathbb{L}$), or happens with a low execution state (that is, guarded by at least a boolean condition that involves variables in $\mathbb{L}$), a non-interference property violation occurs and the analysis can report it.

7.3.3 Taint Analysis

Taint analysis [67, 194] is an instance of information flow analysis that can be seen as a simplification of non-interference considering only explicit flows. In this context, variables (denoted by $\mathbb{V}$) are partitioned into *tainted* (denoted by $\mathbb{T}$) and *untainted* (denoted by $\mathbb{U}$), where $\mathbb{V} = \mathbb{T} \cup \mathbb{U}$ and $\mathbb{T} \cap \mathbb{U} = \emptyset$.

The variables contained in $\mathbb{T}$ represent those that can be tampered with by an attacker and the variables contained in $\mathbb{U}$ represent those that should not contain tainted values across all possible program executions. Roughly, taint analysis corresponds to the language-based non-interference instantiation without the execution state, thus unable to detect implicit flows.

This generic schema has been instantiated to detect many vulnerabilities in real-world software (e.g., SQL injection, cross-site scripting, redirection attacks), achieving significant practical results (see [71] as an example). Recently, we applied taint analysis to injection and privacy issues [75], also related to GDPR compliance [73].

7.4 Detection of Non-Determinism with GoLiSA

The main idea of our approach is to track with GoLiSA the values generated by the sources identified in Table 7.1 during the execution of a program using either taint analysis or non-interference. Similarly, after the analysis completes, we use a GoLiSA semantic checker to inspect the provided information by abstract computations, checking if any of the sinks specified in Table 7.2 receives one such non-deterministic value as a parameter or, in the case of non-interference, if the sink is found in a *low* execution state. In this way, it is possible to detect only potential harmful non-deterministic behaviors, allowing those which do not effect the blockchain consensus.

In GoLiSA, the analyses are instantiated as follows:

- taint analysis and non-interference are implemented as value domains, both of them being non-relational domains (i.e. mapping from variables to abstract values — taintedness and integrity level respectively — with no relations between different variables), with non-interference keeping track of the abstractions for each guard;
- field-insensitive program point-based heap domain [169, Chapt. 8.3.4], where any concrete heap location allocated at a specific program point is abstracted to a single abstract heap identifier;
- context-sensitive [114, 178] interprocedural analysis, abstracting full call-chain results until a recursion is found;
- run-time types-based call graph, using the run-time types of call receivers to determine their targets;
- two semantic checkers, for taint analysis and non-interference, that scan the code in search for sinks, checking the taintedness or integrity level of each sink and triggering an alert when an issue of non-determinism is detected.

The choice of domain and checker is left to the user and can be done with a specific option flag. Given an input program, the analysis begins detecting the statements annotated as sources and propagating the information from them. The analyses produce, for each program point, a mapping stating if each variable is the result of a non-deterministic computation. These mappings consist of abstract computations that are used by our semantic checkers. At this point, the checkers visit the program in search for statements annotated as sinks. When one is found, the mappings are used to determine if values used as parameters of the call are critical or, in the case of non-interference, if the call happens in a critical state. Then, a warning will be triggered in case of non-determinism detection.

For instance, let us consider the fragment reported in Figure 7.2. At Line 5, despite variable `elapsed` being marked as tainted, no warning is raised by GoLiSA regardless of the chosen analysis, as it does not reach any sensitive sink. Instead, the analysis of the fragment from Figure 7.3 results in the following alarm:

```
The value passed for the 2nd parameter of this call is tainted,
and it reaches the sink at the parameter "value"
```

The warning is issued with both analyses, since variable `t` is marked as tainted and reaches a blockchain state modifier through an explicit flow.

Consider now the example reported in Figure 7.1. Here, no explicit flow happens at Line 3, which contains the blockchain state modifier `Wrap`, but its execution depends on the non-deterministic value used in the condition at Line 2, that is, `time.Now().Unix()`. As this is an implicit flow, the taint analysis is not able the detect it. However, GoLiSA will discover it with non-interference, raising the following alarm:

```
The execution of this call is guarded by a tainted condition,
resulting in an implicit flow
```

7.4.1 Detection of Sources and Sinks with GoLiSA

As already reported in Section 7.3.1, the information flow analyses that we designed to detect non-deterministic issues must know which are the sources and sinks of the program. In this regard, GoLiSA provides a solution based on annotations, marking the corresponding statements as sources and sinks. Tables 7.1 and 7.2 show these components, while in the following, we describe how GoLiSA detects them.

Methods and functions. GoLiSA contains a full list of the signature of functions and methods to be annotated and it automatically annotates the corresponding calls in the program by syntactically matching them. As shown in Tables 7.1 and 7.2, all sinks and several sources correspond to functions and methods of APIs from either the Go run-time or the blockchain frameworks.

For instance, considering the following code snippet, GoLiSA is able to match the call to `time.Now`, that gets annotated as source, and the one to `PutState`, whose parameters get annotated as sinks:

```go
key := "key123"
tm := time.Now()
stub.PutState(key, []byte(tm))
```

Then, the flow analysis propagates information from the return value of `time.Now` to the second parameter of `PutState`, thus issuing an alarm at Line 3.

Map Iterations. The iterations on the maps are detectable by identifying the instructions that allow one to iterate data and by checking the type of the iterated object. In Go, the statement which makes iterating of data structures possible is `range`. GoLiSA exploits semantic reasoning about run-time types to be inferred by the analysis when the object in a `range` statement is inferred to be a map. Then, GoLiSA marks as sources the variables used to store keys and values of the map. Consider as an example the following code snippet:

```go
s := ""
mymap := map[string]string{"1": "Hello", "2": "World!"}
for key, value := range mymap {
  s += value
}
stub.PutState("key", []byte(s))
```

Analyzing the code, GoLiSA matches the `range` instruction at Line 3 and checks the type of its target parameter. GoLiSA annotates as sources both `key` and `value`, as `mymap` is inferred to be a map, while the sink at Line 6 is detected through already discussed method annotations. Flow analyses can then propagate the information from `value` to `s`, that in turn flows to the second parameter of `PutState`, issuing an alarm at Line 6.

Global variables. GoLiSA annotates every global variable syntactically matching them over all program components. The reason is global variables may be modified independently on each peer leading to non-deterministic issues, such as due to differences in the endorsement policy of each peer [126]. For instance, in the following code snippet, the value of global variable `glb` could differ from peer to peer depending on the number of times function `concat` has been executed. In fact depending on the policies of a peer `concat` could be performed or not.

```go
var glb string
func concat() {
  glb += "Hello World!"
}
func (s *SmartContract) transaction(stub shim.ChaincodeStubInterface) sc
    .Response {
  stub.PutState("key", []byte(glb))
}
```

Analyzing the code, GoLiSA iterates over all program components, annotating `glb` as a source. The `PutState` at Line 6 is annotated as sink as previously discussed for methods. Then, the information flow analysis propagates taintedness from `glb` to the second parameter of the call to `PutState`, raising an alarm at Line 6.

Go routines. GoLiSA inspects the code of Go routines, checking the scope of variables they use. If these are defined outside the routine using them, they are effectively shared among threads, potentially leading to race conditions and non-deterministic behaviors. Hence, GoLiSA annotates such variables as sources. For instance, the following snippet defines and invokes two simple Go routines that modify a variable defined in an enclosing scope:

```
1   s:= ""
2   wg.Add(2) //set a waitgroup for two go routines
3   go func(){
4     for i := 1; i <= 10000; i++ {
5       s += "0"
6     }
7   }
8   go func(){
9     for j := 1; j <= 10000; j++ {
10      s += "1"
11    }
12  }
13  wg.Wait() // waits until all the two go routines are done
14  stub.PutState("key", []byte(s))
```

Analyzing the code, GoLiSA detects the Go routines at Lines 3 and 8. It checks the scopes of each variable, inferring that `s` is declared outside the routines themselves. Hence, GoLiSA annotates `s` at Line 1 as a source, while the sink at Line 14 is annotated as previously discussed. Then, the flow analysis propagates the information from `s` to the second parameter of `PutState`, issuing an alarm at Line 14 since the value of `s` depends on the execution order of Go routine loop bodies, which it is non-deterministic because go routines run independently in parallel and without any check on the shared variable.

Go channels. Channels are pipes that connect concurrent Go routines through the operator `<-`. The allow blocking interaction among Go routines . GoLiSA annotates as sources the instructions reading values from channels, as the order in which these are written is intrinsically non-deterministic. As an example, consider the following code snippet:

```
1   s := make(chan string)
2   go routineA(s)
3   go routineB(s)
4   x, y := <- s, <- s
5   stub.PutState("key", []byte(x))
```

Analyzing the code, GoLiSA detects the occurrences of the operator `<-`. Hence, it annotates variables `x` and `y` as sources, because they receive a value from channel `s`. The sink at Line 5 is detected as previously discussed for the methods. Then, the flow analysis propagates the information from `x` to the second parameter of `PutState`, resulting in an alarm at Line 5.

7.5 Related Work

Non-determinism is a well-known issue in the development of smart contracts written in GPLs [126, 205]. Restricted languages such as Takamaka [187, 188] enforce determinism allowing only fully deterministic APIs. The white-listing approach ensures safe development while preventing API extensions coming with new language versions can bypass the check. However, a restricted language also severely limits the exploitable features of the GPL. On the other hand, black-listing undesired APIs is a much harder approach to maintain, but it seems the most widespread technique in Go analyzers. For instance, Chain-Code Analyzer [119] and Revive^CC [180] detect mainly black-listed imports related to non-deterministic APIs using a syntactical approach. Besides, they can detect non-deterministic map iterations by AST traversal with minimal syntactic reasoning. Signature of invoked functions can also be black-listed instead of imports [126]. These tools and frameworks inherently limit API usage, sensibly reducing the benefits of adopting a GPL even when the code poses no harm to the blockchain.

7.6 Experimental Evaluation

We experimentally evaluated the analyses implemented in GoLiSA to detect non-determinism issues in blockchain software. First, we studied them quantitatively, on a set of almost 300 real-world HF smart contracts retrieved from public GitHub repositories. Then, we evaluated the quality of our results on two applications, to show how the analyses work and how the information is propagated in programs. In particular, we selected the first application from the HF benchmark, while the second one is the Cosmos SDK code reported in Figure 7.1.

We chose HF framework for the quantitative evaluation compared to the other because it is the only framework supported by several static analyzers detecting non-determinism issues, and in particular by the ones involved in our comparison with GoLiSA. Furthermore, HF is currently the most popular and widespread blockchain framework among public GitHub repositories, with most chaincodes written in Go.

Environment Setup. All the experiments have been performed on a HP Elite-Book 850 G4 equipped with an Intel Core i7-7500U at 2,70/2,90 GHz and 16 GB of RAM memory running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17.

7.6.1 Quantitative Study: Hyperledger Fabric Benchmark

We created a benchmark of HF smart contracts written in Go, retrieving artifacts from 331 GitHub repositories. The artifacts have been selected by querying the keyword *chaincode* (i.e. the wording for smart contracts in HF) and selecting only non-forked Go repositories[4]. Then, we filtered out files unrelated to smart contracts, such as client software, build scripts, etc. We have kept only the Go files, and their dependencies, that use the `shim` package[5], because this package imports APIs able to access the blockchain state, the transaction context and to call other chaincodes. Hence, it contains critical components for the analysis of non-determinism. This resulted in 298 files ($\sim$74685 LoCs), that we refer to as $\mathbb{ND}$.

We first evaluate our semantic analyses (taint and non-interference), both from a precision and performance point of view, also showing a specific case study taken from $\mathbb{ND}$. Then, we compare GoLiSA with two open-source static analyzers: Revive^CC and ChainCode Analyzer, which are also discussed in Section 6.3. The first evaluation runs over $\mathbb{ND}$, showing that GoLiSA is able to semantically analyze over 95% and 93% of files with taint analysis and non-interference, respectively, with the other ones failing due to missing/incorrect handling of some language constructs or failures during the analysis. The comparison experiments, instead, run over a subset of $\mathbb{ND}$, corresponding to the common set of smart contracts that Revive^CC and ChainCode Analyzer are able to analyze. With these experiments, we will show that GoLiSA produces more precise alerts, outperforming the current state of static analyzers for blockchains targeting Go.

Table 7.3 reports the results of the experimental evaluation of GoLiSA over the benchmark $\mathbb{ND}$, where **AT** is the average execution time on each file, **#W** is the total number of warnings issued by the analysis, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings. The column of false negatives is not specified because analyses performed are *sound* (Section 4.3.1). Note that they are sound with respect to the property considered during the analysis while they check the same sources and sinks. Indeed, if *Taint* is used to detect every possible flows of non-determinism it would no longer sound, as the property it models focuses on explicit flows only and not implicit ones, leading the analysis to generate possible false negatives. Instead, *Non-interference* detects both types of flows. Nevertheless, only explicit flows were detected into $\mathbb{ND}$ thus no differences in the warning results were specified.

[4] `https://api.github.com/search/repositories?q=chaincodes+fork:false+language:Go+archived:false&sort=stars&order=desc`. Accessed: 12-10-2021.

[5] Available at `https://github.com/hyperledger/fabric-chaincode-go/shim`

Analysis	**AT**	**#W**	**#TP**	**#FP**
Taint	44.42s	2	2	0
Non-Interference	44.64s	2	2	0

Table 7.3: Evaluation of analysis for non-determinism detection.

In particular, GoLiSA is able to semantically analyze more than 280 Go files of the benchmarks, namely over 93% of them[6]. In terms of execution time, the analyses took on average under 45 seconds per file. Taint analysis is shown to be powerful enough on this benchmark, as it can detect all real issues with no false positives. Note that files were manually checked to ensure that no critical non-determinism was missed by the analysis. In fact, a syntactic analysis was performed on all the files to detect sources of non-determinism, identifying 1232 components. Each such component was manually checked to correctly classify the results of GoLiSA.

Tool Comparison

We compared GoLiSA with the available static analyzers described in Section 7.5, namely ChainCode Analyzer and Revive^CC.

The benchmark used for the comparison is limited to a subset of $\mathbb{ND}$ consisting of 46 smart contracts. This limitation is due to the ingestion problems found in the analyzers ChainCode Analyzer and Revive^CC, which did not allow us to correctly analyze a large part of the benchmark $\mathbb{ND}$. Then, the 46 smart contracts are the contract that can be analyzed by all tools (Table 7.4). Moreover, to avoid unsupported APIs and checks, the evaluation considers as sources only the categories *Random API, File system APIs, OS APIs, Database APIs, Internet APIs*, as these are the ones recognized by all analyzers. Table 7.5 shows the comparison between the tools, where **AT** is the average execution time on each file, **#W** is the total number of warnings issued by the analysis, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings, and **#FN** is the number of false negatives that were not warned.

As reported in Section7.5, ChainCode Analyzer [119] and Revive^CC [180] use a conservative approach denying every possible import contained in a black-list. This approach is thus *sound* when the black-list is exhaustive. However, the black-list of ChainCode Analyzer misses some harmful imports compared to Revive^CC and the sources considered by GoLiSA. For this reason, we classified ChainCode Analyzer as an *unsound* tool.

[6] GoLiSA fails to analyze the remaining smart contracts due to missing support to calls to C code via the built-in Go `cmd/go` package.

Tool	**Analyzed files**	$\mathbb{ND}$ **Coverage**
GoLiSA - Taint	284/298	95.30%
GoLiSA - Non-Interference	280/298	93.96%
ChainCode Analyzer	46/298	15.44%
Revive^CC	78/298	26.18%

Table 7.4: Coverage percentages of different static analyzers.

Tools	**#W**	**#TP**	**#FP**	**#FN**
GoLiSA - Taint	1	1	0	0
GoLiSA - Non-Interference	1	1	0	0
ChainCode Analyzer	0	0	0	1
Revive^CC	25	2	23	0

Table 7.5: Warnings triggered by the analyzers on the 46 files.

Regarding the results, GoLiSA finds the only true issue without noise (i.e. false positives), achieving the best and most accurate result. ChainCode Analyzer does not trigger any warning, missing one true vulnerability because not considered in the black-list. Instead, Revive^CC triggers 25 warnings out of which 96% are false positives with only two real issues reported. Note that the difference in true positives between GoLiSA and Revive^CC is due to GoLiSA issuing warnings on the sink rather than the source: in fact, the two Revive^CC warnings are reported on two different sources of non-deterministic values, but those are consumed by the same sink that corresponds to the one GoLiSA reports the warning on.

7.6.2 Qualitative Study

Explicit Flow: the Boleto contract. The *boleto* contract[7] is part of $\mathbb{ND}$. It seems to be a proof of concept application handling tickets in an e-commerce store, with the method `registrarBoleto` used to register a ticket (Figure 7.8). In particular, this method contains a real non-determinism issue that has been detected by GoLiSA and also by Revive$\hat{C}$C during the tool comparison.

Analyzing the code of *boleto*, GoLiSA identifies the explicit flow leading to a non-deterministic behavior with both taint and non-interference analyses. The affected method is `registrarBoleto`, which contains two different sources of non-determinism that directly flow into the same sink. The first is the generation of random values to generate a barcode at Line 3, through the method `rand.Intn` (*Random API*). Instead, the second source is the usage of the local machine's time to set a date at Line 4, through the method `time.Now` (*OS*

[7] https://github.com/arthurmsouza/boleto/blob/master/boleto-chaincode/boleto.go

```
1   func (s *SmartContract) registrarBoleto(APIstub shim.
      ChaincodeStubInterface, args []string) sc.Response {
2     // [...]
3     objBoleto.CodigoBarra = strconv.Itoa((rand.Intn(5) + 10000000 + //
        [...]
4     var notExpiredDate = time.Now()
5     objBoleto.DataVencimento = notExpiredDate.Format("02/01/2006")
6     // [...]
7     boletoAsBytes, _ := json.Marshal(objBoleto)
8     APIstub.PutState(args[0], boletoAsBytes)
9     // [...]
10  }
```

Figure 7.8: Method `registrarBoleto` of *boleto* contract.

API). The information from both sources flows to the fields of `objBoleto`, then to `boletoAsBytes` as well. Finally, the information tracked by the analysis flows into the second parameter of `PutState`. As reported in Table 7.2, GoLiSA considers the `PutState`'s parameters as sinks, then after the analysis computation, the checker will trigger an alarm at Line 8. Note that, according to the official documentation of HF[8], the `PutState` method does not affect the ledger until the transaction is validated and successfully committed. However, a transaction needs to produce the same results among different peers to be validated, thus causing the transaction to fail in case of non-deterministic values.

Implicit Flow: Cosmos SDK v.43. In the official release v. 0.43.x and v. 0.44.{0,1} of Cosmos SDK, as reported by the NIST database [145], there is a bug related to non-determinism. Analyzing the code in Figure 7.1, GoLiSA is able to detect an implicit flow that leads to a non-deterministic behavior, that can only be detected using non-interference. The `ValidateBasic` method was designed to validate a grant to ensure it has not yet expired. However, the implementation naively used the local machine time to perform the operation. In this case, GoLiSA detects the `time.Now` as source *OS API* at Line 2. By propagating the information, GoLiSA infers that the expiration check governs the execution of the return statement. Since the `Wrap` method is annotated as a sink, GoLiSA raises an alarm at Line 3 because it is contained in a block whose guard depends on non-deterministic values.

7.6.3 Limits

We conclude this section by highlighting the limits of the proposed approach.

Our experimental results may be surprising, given the absence of false positives and the few true positives detected by GoLiSA, especially for a fully-

[8] `https://github.com/hyperledger/fabric-chaincode-go/blob/1476cf1d3206f620db7eea12312c98669d39fa22/shim/interfaces.go`.

based static analysis approach, being the analyses proposed based on over-approximations. Although few true alarms have been detected, nondeterminism is an issue that should not be underestimated and is also clearly felt by the communities of the blockchain frameworks covered in this chapter. As a representative example, the Tendermint Core documentation [108], while discussing non-determinism, reports:

> *While programmers can avoid non-determinism by being careful, it is also possible to create a special linter or static analyzer for each language to check for determinism. In the future, we may work with partners to create such tools.*

Nevertheless, the low ratio of true positives and the absence of false positives reported by our experimental evaluation was at first surprising even to the authors of this chapter. We justify this by highlighting the context of applications of the blockchain frameworks discussed here, such as HF. Unlike some frameworks and GPLs used in other blockchains, these are used to develop *permissioned*, and often *private*, blockchains, meaning that the related software is not publicly available or released with open-source licenses. This is also the reason why the benchmark $\mathbb{ND}$ crawled from GitHub consists of a few hundred chaincodes, a number that is not comparable with smart contract benchmarks obtained investigating other (public and permissioned) blockchains. For instance, [198] collects 3075 distinct smart contracts from the Ethereum blockchain, resulting in a wider benchmark.

The proposed solution for detecting non-deterministic behaviors is fully static. It is well known that static analysis is intrinsically conservative and may produce false positives. Even if none have been raised by GoLiSA on the selected benchmark, one should expect false positives when applying our approach to arbitrary DApps.

7.7 The Industrial Case of Study: Commercio.network

This section studies the impact of non-deterministic behaviors in a blockchain from an industrial perspective analyzing the code of the open-source software provided by the Commercio.network company. Besides identifying these issues, it also discusses the implications during the development phase, such as mitigating risks before release and problems related to a possible patch.

7.7.1 The Blockchain Application of Commercio.network

Commercio.network [42] is an open-source decentralized application provided by the homonymous company[9]. As a blockchain, it can be described as a *permissioned* Proof-Of-Stake network, where a validator must join a consortium for being able to participate in the consensus. It can be described also as *public* since anyone can set up a node[10] and synchronize it with the Commercio.network main-net. The main purpose of this blockchain is to exchange electronic documents in a legally binding way thanks to the eIDAS Compliance[11], while following the principles of Self-Sovereign Identity [4].

As shown in Figure 7.9, the architecture of Commercio.network is based on the *Cosmos SDK* and its functionalities are contained in *modules*. A module conventionally revolves around the *keeper*, a package and entity implementing its core functionalities. The Cosmos SDK can also be seen as a collection of modules, that can be used to build custom ones, following the object-capability model [58]. For example, the Commercio.network module `commerciokyc` uses the keeper of another custom module, `commerciomint`, along with other modules coming from the library of Cosmos SDK.

7.7.2 Limits of the Cosmos SDK Toolbelt

A toolbelt is a set of applications useful for code development and software maintenance.

For the Cosmos SDK, there is a limited number of tools tailored to the framework. For instance, *Ignite CLI*[12] (formerly known as Starport) is the most popular platform to build, launch, and maintain blockchain applications based on Cosmos SDK. Although it facilitates software development, it has limitations. One of its most used features is to start a blockchain node in development with live reloading, i.e. when Ignite CLI detects that the source code of a Cosmos application has changed, it restarts the build process, and then it launches a network using the updated software. However, at the time of writing, developers using it cannot observe non-determinism problems while testing the application. In fact, the execution happens on a single node network and therefore the underlying consensus mechanism will never conflict as a cause of non-deterministic executions. Hence, some kinds of issues related to non-determinism are really difficult to be detected during the development phase, since there is only one participant. Still, it is possible to use the Go toolbelt for testing a Cosmos SDK application. But this leads to limitations in the development since the testing and verification phase have not been designed ad hoc for blockchain frameworks.

[9] https://commercio.network/
[10] https://github.com/commercionetwork/commercionetwork
[11] https://digital-strategy.ec.europa.eu/en/policies/eidas-regulation
[12] https://github.com/ignite-hq/cli

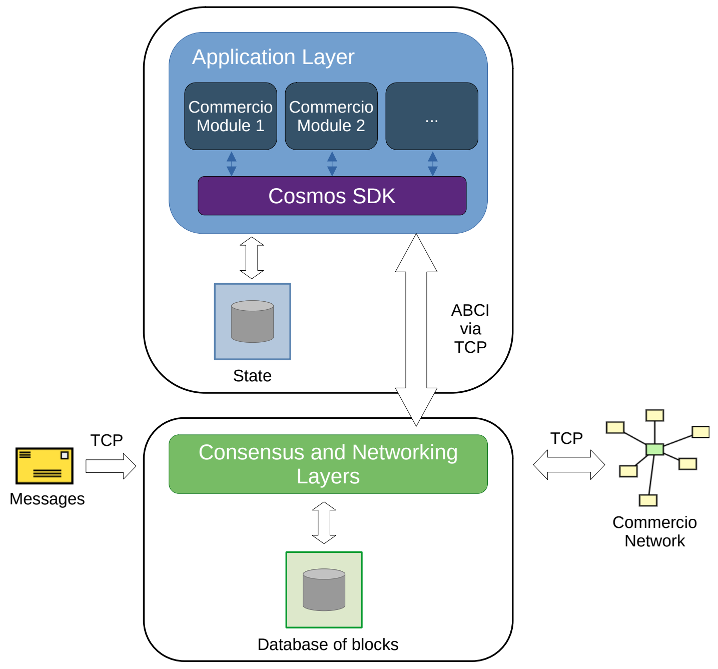


Figure 7.9: Commercio.network architecture.

As reported by the company, determinism is hard to ensure also after a complete test cycle, without the help of formal verification. Moreover, the complete test cycle is expensive in terms of resources and requires tools able to simulate the blockchain consensus. First, testing happens on a local-net. That is, a lightweight network running in a sandbox environment, is destroyed and relaunched before each test. Commercio.network relies on a containerized local-net solution. Then, at least one network exposed to the public is required. These are fully-fledged networks, mirroring the functionalities of the current main network version (test-net) or featuring experimental features (dev-net). Similarly to a main-net, these networks should be composed of a diversified ecosystem of devices with a significant amount of nodes, different operating systems, system settings, geolocations, and so forth. However, also this level of testing does not guarantee the detection of faults, because it is not a sound procedure. If it is based on a limited number of transactions, then the problems might exist but remain undetected, since the conditions required to spot them have not been reached.

Regarding analysis and verification tools, it is possible to start verifying Cosmos SDK applications with tools for the Go language, to check the code quality and detect issues. For instance, the Commercio.network company performs different iterations with Go analyzers on its code and publicly shares the results

Analysis	#A	#U	AT	#W	#TP	#FP
Non-interference	2 files	246 files	13.17s	2	2	0

Table 7.6: Analysis evaluation.

with the community using a Go Report Card[13]. However, as reported by the Commercio.network company, the code coverage reached by dynamic testing is limited because the standard Go test suite has not been designed for blockchain development. Nevertheless, the advancement of coverage gets publicly reported, too[14].

We should recall that bugs in the testing tools exist, too, along with incorrect test design and malformed testware. Incomplete or incorrect sets of test cases that do not fail, displaying the green *OK* flag, add a false sense of trust to the programmer, which could let their guard down. Therefore, it is important to apply formal verification to detect problems from the early stages of implementation. Indeed, these tools are not enough to guarantee the safety of blockchain software because, while analyzing generic properties for Go, they do not take into account the particularities of blockchain development nor specifically the problems related to the *Cosmos SDK* framework. Hence, the need to develop customized tools for the framework of interest, as in our case the analysis on determinism with GoLiSA.

7.7.3 Code Evaluation

Our target application is Commercio.network v2.2.0[15], and in particular the evaluation is performed on the 248 Go files (14961 LoCs) contained in the repository.

Environment Setup. All the evaluations have been performed on a HP EliteBook 850 G4 equipped with an Intel Core i7-7500U at 2.70/2.90 GHz and 16 GB of RAM memory running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17.

Table 7.6 shows the result of the analysis performed by GoLiSA, **#A** is the number of affected files (i.e. files where at least a warning was issued), **#U** is the number of unaffected files (i.e. files where no warnings were raised), **AT** is the average execution time on each file, **#W** is the total number of warnings issued by the analysis, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings.

[13] `https://goreportcard.com/report/github.com/commercionetwork/commercionetwork`
[14] `https://app.codecov.io/gh/commercionetwork`
[15] `https://github.com/commercionetwork/commercionetwork/tree/v2.2.0`

The analysis highlighted two problems regarding the same insidious issue. Something similar affected an older version of the Cosmos SDK, which was reported by the NIST database as *"vulnerable to a consensus halt due to non-deterministic behavior"* [145] caused by the use of the local clock time, obtained with the Go library `time`.

Bug Discussion

Bug #1. The bug appears in the `keeper` package of the module `commerciokyc`, in the method `Membership`. It is located at Line 89 of file `keeper.go`[16]. In a nutshell, the method allows assigning a Commercio.network *membership* of the given type to the specified user. As shown in Figure 7.10, the issue involves two main components: the method `time.Now` and the return of an error wrapped error. The first is a standard API of the Go language providing the current time of the device on which it runs. The second returns an error (wrapped with the `Wrap` method of the Cosmos SDK library[17]) to the method caller, leading to a transaction failure. Transaction executions among nodes must return a common result to achieve an update of the global status of the blockchain through the consensus mechanism. However, the current time provided by `time.Now` could be different from device to device because of custom settings (e.g. unsynchronized time, different time zones, etc.). Then, the same code execution on the nodes of the blockchain network could result in different values, breaking the consensus.

An invocation to the buggy method may return an error or not depending on the result of the following guard:

```
expited_at.Before(time.Now())
```

Inspecting the invocations of `AssignMembership`, it can be found that the input variable `expited_at` is a timestamp computed with a support function that adds one year to the block time. Nodes with the current local time set to a timestamp that makes the guard evaluate to true (a timestamp bigger than `expited_at` is enough) will mark transactions invoking this code as failing since it returns an error. If the majority of nodes behave in this way, a denial of service may occur and block the assignment of new memberships. However, in this case, the blockchain is not compromised only if malicious actors control the majority of the nodes, but the problem is due to a code bug during software development that does not allow it to reach a majority on the blockchain with the same result. Because of these reasons, `AssignMembership` does not ensure determinism, and its invocation might break the consensus mechanism.

[16] Source code available at: https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciokyc/keeper/keeper.go#L89

[17] https://docs.cosmos.network/master/building-modules/errors.html

```
func (k Keeper) AssignMembership(ctx sdk.Context, /*[...]*/,
    expited_at time.Time) error {

  /* [...] */

  if expited_at.Before(time.Now()) {
    return sdkErr.Wrap(sdkErr.ErrUnknownRequest, fmt.Sprintf("Invalid
        expiry date: %s", expited_at))
  }

  /* [...] */
}
```

Figure 7.10: *Bug #1.* A snippet of the `AssignMembership` method not ensuring determinism in the `commerciokyc` module.

Bug #2. The bug appears in the `keeper` package of the `commerciomint` module, in the method `BurnCCC`. It is located at Line 174 of file `keeper.go`[18]. In a nutshell, this method allows burning (i.e. removing) an amount of currency to the conversion rate stored in a *position*, retrievable from the keeper's store with a user account address and an id. If successful, `BurnCCC` gives back to the user the collateral amount, then updates or deletes the considered position but only if enough time, called *freeze period*, has passed since its creation. Similarly to Bug #1, as shown in Figure 7.11, also this issue involves two main components: the non-deterministic method `time.Now` and the return of a wrapped error.

An invocation to the buggy method may return an error or not depending on the result of the following guard:

`time.Now().Sub(pos.CreatedAt) <= freezePeriod`

The `pos` variable represents a position stored in the module's keeper and it is used to read its self-documenting field `CreatedAt`. The timestamp `freezePeriod` is read from the store of the module, too. Nodes with the current local time set to a timestamp that makes the guard evaluate to true (a timestamp in the past is enough) will mark transactions invoking this code as failing since it returns an error. If the majority of nodes behave in this way, a denial of service may occur and block the redemption of funds in the positions. This is not caused by malicious actors but is due to a code bug during software development, similar to the previous case. Because of these reasons, `BurnCCC` doesn't ensure determinism, and its invocation might break the consensus mechanism.

[18] Source code available at: https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciomint/keeper/keeper.go#L174

```
func (k Keeper) BurnCCC(ctx sdk.Context, user sdk.AccAddress, id
    string, burnAmount sdk.Coin) error {

  pos, found := k.GetPosition(ctx, user, id)
  if !found { /* [...] */ }

  // Control if position is almost in freezing period
  freezePeriod := k.GetFreezePeriod(ctx)
  if time.Now().Sub(pos.CreatedAt) <= freezePeriod {
    return sdkErr.Wrap(sdkErr.ErrInvalidRequest, "cannot burn position
         yet in the freeze period")
  }

  /* [...] */
}
```

Figure 7.11: *Bug #2*. A snippet of the `BurnCCC` method not ensuring determinism in the `commerciomint` module.

Bug patching

After a deep investigation, the company reports that no incidents or transaction failures happened because of these bugs during the live period of the release V2.2.0. Both bugs were patched in the major release v3.0.0. Then, we performed the analysis on this release and we could not find problems. The issue has been resolved by getting the time directly from the current Tendermint block header, a source that is both deterministic and supported by consensus. More in detail, the Cosmos SDK context method `ctx.BlockTime()` has been used instead of `time.Now()` when the current time was needed.

Dynamic Testing Considerations

The packages containing the bugs were tested with the standard Go testing framework and the libraries supported by Cosmos SDK, obtaining a satisfying level of code coverage. In particular, for the `keeper` packages of `commerciomint` and `commerciokyc` at version v2.2.0 test coverage is respectively 83.9% and 91.9%. However, both defects could not be detected by the test cases. In particular, these tests are not based on formal methods and cannot ensure the absence of bugs. Moreover, the high scores returned by those tools gave a false sense of security, which led to the deployment of buggy software in the real world.

7.8 Conclusions

The issues related to non-determinism are non-trivial and subtle. Several sources lead to these behaviors, and many are often overlooked by programmers, leading them to be the worse enemy of themselves. For this reason, it is necessary to apply automatic tools for verification. Furthermore, attention must also be paid to the choice of these tools, both from the point of view of covering the problems and the quality of the results. Indeed, an inadequate tool can give a false sense of security, while a large amount of false positive warnings could discourage programmers or cause them to let their guard down during bug fixing.

The analyses proposed in this chapter allow one to take a step forward in the state of the art and practice regarding non-determinism problems in blockchain software, empirically proving their applicability and their accuracy on real-world software. Information flow techniques are used in various fields for tracking information within the software, but to the best of our knowledge, they have never been applied so far to track non-determinism properties on blockchain software. In addition, we think the intuition of *"only those that affect the status or response of the blockchain can cause problems within the blockchain"* may also be used to increase the accuracy of other analyses for blockchain software, or at least set a different severity for warnings (e.g. *high* if it affects blockchain state or responses, otherwise *low*) helping developers in bug fixing activities.

The next chapter deals with another possible application of information flow analysis for the detection of problems related to *untrusted cross-contract invocation* issues.

Chapter 8

UNTRUSTED CROSS-CONTRACT INVOCATION DETECTION

When deployed in blockchain, smart contracts are immutable programs with parametric inputs given by transactions. Although the code cannot be changed, they are subject to different execution behaviors depending on the inputs received. Indeed, it is possible to use particular inputs, depending on how a contract is implemented, to trigger harmful and potentially unexpected behaviors. In this chapter, we introduce the issues related to *untrusted cross-contract invocation* (UCCI) and we provide an analysis based on *taint* analysis able to detect this kind of issues. To the best of our knowledge, this chapter proposes the first analysis for UCCI detection for Go and Michelson based on information flow approach. Some contents of this chapter are under publication at Workshop on Blockchain theoRy and ApplicatIoNs (BRAIN'23).

8.1 Untrusted Cross-Contract Invoking

The methods of a smart contract C deployed in blockchain can be executed directly, with a call originated from outside the blockchain, or indirectly, as an internal *cross-contract* call (a.k.a. *delegate call* or *external contract call*) from inside another contract. This latter case is used for instance to query the state of C or to execute one of its external methods. A typical example is the execution of a token transfer call on C, where C is passed as input to another contract. However, the input coming from outside the blockchain is untrusted: any user can provide it, also anonymously, at least in permissionless blockchains. This is fine as long as the method of C that gets invoked is not redefined.

```
1    // Get the args from the transaction
2    args := stub.GetStringArgs()
3
4    contract := args[0]
5    /* [....] */
6    queryArgs[0] = "my-method"
7    queryArgs[1] = myasset
8    response := stub.InvokeChaincode(contract, queryArgs, "main-channel")
```

Figure 8.1: A simplified smart contract for Hyperledger Fabric containing an UCCI.

Otherwise, it is possible to induce the execution of the arbitrary code placed in its redefinition [39, 44, 181]. That code could move assets or currencies among contracts, in a way that was not expected. This issue affects both GPLs and DSLs, including Go and Michelson languages.

Consider for instance the Go smart contract snippet in Figure 8.1. At Line 2, it retrieves the input of a transaction request with function `GetStringArgs`. At Line 4, it stores the first element of the input in variable `contract`, later used at Line 8 as receiver of a cross-contract invocation of method `my-method` contained in `queryArgs` with some arguments. This is a security problem since the user controls `contract`. Since there is no check in the smart contract, the user can send execution requests on the channel `main-channel` to any deployed contract, including a contract not provided by the developer of the snippet in Figure 8.1. For instance, `contract` might be a malicious contract and one of its methods could unexpectedly transfer the ownership of `myasset`.

8.2 Detection by Taint Analysis

The detection of UCCIs can be mapped as a *tainteness* problem. As reported in Section 7.3.3, taint analysis allows one to detect if information from a source explicitly flows to critical program points called sinks. Likewise, we can assume as sources the input parameters given by the users through the transactions. The parameters of cross-contract calls specifying a contract are sinks. In this way, by performing a taint analysis, it is possible to trace arbitrary input values within a smart contract and check if there are flows that lead to cross-contract calls, then execute an arbitrary contract.

Note that we have specifically chosen taint analysis to detect only explicit flows. The reason is that attackers have rarely exploited implicit flows to arbitrarily change a contract. Even the side channels are not of interest as they do not affect the type of attack. Moreover, the identification of implicit flows leads to very conservative results, i.e. too many false alarms.

The main idea of our approach is to track with GoLiSA and MichelsonLiSA the values generated by the sources identified in Tables 8.1 and 8.2 during the execution of a program using taint analysis. Similarly, after the analysis completes, we use semantic checkers to inspect the provided information by abstract computations, checking if any of the sinks specified in Tables 8.1 and 8.2 receives one such untrusted input value as parameter.

GoLiSA and MichelsonLiSA analyses are instantiated as follows:

- taint analysis is implemented as non-relational value domains (i.e. mapping from variables to abstract value taintedness with no relations between different variables);
- field-insensitive program point-based heap domain [169, Chapt. 8.3.4], where any concrete heap location allocated at a specific program point is abstracted to a single abstract heap identifier;
- context-sensitive [114, 178] interprocedural analysis, abstracting full call-chain results until a recursion is found;
- run-time types-based call graph, using the runtime types of call receivers to determine their targets;
- semantic checkers for taint analysis to scan the code in search for sinks, checking the taintedness of each sink and triggering an alert when an issue of UCCI is detected.

Given an input program, the analyses begin by detecting the statements annotated as sources and propagating the information from them, through a fixpoint algorithm. After the fixpoint converges, the analyses produce, for each program point, a mapping stating if each program variable is the result of an untrusted input computation or not. At this point, the checkers visit the program in search of statements annotated as sinks. When one is found, the mappings are used to determine if the values used as parameters of the call are critical. Then, an alert will be triggered in case of UCCI detection.

8.2.1 Detection of Sources and Sinks with GoLiSA and MichelsonLiSA

The first step before performing a taint analysis is the identification of sources and sinks of the target blockchain framework.

Table 8.1 summarizes the Go instructions and libraries that we considered as causes of arbitrary inputs and cross-contract invocations. Currently, GoLiSA supports three different blockchain frameworks, i.e. Hyperledger Fabric, Cosmos SDK, Tendermint Core. As described in Section 5.1, only HF provides natively smart contract APIs written in Go. The others do not provide official APIs to

Framework	*Package*	*Type/Interface*	*Statements/Methods*	*Critical point*	*Category*
HyperLedger Fabric	`shim`	`ChaincodeStubInterface`	`GetArgs`	return value	Source
			`GetStringArgs`	return value	Source
			`GetFunctionAndParameters`	return value	Source
			`GetArgsSlice`	return value	Source
			`InvokeChaincode`	2^{nd} parameter	Sink

Table 8.1: HF methods of interest for the detection of UCCIs.

Framework	***Michelson IR Statement***	***Critical point***	***Category***
Michelson	`parameter_storage`	return value	Source
	`TRANSFER_TOKENS`	3^{rd} parameter	Sink

Table 8.2: MichelsonLiSA statements of interest for the detection of UCCIs.

perform cross-contract invocation, although they may support smart contract frameworks with custom or third-party implementations. For sake of simplicity, we cover only HF but the same approach can be applied to any other smart contract framework. Anyway, in HF, all sources are methods that return the arguments of the transaction invocation. Instead, the method `InvokeChaincode` allows one to request the execution of another installed contract.

For Michelson, Table 8.2 summarizes the instructions in MichelsonLiSA IR form. Michelson retrieves inputs implicitly, i.e. pushes them directly on the stack at the start of execution without an explicit call to some instruction, which in our MichelsonLiSA IR is explicitly reported as `parameter_storage()` (Section 6.2.2). It provides only an explicit sink called `TRANSFER_TOKENS`. This instruction consumes three stack elements, including the contract to be executed, and pushes a transfer operation element emitted by the contract.

Methods and functions. As shown in Tables 8.1 and 8.2, all sinks and sources correspond to functions and methods. GoLiSA and MichelsonLiSA contain a full list of the signature of these instructions. They automatically annotate the corresponding calls in the program by syntactically matching them.

8.3 Related Work

Verification tools can prevent untrusted cross-contract invocations before software deployment.

For instance, ContractFuzzer [111] generates fuzzing inputs and defines test oracles to detect security vulnerabilities including problems related to delegate calls in Solidity. The tool contains an offline EVM instrumentation and an online fuzzing tool. The offline EVM instrumentation process is responsible for monitoring the execution of smart contracts to extract information for vulnerability analysis. The online fuzzer analyses the smart contract under test with additional information, such as its *ABI* interface. SolGuard [163] detects untrusted external call issues at compile time in Ethereum and focuses mainly on smart contract-based multi-agent robotic systems. It implements the analyses using AST traversing and semantic flow checking.

Mythril [154] bases the analyses on symbol execution and concrete execution techniques to discover vulnerabilities including which untrusted external and delegate calls. It combines static execution with dynamic execution to improve path coverage and detection accuracy. Note that the symbolic execution approach does not guarantee the exploration of all program paths, leading potentially to false negatives.

SMARTSHIELD [208] dynamically highlights state changes and alterations after external calls. It analyses both the AST and the unrectified EVM bytecode of each contract to extract its bytecode-level semantic information. Then, the tool fixes insecure control flows and data operations through control flow transformation and the insertion of instruction sequences that perform certain data validity checks.

The tools described up to this point are all related to Ethereum smart contracts. To the best of our knowledge, only Wang et al. [199] propose a general platform to detect UCCI issues for smart contracts written in other programming languages. They describe a general platform that builds the ASTs for each smart contract and obtains the semantic descriptions of corresponding functions and variables. Hence, it generates assertions by knowledge of security model libraries and semantic descriptions of ASTs, and expressions, and then detects the defects of smart contracts. However, as also stated by the authors, there are still some problems that need further research and improvement. In particular, they use manual assertions, which in case of implementation errors can lead to omissions.

However, specifically for Go and Michelson, as far as we know, this chapter proposes the first analysis for UCCI detection for smart contracts written in these languages.

8.4 Experimental Evaluation

This section experimentally evaluates the analyses implemented in GoLiSA and MichelsonLiSA to detect UCCI issues in smart contracts. The evaluation is performed on a set of real-world smart contracts.

We created the $\mathbb{HF}$ benchmark of HF smart contracts written in Go, retrieving artifacts from 954 GitHub repositories. The artifacts have been selected querying for the *chaincode* keyword (the term used for HF's code) and selecting

Analysis	$\mathbb{HF}$ **Coverage**	**AT**	**#W**	**#TP**	**#FP**	**Timeout** (over 10 min)
Taint	93.66%	21.56 sec	46	46	0	5 chaincodes

Table 8.3: UCCI analysis evaluation for GoLiSA.

Analysis	$\mathbb{TZ}$ **Coverage**	**AT**	**#W**	**Timeout** (over 10 min)
Taint	100%	20.89 sec	3063	0

Table 8.4: UCCI analysis evaluation for MichelsonLiSA.

smart contracts from unforked Go repositories only[1], that include the `Invoke` and `Init` methods[2]: these are the transaction requests' entry points. $\mathbb{HF}$ consists in 962 chaincodes, ∼272307 LoCs.

For MichelsonLisa, 1000 smart contracts written in Michelson and containing the instruction `TRANSFER_TOKENS` from [165] were randomly taken. This resulted in a benchmark of 770060 LoCs, which we refer to as $\mathbb{TZ}$.

Environment Setup. All the experiments have been performed on a MacBook Air equipped with an Intel Core i5 dual-core at 1.6 GHz and 8 GB of RAM, macOS Big Sur 11.5.2, Oracle JDK version 11, and Go version 1.17.

8.4.1 Experimental Result Discussion

Table 8.3 and Table 8.4 report the evaluation results for GoLiSA and Michelson-LiSA, respectively, where **AT** is the average execution time on each file, **#W** is the total number of warnings issued by the analysis, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings. Note that the column of false negatives is not specified because the UCCI analyses performed are *sound* (Section 4.3.1) for the detection of *explicit flows that leads to UCCIs* considering the sources and sinks of Section 8.2.1.

Comparing the data of Table 8.3 and Table 8.4, it can be noted that the use of cross-contract calls is much more frequent in the blockhain of Tezos than in HF. The main reason is that in public permissionless blockchains it is the only method to perform automatic interactions between peers and blockchain components. In addition, being a trusted environment, the peers of a permissionless blockchain want to operate almost exclusively on-chain, in order to avoid fraud or misconduct. Instead, HF is most involved in permissioned blockchains which are generally private or consortium. Here, the tendency is to use the blockchain as a means of support, preferring other off-chain software channels to perform

[1] `https://api.github.com/search/repositories?q=chaincode+fork:false+language:Go+archived:false&sort=stars&order=desc`. Accessed: 17-10-2022.
[2] See `https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim`.

complex operations that do not strictly require the guarantees provided by the blockchain.

The type of problem as already mentioned depends on the semantics of the contract and its functional requirements in the design phase. Indeed, they could be safely used to query information from another contract or to execute a safe function. However, not knowing whether the cross-invocation requirement was expected or not, in the benchmarks, we considered true positive warnings all the *cross-contract call that receives as target contract information from an untrusted user input.* In general, this analysis is designed for those who need to ensure that an UCCI does not take place before deploying or run-timing a contract code.

Experimental Results of GoLiSA

Table 8.3 shows the results of the cross-contract invocation analysis checker over $\mathbb{HF}$. GoLiSA works for 901 chaincodes (93.66% of $\mathbb{HF}$) by using $Taint$. Analysis time is a few seconds per chaincode, on average. We found that in the benchmark there are 186 sinks distributed in 72 chaincodes. In total, the checker issues warnings for 46 of these sinks. After manual investigation, all are true positives. This result is exciting, especially because it is well known that the taint analysis is subject to over-approximation and the production of false positives since it only tracks binary information (*taint*/*untaint*) among the program. We remember that static analysis can still generate false positives on other examples of code.

Investigating the reasons for this precision, we realized that it is mainly due to the adoption of best practices and principles of good programming in the blockchain software development by developers:

- ***Design interactions a priori***. During the design phase, it is common to decide the interactions that a contract will have both with users and with deployed contracts within the blockchain, in order to avoid unexpected behaviors. Many sinks in $\mathbb{HF}$ take hardcoded strings as target contracts, leading them to make invocations only to known contracts already present in the blockchain. In terms of analysis, this allows reducing the over-approximations because a hardcoded constant string will be *untainted* if not modified by other untrusted statements.
- ***Keep contracts simple***. The complexity increases the likelihood of errors and the cost of failure can be high in blockchain, given the code's immutability. Hence, the code is modularized to keep contracts and functions small (e.g., in $\mathbb{HF}$ each chaincode is $\sim$283 LoCs on average). This leads to propagating less information than in traditional software, thus limiting the possible over-approximations in the analysis due to a greater number of inference passages. Moreover, in $\mathbb{HF}$, the information is propagated mainly through

```
1      func (t *UnionLoanChaincode) Invoke(
2      stub shim.ChaincodeStubInterface) pb.Response {
3        function, args := stub.GetFunctionAndParameters()
4        if function == "offer" {
5          return t.offer(stub, args)
6          /* [....] */
7        }
8
9        func (t *UnionLoanChaincode) offer(
10       stub shim.ChaincodeStubInterface, args []string) pb.Response {
11         /* [....] */
12         var chainCodeToCall = args[0]
13         var loanId = args[1]
14         /* [....] */
15         var participants [10]Participant
16         /* [....] */
17         invokeArgs := util.ToChaincodeArgs(f,
18         participants[i].BankName, customer,
19         strconv.FormatFloat(participants[i].Balance,
20         'f', 0, 64))
21         response := stub.InvokeChaincode(chainCodeToCall, invokeArgs, ""
               )
22         /* [....] */
23       }
```

Figure 8.2: Simplified code from *chaincode_union_loan*.

local assignments of simple types, then even the approximations of complex data structures are limited. As reported in [126], the use of external libraries is also discouraged, therefore the approximation due to lack of external code is avoided.

An example of true positive is in the *chaincode_union_loan* contract in Figure 8.2, a proof of concept implementation of bank loans in the blockchain. Users call the method `offer` to offer a loan. GoLiSA detects a flow that leads to an untrusted cross-contract invocation on tainted data about loan participants. Namely, at Line 3 of the method `Invoke`, GoLiSA considers `GetFunctionAndParameters` as a tainted source since it yields a function name and arguments provided as part of the transaction request, hence under user control. This tainted data propagates, through `args`, to method `offer` at Line 5, later to `chainCodeToCall` at Line 12, until it reaches `InvokeChaincode` at Line 21, where GoLiSA issues a warning. Good practice avoids this real issue by hardcoding `chainCodeToCall` at Line 21, or at least by checking it against a constant list of allowed targets. However, note that in the case of allowlist, the untrusted flow would remain, but it would be harmless, leading the analyzer to give a false alarm.

Experimental Results of MichelsonLiSA

Table 8.4 shows the results of the cross-contract invocation analysis checker over $\mathbb{TZ}$. MichelsonLiSA works for 1000 smart contracts (100% of $\mathbb{TZ}$) by using *Taint*. Analysis time is a few seconds per chaincode, on average. In total, the checker issues 3063 warnings, distributed in 843 smart contracts. Unlike benchmark $\mathbb{HF}$, in this case, it was not possible to carry out a deep investigation of the warnings by dividing them into categories (#TP, #FP). At the end of the analysis, MichelsonLiSA can provide additional reports containing the analyzed CFGs in various formats (html, dot, etc.) with details about the computed abstract states. This allows one to check, for each program point, which variables the analysis infers as tainted and which it does not. However, for a deep manual investigation capable of identifying any over-approximations and false positives, one should manually recompute the entire execution stack for every single instruction and check if its execution in the real world can lead to a tainted value or not compared to the MichelsonLiSA report. This activity is time-consuming given the poor readability of Michelson, the difficult to reverse engineer a program because high-level information is lost after compilation (Section 3.4.2), and the complexity of some contracts. For this reason, we could not manually investigate each of these files and compute the rate of true and false positives.

```
parameter address ;
storage unit ;
code {
  DUP ;
  CDR ;
  SWAP ;
  CAR ;
  DUP ;
  NIL operation ;
  SWAP ;
  CONTRACT unit ;
  { IF_NONE { PUSH unit Unit ;
      FAILWITH } {} } ;
  AMOUNT ;
  PUSH unit Unit ;
  TRANSFER_TOKENS ;
  CONS ;
  SWAP ;
  DROP ;
  PAIR }
}
```

(a) Michelson smart contract

```
v0 = parameter_storage();
v1 = DUP(v0);
v2 = CDR(v1);
SWAP();
v3 = CAR(v0);
v4 = DUP(v3);
v5 = NIL();
SWAP();
v6 = CONTRACT(v4);
IF v7 = extract_value(v6) {
  v8 = PUSH("Unit");
  FAILWITH();
}
v9 = AMOUNT();
v10 = PUSH("Unit");
v11 = TRANSFER_TOKENS(v10,v9,v7);
v12 = CONS(v11, v5);
SWAP();
DROP();
v13 = PAIR(v12, v2);
```

(b) Michelson IR in SSA form

Figure 8.3: Smart contract *expruqYPRHnQyNih8sK1vhNLRBLx37VeuZ3T58SWax-Pj5WwbCQJb2V.tz.*

Figure 8.3 shows a smart contract of 𝕋ℤ affected by the UCCI issue. MichelsonLiSA detects a flow that leads to an UCCI. It begins from `v0 = parameter_storage()`, the information is propagated into `v3 = CAR(v0)`, then `v4 = DUP(v3)`. At this point, the untrusted information flows in `v6 = CONTRACT(v4)` as the contract address, making the target contract untrusted. Finally, the contract is propagated into `v7 = extract_value(v6)`, to after flow in `TRANSFER_TOKENS(v10,v9,v7)`, where the analysis detects that an untrusted contract is invoked to transfer, without any check, the amount of currency loaded by `v9 = AMOUNT()`.

```
parameter unit ;
storage unit ;
code {
  CDR ;
  NIL operation ;
  PUSH address "tz1RwoEdg4efDQ...
      UYvg278Gv1ir" ;
  CONTRACT unit ;
  IF_NONE { FAILWITH }
  { BALANCE ;
  UNIT ;
  TRANSFER_TOKENS ;
  CONS ;
  PAIR }
}
```

(a) Michelson smart contract

```
v0 = parameter_storage();
v1 = CDR(v0);
v2 = NIL();
v3 = PUSH("tz1RwoEdg4efDQ...
    UYvg278Gv1ir");
v4 = CONTRACT(v3);
IF v5 = extract_value(v4) {
  FAILWITH();
} ELSE {
  v6 = BALANCE();
  v7 = UNIT();
  v8 = TRANSFER_TOKENS(v7,v6,v5);
  v9 = CONS(v8,v2);
  v10 = PAIR(v9,v1);
}
```

(b) Michelson IR in SSA form

Figure 8.4: Smart contract exprthPm93Nt4TBdDSd9LVG829YcgbK9VKE4TRDXt-ZiU8Fv7gFEBod.tz

Instead, Figure 8.4 shows a smart contract of 𝕋ℤ with a safe token transfer. MichelsonLiSA does not detect any untrusted flow that leads to an UCCI. The analysis starts by propagating the parameter and storage inputs in `v0 = parameter_storage()`. The untrusted value of `v0` is used only after the `TRANSFER_TOKENS` (sink for the analysis), then it cannot affect the cross-contract invocation. Indeed, this sink targets only a contract derived by the hardcoded address declared in `v3 = PUSH("tz1RwoEdg4efDQ...UYvg278Gv1ir")`, therefore the address cannot be changed by any arbitrary input ensuring its safety from UCCI.

8.5 Conclusions

The issues related to UCCI have dangerous consequences in blockchain software. It is important to predict by design the possible interactions of contracts once deployed, to use appropriate evolution standards such as EIP-2535 [139] or EIP-1822 [80], and to apply verification tools to prove the absence of unwanted behaviors. In this chapter, we proposed an analysis for the detection of the UCCI exploiting taint analysis. Furthermore, as far as we know, these are the first implementations based on information flow analysis for UCCI detection available for those languages.

The next chapter will discuss abstract interpretation techniques different from information analysis. Abstract domains will be used to detect numerical problems.

Chapter 9

NUMERICAL ISSUES DETECTION

In mathematical theory, there are infinite numbers between $-\infty$ and $+\infty$. However, computers have a finite memory capacity, and there is a limit to the numbers that can be represented. When this limit is breached a condition called *numerical overflow/underflow* occurs. This chapter deals with an unpublished work in progress which as a first purpose has to empirically demonstrate the applicability of non-trivial abstract numerical domains for the verification of numerical issues, using *numerical overflow/underflow* as a case study.

9.1 Numerical Overflow/Underflow

When an arithmetic operation attempts to create a numeric value that exceeds the bound that can be represented with a given numerical type, it can lead to unintended behaviors. If the numerical value exceeds the maximum value a *numerical overflow* occurs; if it exceeds the minimum value a *numerical underflow* occurs.

These behaviors can sometimes be exploited in a harmful way by programs to revert to an initial state or recalibrate, such as for timers and clocks. However, in general, overflows and underflows may have critical consequences in blockchain. For instance, an attacker can exploit it by repeatedly invoking a smart contract function with a numerical overflow bug that increases a value, to drain more money than it should as in the case of EOSFomo 3D's [135, 136] and the Ethereum ERC-20 token used in the Beauty Chain economic system [54].

```
1     var c int8 = math.MaxInt8 - 3
2     for i := int8(0); i < 10; i++ {
3       c = c + i
4     }
```

Figure 9.1: Go fragment affected by integer overflow.

These issues affect both DSLs and GPLs. However, the program behavior can be different when these issues occur. For instance, Michelson triggers by default a run-time error overflow/underflow. Instead, Go does not raise any run-time error and just lets the execution proceed [86].

In the Go fragment of Figure 9.1, variable `c` defined at Line 1 has type `int8`, hence it can contain integers from -128 to 127, and it is initialized to 124. Go adopts a *wrap-around semantics*, meaning that if a certain integer value exceeds its largest possible value (specified by its type), it continues from its smallest possible one. Thus, at Line 3, when `i` holds 3, the variable `c` *overflows*. The current value of `c` is 127, the addition of 3 exceeds the maximum value for `int8`, and the value assigned to `c` is, unexpectedly, -125.

This default behavior of Go is critical for blockchain software and also affects all the blockchain frameworks discussed in Section 5.1. Below, we propose several approaches to detect these issues using GoLiSA.

For sake of simplicity, we do not provide analyses for MichelsonLiSA because, when an overflow happens, Michelson mitigates the problem triggering run-time errors and thus blocking fraudulent transactions. However, the same analyses can be implemented in MichelsonLiSA.

9.2 Detection by Numerical Abstractions

Syntactic checks cannot help in detecting overflow problems because they involve program semantics. Hence, abstractions are needed. There are many studies related to numerical abstractions [37, 40, 47, 49, 88, 89, 138, 179]. According to Miné [110], they vary in expressiveness and in the performance/precision trade-off. However, some abstract domains have no publicly available implementation while others lack operators to be useful in more general settings. Below, we will describe the most popular abstractions.

Interval Domain. The Interval abstract domain is one of the first numerical domains proposed in abstract interpretation. The idea of the interval domain [47] is to abstract a set of numerical values by using the least single interval enclosing them. For instance, a set $\mathbb{S}$ of integers is approximated with the interval $[a, b]$, where a is the minimum value in the set $\mathbb{S}$, and b is the maximum value in the set $\mathbb{S}$. However, it is not always possible to compute the upper/lower limit of a numerical set, thus it is necessary that a may be $-\infty$ and b may be $+\infty$. According to Cousot [47], this domain is a lattice then we can define an ordering relation $\sqsubseteq$ such that $[a, b] \sqsubseteq [c, d] \iff a \geq c \wedge b \leq d$, i.e. the whole interval $[a, b]$ is contained in $[c, d]$. The bottom element $\perp$ corresponds to an empty interval without any value. The top element $\top$ corresponds to the interval $(-\infty, +\infty)$, i.e. the widest in which all the other intervals can be contained. The main drawback is related to precision. For instance, consider the set $\{-1000, 1000\}$, that it is composed of two values. This set will be approximated by the interval $[-1000, 1000]$, thus undergoing a large over-approximation involving thousands of values. Therefore, this excessive approximation will potentially lead to the generation of numerous false positive alarms.

Polyhedron Domain. The abstract polyhedron domain [49] denotes abstractions based on polyhedra [173, 174]. The elements of this domain are the $\perp$ element that corresponds to an empty set $\emptyset$ and the conjunctions of linear inequality constraints over the program variables to constraint sets of memory program states. This domain guarantees a high level of analysis precision even if it features no best abstraction function, although certain concrete sets do have best abstraction [169, Chapt. 3.2]. The main drawback is that when the number of variables increases the domain has a significant memory cost because its complexity is exponential in the number of variables [138].

Octagon Domain. According to Miné [138], the purpose of the octagon domain is to be a numerical abstract domain that, in terms of expressiveness and cost, stage between the interval and the polyhedron domains. This domain allows one to manipulate invariants of the form $(\pm x \pm y \leq c)$, where x and y being numerical variables and c a numeric constant. These invariants are a special kind of polyhedra called *octagons* because they feature at most eight edges in dimension 2 (Figure 9.2c). The elements of this domain are the $\perp$ element that corresponds to an empty set and the conjunctions of linear inequality constraints of the form $\pm x \pm y \leq c$ or $\pm x \leq c$.

Performance vs Precision

While the interval domain allows one to track the approximation of a variable in isolation, octagon and polyhedron domains track the relation between program variables. For this reason, the interval domain is a *non-relational analysis*, while octagon and polyhedron domains are *relational analyses*. There exists a precision relation between the aforementioned analyses: polyhedron *is more precise than* octagon *that is more precise than* interval. A graphical representation is shown

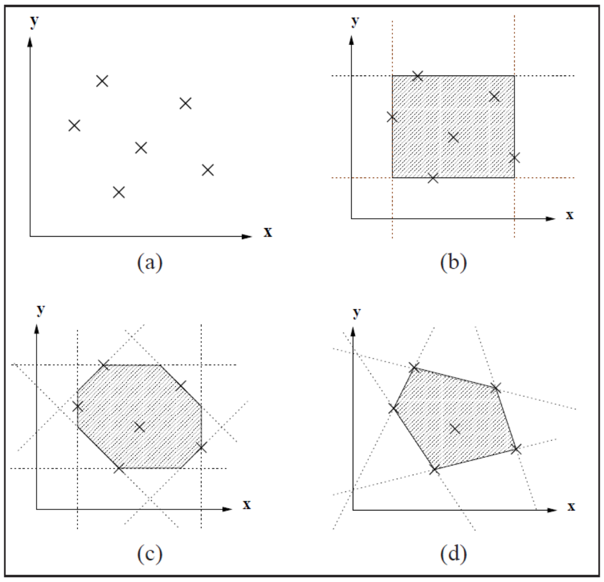


Figure 9.2: A set of points (a), and its best approximation in the interval (b), octagon (c), and polyhedron (d) abstract domains [138].

in Figure 9.2. As typical in static analysis, more precise means also less efficient, as our experiments confirm (Section 9.4).

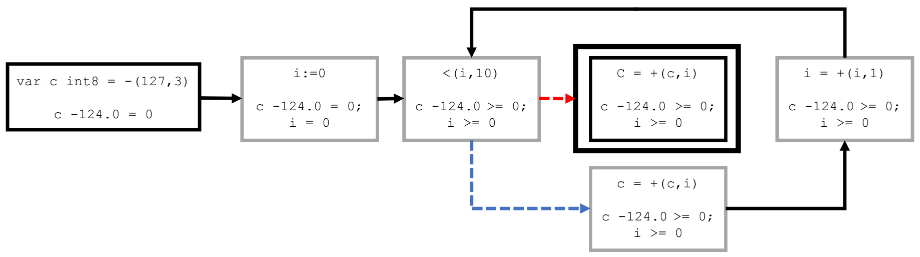
9.2.1 Implementation in GoLiSA

Some numerical abstractions require a lot of implementation effort. As GoLiSA is extensible, it can be configured to use additional frameworks which support them, such as Apron [110].

Apron is a library offering a suite of advanced numerical abstractions to implement sound analyses:

- Box: using interval domain at a specific program point, it approximates each numerical variable `x` as the interval of numerical values (e.g., $[1, 7], [0, +\infty]$) that `x` may contain;
- Octagon: at a specific program point, it approximates numerical variables by using the octagon domain as a conjunction of constraints of the form $\pm \mathtt{x} \pm \mathtt{y} \leq c$, where `x` and `y` are numerical variables, and c is a numerical constant;
- ConvPoly: at a specific program point, it approximates numerical variables by using the polyhedra domain as a conjunction of linear inequalities of the form $c_0\mathtt{x}_0 \pm c_1\mathtt{x}_1 \cdots \pm c_n\mathtt{x}_n \leq c$, where $\forall i \in [0, n]$ $\mathtt{x}_i$ is a numerical variable and c_i is a numerical constant. Apron provides two different implementations for convex polyhedra. We used the one built on the Parma Polyhedra Library [16].

a)



b)

```
['example.go':4] on
'LiSAProgram::main'
          the variables c and i may overflow.
```

Figure 9.3: (a) CFG with interval analysis result of Figure 9.1. (b) Overflow checker report file for the exit node.

In GoLiSA, to detect numerical issues, we have adopted in semantic checkers the abstractions provided by Apron. If a warning about the overflow of a certain variable is raised by the checker, the overflow *may* not occur, but, conversely, if it is not raised, the overflow *surely* does not occur. Given a numerical variable `x` at a certain program point, let us denote by $max_{type}(\texttt{x})$ and $min_{type}(\texttt{x})$ the maximum and minimum value for the type of `x`.

$$of^+(x) \triangleq x > max_{type}(x) \qquad \text{(positive overflow check)}$$
$$of^-(x) \triangleq x < min_{type}(x) \qquad \text{(negative overflow check)}$$

These checks are performed for each program point to detect if an overflow occurs there. Given a certain program point and a numerical variable x, the warning generation may lead to one of the following results: (i) a *definite alarm* is raised if the analysis infers that surely an overflow occurs at the given program point for the variable x; (ii) a *possible alarm* is raised if the analysis infers that an overflow may occur at the given program point for x. Typically this happens when the analysis goes to $\top$; (iii) no alarms meaning that no other cases can occur since, as we have already mentioned before, the analyses provided by Apron are sound.

In GoLiSA, the numerical overflow checkers are supported by the abstract domains Box, Octagon, and ConvPoly. For instance, let us consider the Go fragment reported in Figure 9.1. If we run GoLiSA with the interval analysis and the semantic checker described above, we obtain the CFG reported in Figure 9.3a, where each node is labeled with the interval analysis results (expressed as a set of linear inequalities). Figure 9.3b reports the warning raised by the checker for the return node. Note that the warning raised for the variable `i` is a false alarm, since it does not overflow, while the one concerning the variable `c` is a true alarm.

9.3 Related Work

Numerical issues have plagued smart contracts on blockchain since the dawn of their implementation, as in the case of Ethereum. Although the problem was known, for several years the fixes developed in Solidity language, the most popular for Ethereum, were left to the discretion of developers. During this period, third-party libraries such as SafeMath [14, Chapt. 9] have been involved to solve these problems. However, this was not enough to reduce the risk of developing and deploying buggy contracts. Only since v.0.8.0 [68], released five years after the first version of Solidity, the language has taken official countermeasures. This release allowed arithmetic operations to be reset by default when a numeric overflow/underflow occurs. This was possible by applying low-level checks when compiling arithmetic operations from the Solidity language to Ethereum bytecode. This default behavior allowed developers to deny untrusted executions and increase code readability, even if it led to increased gas costs (more bytecode statements are executed). However, the problem remains that previous versions and in different languages at a low level compile vulnerable Ethereum bytecodes.

As for numerical analyses through abstract interpretation in the blockchain context, there are only a few uses. The Securify [195] analyzer extends the ELINA library [81] to perform analyses on Ethereum by using abstract domains and by checking numerical properties, including overflows. In [20], Guillaume et al. suggest the possibility of using the MOPSA [137] analyzer and Apron [110] to investigate numerical properties on the Michelson language. The ZEUS [112] analyzer allows one to perform the verification of smart contracts written for Ethereum and Hyperledger Fabric, by converting source code into LLVM bytecode and by exploiting abstract interpretation and symbolic models to detect vulnerabilities. However, in [112], there is no detail about the abstractions used for numerical issues detection.

For other approaches, HFContractFuzzer [62] applies fuzzing techniques using go-fuzz [87] to detect integer overflow on Go chaincodes. Lai et al. [120] propose a static detection tool inspired by SmartCheck [192] and based on XPath patterns for integer overflow of Solidity smart contracts in Ethereum. However, this pattern-matching approach does not give formal guarantees, so it is subject to true positives and false negatives.

9.4 Experimental Evaluation

Table 9.1 reports the results of the experimental evaluation of the semantic module of GoLiSA, over the benchmark $\mathbb{HF}$. In particular, GoLiSA is able to semantically analyze up to 895 chaincodes of the benchmarks, namely 93.03% of

Analysis	$\mathbb{HF}$ **Coverage**	**#W**	**#DW**	**#PW**	**AT**	**Timeout** (over 10 min)
Box	93.03%	12408	1505	10903	24.84 sec	4 chaincodes
Octagon	92.93%	12398	1541	10857	22.75 sec	5 chaincodes
ConvPoly	82.23%	10156	1223	8933	87.13 sec	51 chaincodes

Table 9.1: Results of overflow checker.

Analysis	**#W**	**#DW**	**#PW**
Box	10157	1212	8945
Octagon	10151	1212	8939
ConvPoly	10146	1213	8933

Table 9.2: Numerical comparison between common results.

them. Table 9.1 reports the evaluation for each numerical analysis implemented in GoLiSA, namely Box, Octagon, and ConvPoly, where **#W** is the total number of warnings issued by the analysis,**#DW** is the number of definite warnings among the raised warnings and **#PW** is the number of possible warnings among the raised warnings, and **AT** is the average execution time on each file.

Instead, Table 9.2 shows a numerical warning comparison, on the 791 chaincodes that are the intersection where all three domains work. The high number of warnings is due to chains of mathematical operations sharing the same variables. If one of them seems to overflow, GoLiSA will reach the same conclusion for the subsequent operations as well, which results in a sequence of alarms. This will require GoLiSA to focus on more likely warnings in the future.

The evaluation does not highlight a big difference between interval and octagon-based analyses. In terms of execution time, Octagon takes a little longer because it computes relations between components to increase precision. However, the computed relations are not sufficient in this case to improve the results by discarding false positives. Let us consider instead the ConvPoly-based analysis. Even if the number of smart contracts potentially affected by overflow vulnerabilities is almost unaltered, the total number of potential warnings is decreased in comparison to the previous analyses. Being ConvPoly more precise than Octagon and Box. Indeed, the ConvPoly analysis triggers fewer warnings than others, and missing ones correspond to false positives. ConvPoly is considered one of the most precise numerical analyses, but the performances in terms of time and space could become exponential in the worst-case scenario. However, smart contracts allow one to scale with ConvPoly, because in general they are simple (reduced number of LoC) in comparison to industrial software (more than 100K LoCs). As expected, ConvPoly leads to an increment of the execution time, which is clearly higher than that of Box and Octagon. Nevertheless, the execution time on $\mathbb{HF}$ is still reasonable and benchmark coverage remains high,

```
1  func (t *Transfer) invoke(
2  stub shim.ChaincodeStubInterface,
       args []string) pb.Response {
3    var A, B string
4    var Aval, Bval, X int
5    /* [....] */
6    Avalbytes, err := stub.GetState(A)
7    /* [....] */
8    Aval, _ = strconv.Atoi(string(
         Avalbytes))
9
10   Bvalbytes, err := stub.GetState(B)
11   /* [....] */
12   Bval, _ = strconv.Atoi(string(
         Bvalbytes))
13   X, err = strconv.Atoi(args[2])
14   /* [....] */
15   Aval = Aval - X
16   Bval = Bval + X
17   err = stub.PutState(A,
18   []byte(strconv.Itoa(Aval)))
19   err = stub.PutState(B,
20   []byte(strconv.Itoa(Bval)))
21   /* [....] */
22   return shim.Success(nil)
23 }
```

(a)

```
Aval = Aval - X
if Bval > 0 &&  X > 0 && Bval + X >
    Bval {
  Bval = Bval + X
} else {
  return shim.Error(err.Error())
}
```

(b)

Figure 9.4: (a) A simplified chaincode of invoke. (b) Patch for Lines 15–16.

with an average per smart contract under two minutes and ℍ𝔽 coverage over 80%, respectively.

We highlight that the trade-off between precision and performance is justified: GoLiSA can detect the aforementioned vulnerabilities since its checkers are based on an underlying semantic analysis (that would be impossible by just analyzing the source code), as witnessed also by the case study reported below.

9.4.1 Case Study

We conclude this section by discussing the overflow checkers implemented in GoLiSA over a case study taken from ℍ𝔽. In particular, we discuss a banking solution[1], that allows banks and users to conduct cross border transactions in real-time without the need of central authorities. In particular, its smart contract contains the method `invoke` to make a payment of `X` units from the account `A` to the account `B`. As shown in Figure 9.4a, the balances of `A` and `B` are stored in the variables `Aval` and `Bval` at Lines 6 and 10, respectively.

[1] https://github.com/deenario/Banking-System-Blockchain/blob/master/fabric/chaincode/chaincode_example02/go/chaincode_example02.go

The requested transfer amount is then stored in the variable `X` at Line 13, subtracted from `Aval` at Line 15, and added to `Bval` at Line 16. Finally, the state is updated with the value of `Aval` and `Bval` at Lines 17 and 19, respectively. Note that the mathematical operations at Lines 15–16 are executed without any preliminary checks and transactions with a very large value of `X` may get declined by the application due to endorsement policy failure. Thus, this smart contract is vulnerable to numerical overflow. Therefore, faulty smart contracts badly affect the performance of the application and in the worst case can cause huge losses for the organizations. Unlike traditional software where patches are built to mitigate the detected errors, smart contracts cannot be easily patched due to the immutability feature of blockchain, and consensus algorithms require the consent of all concerned peers.

Let us focus on Lines 15–16. Running GoLiSA with any of the numerical analyses among Box, Octagon, and ConvPoly raises warnings at Lines 15–16, concerning the variables `Aval` and `Bval`, respectively, since the overflows may arise depending on the values `Avalbytes` and `Bvalbytes` read from the state. Nevertheless, let us consider a slightly different case in order to show the difference between Box (that is a non-relational analysis) and Octagon and ConvPoly (that are relational analyses). Let us suppose to patch Lines 15–16 with the fragment reported in Figure 9.4b. In this case, the overflow may still occur for variable `Aval`, while for `Bval`, at Line 2, there is a simple overflow protection pattern that protects the variable from overflow: if `Bval` and `X` are positive values and $\texttt{Bval} + \texttt{X}$ does not exceed `math.MaxInt64`, the sum is safe and `Bval` is correctly updated, otherwise, an error is returned. If we analyze the function with Box, the semantic checker still raises an overflow warning for `Bval`, which, in this case, is a false alarm. This happens because Box is not precise enough to track that $\texttt{Bval} + \texttt{X}$ does not exceed `math.MaxInt64`. Conversely, if we analyze the function with Octagon or ConvPoly, the analyses track that $\texttt{Bval} + \texttt{X} - \texttt{math.MaxInt64} < 0$, and consequently the semantic checker does not raise any alarm when `Bval` is updated.

9.5 Conclusions

In this chapter, we provided several abstract domains used to prove numerical properties such as *overflow/underflow* in smart contracts written in Go. Although the analysis produces a considerable amount of warnings, the experimental evaluation has empirically demonstrated the applicability of non-trivial domains such as those based on polyhedra, which, due to the large consumption of resources and time, preclude their usability in other software contexts. To the best of our knowledge, this is the first evaluation for Go smart contracts of abstract numerical domains based on abstract interpretation. This is intended to be a starting point for any analyses that increase the accuracy of the results towards better verification of numerical issues related to blockchain software.

This chapter concludes the topic of analyses and tools based on abstract interpretation. Next, the last chapter about the verification part describes an *on-chain* architecture capable of performing verification directly within the blockchain.

Chapter 10

ON-CHAIN VERIFICATION OF SMART CONTRACTS

In this chapter, we define the *on-chain* code verification paradigm, i.e. an approach which involves the blockchain to verify the code being deployed. The chapter describes an implementation of a blockchain with on-chain verification, built as a Tendermint application that runs smart contracts written in the Takamaka framework. Currently, the implementation includes 26 on-chain checks, that mostly verify the correct use of Takamaka's primitives and code annotations and the use of a deterministic subset of Java. To the best of our knowledge, this is the first paradigm for blockchain verification ables to verify and to guarantees that all code executed in the blockchain has been successfully verified over time. On-chain verification architecture has been developed before GoLiSA and MichelsonLiSA, and has thus been experimented using a custom Java static analyzer for Takamaka framework. The architecture is however independent from the analyzers and the smart contract framework used. Therefore, it can also be used to perform the analyzes proposed in the previous chapters. Some contents of this chapter are also published in [147].

10.1 On-Chain Code Verification

Given the immutability of data in the blockchain, it is of the utmost importance to apply software verification in the development process to ensure the security and quality of the code, in order to avoid the presence of immutable bugs that can be exploited with malicious intent. In addition to those already treated, not surprisingly, Turing-completeness for smart contracts introduces the risk of all sorts of bugs [15, 161, 205]. Since smart contracts deal with money and cannot be replaced, it is paramount to deploy only *correct* code in the blockchain. For this reason, many analyzers are involved in vulnerability and bug detection. Furthermore, there are companies that provide code audit services, using both automatic tools and human investigation. A limit of these approaches is that they are *optional* and *external* to the blockchain (hence *off-chain*), i.e. programmers are not forced to use them and thus do not actively protect themselves against the deployment of bugged or dangerous code. Moreover, traditional software paradigms, such as Software Development Life Cycle (SDLC), fail to guarantee the blockchain development requirements [127].

We propose a general architecture able to perform a *mandatory* code verification directly on the blockchain, i.e. *on-chain*, and to approve the analysis results through the consensus mechanism.

10.1.1 The Architecture over Tendermint

This section describes the architecture of a blockchain node with on-chain code verification, built on Tendermint. The choice of the Tendermint platform is due to the ease of implementing a blockchain from scratch and because it separates the logic of the application layer from the rest. Nevertheless, the architecture is platform-independent and can also be deployed on other blockchains.

Tendermint's Node Structure

Each node of a blockchain based on Tendermint is structured on three layers:

– **Networking**: discovers and connects nodes with each other, propagates requests for transactions, and collects their responses from other nodes.
– **Consensus**: compares and approves/rejects the responses obtained by executing the requests on the nodes.
– **Application**: specifies which requests are valid, how their responses are computed, and how the application's state consequently evolves.

The implementation of networking and consensus, without any application layer (its distribution includes a few toy applications, irrelevant for our purposes) is provided by the component *Tendermint Core*. Programmers develop their own application layer and plug it into Tendermint Core via its Application BlockChain Interface (ABCI). Tendermint Core replicates the application state on each machine of the network.

Figure 10.1 shows a detailed picture of Tendermint Core and of an application connected through its ABCI. It shows that Tendermint Core keeps the blocks of the blockchain in its own database, that need not be the same used to hold the application's *state*. For instance, the latter holds the smart contracts code

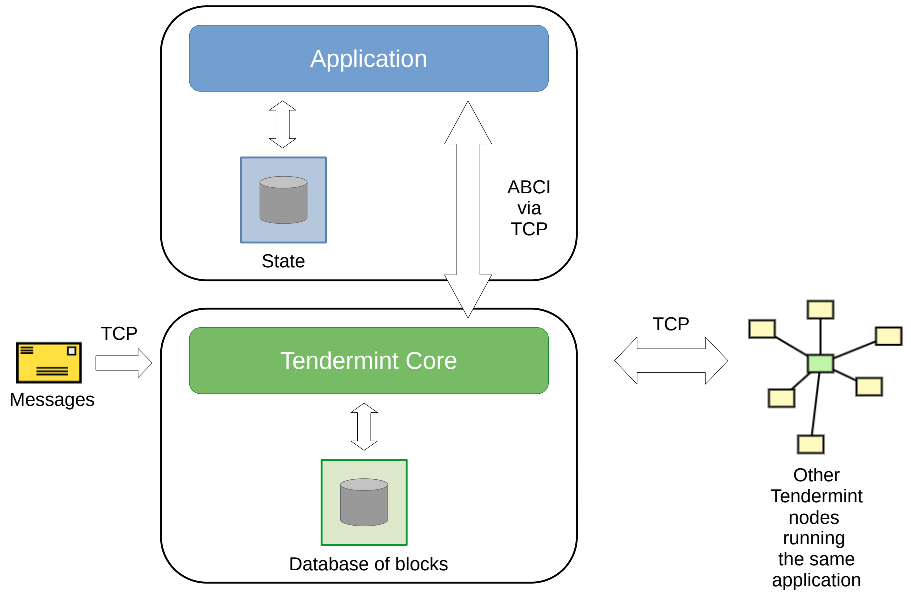


Figure 10.1: Tendermint Core and a Tendermint application, with their respective databases.

installed in the blockchain and the value of their state variables. Tendermint Core needs only the hash of the application state, for consensus, to ensure that all nodes have reached the same application state.

One can define the application state as a map σ from the hash of the requests that the blockchain has executed to the responses that have been computed for them. The application state contains the full responses, but only the hash of the requests. Hence, it can be implemented as a Merkle-Patricia trie. The full requests are contained in the database of blocks of Tendermint Core instead since they are needed to replay the transactions in all nodes of the network.

Code Verification over Tendermint

The application level is responsible for managing transaction requests and custom logic. In the case of smart-contract support, it contains a framework to execute the contract code. Hence, this layer will also include code verification (Figure 10.2). Indeed, the verification must check the code from requests before code deployment or code execution.

Assume that a *request*, whose hash is $request_h$, reaches the blockchain, requiring to install, in blockchain, the code of some smart contracts, reported inside *request*. Figure 10.3 shows the sequence diagram for the execution of

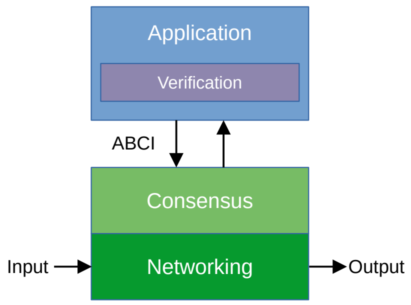


Figure 10.2: High-level architecture of an application running on Tendermint Core and performing on-chain verification.

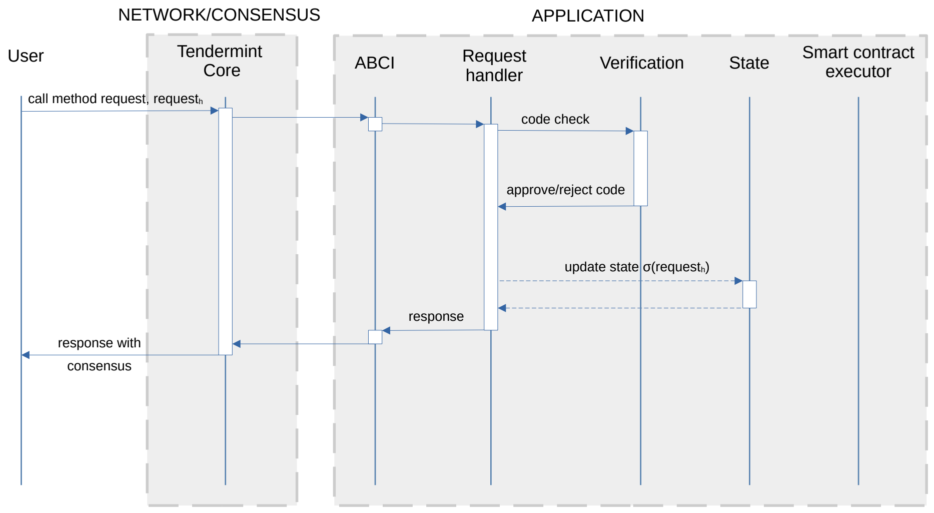


Figure 10.3: Sequence diagram for code verification and installation in the blockchain.

request. Namely, Tendermint Core routes *request* through networking and consensus up to the application, that uses its verification module to either approve or reject the code. If approved, the application includes the code in a *response* and updates its state σ with a new binding: $\sigma(request_h) = response$. The hash $request_h$ is an immutable, machine-independent reference to this code, used later to instantiate and execute smart contracts. If the code is rejected, instead, the application state is expanded with a failure response, that does not contain any code.

Figure 10.4 reports an example of application state evolution. It reports the requests in full, for readability, but remember that only the hash of the

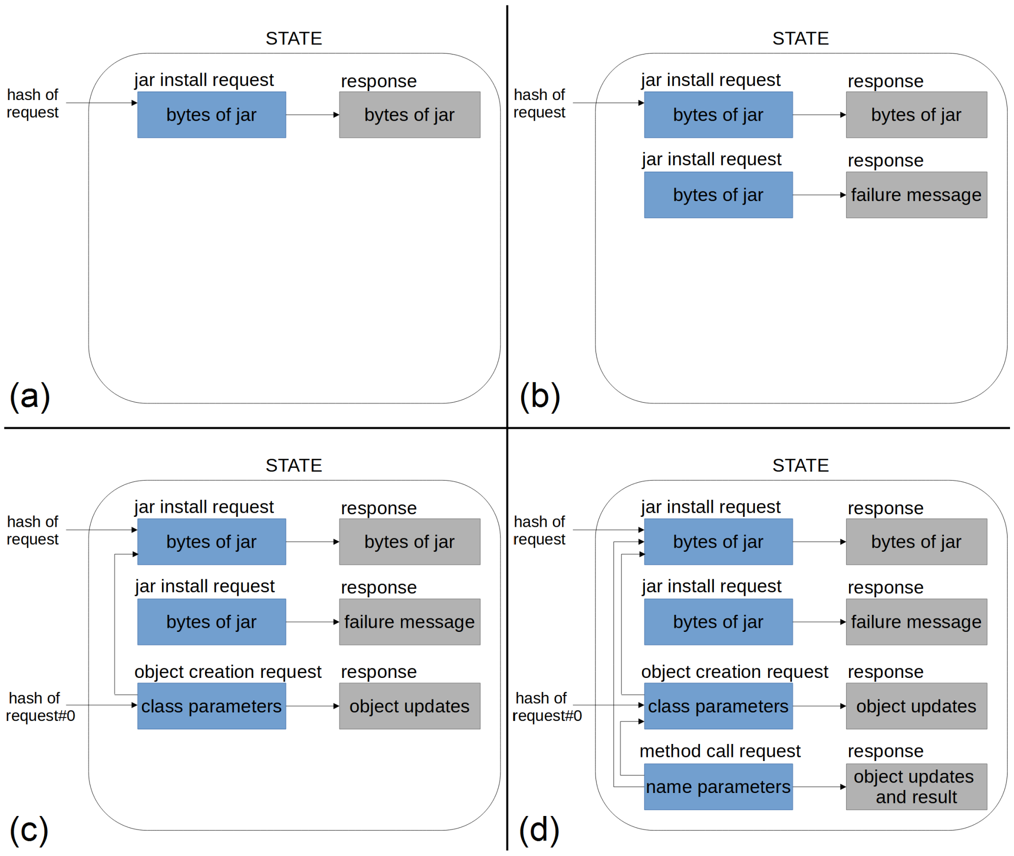


Figure 10.4: The evolution of the application state during a sequence of requests.

requests is kept in the application state. Figure 10.4a shows the application state after the execution of a code installation request for which verification succeeds. The code is Java bytecode, packaged into a *jar*, *i.e.*, a zipped container of Java bytecode. The response contains the same jar (*i.e.*, the same code as the request[1]). In terms of Java, the hash of the request is the *classpath* of subsequent code executions. Figure 10.4b reports, instead, a request whose code fails to verify. The response does not include any code installed in the blockchain. This shows that the verification rules are part of the consensus rules that determine which code installation request is valid and which must be rejected instead (Figure 10.4a and 10.4b). Hence they must be the same in every node of the network and must be deterministic.

[1] The response might also contain instrumentation of the code, as it is the case for the Java subset for programming smart contracts called Takamaka, which we use in our implementation. This is irrelevant here and we refer the interested reader to [188].

On-chain verification performs code verification statically, only once, when the code is installed in the blockchain. For instance, Figure 10.4c shows a subsequent request that asks to instantiate a smart contract whose code has been installed by the request in Figure 10.4a. The request in Figure 10.4c uses the hash of the request in Figure 10.4a as its classpath and contains the parameters for calling the constructor of the smart contract. The execution of the request runs that constructor, without code verification: it has been already performed in Figure 10.4a. The immutable reference *hash of request#0* is used later to refer to the new smart contract instance[2]. The state of the new smart contract is reported in the response as a set of *updates*, that is, instance fields modified during the execution of the request, including those of the smart contract instance *hash of request#0* that has been created in blockchain. Finally, Figure 10.4d shows the execution of a request asking to call a method on the instance of smart contract *hash of request#0*. This last request refers to both the classpath and the target instance smart contract. Its execution, in general, modifies some instance fields of objects in the blockchain, that are reported as *updates* in its response. This last request does not verify the code either, since it is not a code installation request.

The rules of on-chain verification are part of the consensus rules of the blockchain since they determine if the response of a request to install code in the blockchain is successful or failed. Hence, they determine the evolution of the state of the application layer and its hash, which is reported in the blocks of the underlying Tendermint blockchain, that uses it for consensus. This is the standard way of working for Tendermint. Hence, all nodes must use the same verification rules. Nodes that use different rules will be automatically excluded from the Tendermint blockchain.

10.2 Implementation

We have implemented on-chain verification for smart contracts written in the Takamaka subset of Java [188] (the lazy re-verification technique of Section 10.4 is still under development and we leave it for future work). The goal of Takamaka is to write smart contracts in a well-known programming language, leveraging expertise and existing mature development tools. The application layer of Takamaka is a state machine (the Tendermint *application* in Figure 10.1) that executes transactions from request to response. Requests can specify the addition of a jar in the permanent state of the application, or the execution of a

[2] The index #0 refers to the first object created during the execution of a request. In general, a request can instantiate many objects, depending on the code that it executes. For simplicity, this example assumes that only one has been instantiated.

constructor, or of an instance or static method of code previously installed in the state. Responses include the effects of the transaction, as a set of field updates (see Figure 10.4). Updates can be computed since the jar of the Java code is instrumented before being installed in the blockchain, with extra code that keeps track of the affected fields of objects [188]. Determinism is ensured since only a deterministic subset of Java is allowed, restricted to a deterministic API of the Java library [187]. The state machine of Takamaka is implemented in Java and runs on a standard Java virtual machine. The state is kept in a Merkle-Patricia trie that implements a map from the hash of requests to their corresponding response (Figure 10.4). This trie is kept in the Xodus transactional database by JetBrains[3].

The verification module is implemented as a sequence of *checks* performed on methods and classes. Since the request of installing new code in the blockchain contains the compiled bytecode only, such checks run at Java bytecode level, by using the BCEL library for Java bytecode manipulation[4]. The source code is simply not available in the blockchain. Currently, Takamaka's on-chain verification performs 26 checks on every jar that gets installed in the blockchain. They must all pass, or otherwise, the jar will be rejected. Figure 10.5 describes some of them.

Next, we show a specific example of a check. It verifies that method `caller()` is used in the right context. That method corresponds to `msg.sender` in Solidity: it allows programmers to get a reference to the *contract* that calls a method or constructor X.

The method `caller()` can be used inside the code of X only if X satisfies two constraints[5]:

1. X is annotated as `@FromContract(class)`, for some `class`;
2. the invocation of `caller()` occurs on `this`.

The rationale of Constraint 1 is that `@FromContract(class)` guarantees that X can *only* be called from a contract of type `class`, or subclass, or from an external wallet whose paying account has type `class`, or subclass. Hence the caller exists. For instance, the following contract stores its creator in field `owner`. The use of `caller()` is correct here, since it occurs inside a `@FromContract` constructor:

```java
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;

public class C1 extends Contract {
  private C1 owner;
```

[3] https://github.com/JetBrains/xodus
[4] https://commons.apache.org/proper/commons-bcel
[5] `@FromContract` and, later, `@Payable` are Java *annotations*, that is, a mechanism for adding metadata information to source and compiled code. They are irrelevant to the code executor, but can be used by code analysis and instrumentation tools.

Correct context for `@FromContract`	`@FromContract` is only applied to instance methods or to constructors of storage classes (*i.e.*, classes whose instances can be kept in the blockchain).
Correct calls to `@FromContract`	`@FromContract` methods or constructors are only called from instance methods or constructors of contracts.
Correct context for `@Payable`	`@Payable` is only applied to `@FromContract` methods or constructors of contracts (since only contracts have a balance).
Correct fields in storage classes	Classes whose instances can be kept in blockchain can only have a restricted set of types for their fields.
Correct context for `caller()`	See the description in this work.
No finalizers	Since their execution is non-deterministic in Java.
Only white-listed Java APIs	To enforce determinism (see [187]).

Figure 10.5: Some of the 26 on-chain verifications currently performed by Takamaka.

```
    public @FromContract(C1.class) C1() {
      owner = (C1) caller(); // ok
    }
  }
```

Instead, it is incorrect to invoke `caller()` in a method or constructor not annotated as `@FromContract`, since its caller is not necessarily a contract and `caller()` would be meaningless in that case:

```
  import io.takamaka.code.lang.Contract;

  public class C2 extends Contract {
    public void m() {
        /* [....] */ = caller(); // error at deployment time
      }
    }
```

The reason of Constraint 2 is that its violation lets one access the caller of other contracts, with possible logical inconsistencies and security issues. For the same reason, the use of `tx.origin` is normally an antipattern in Solidity (see *Tx.origin Authentication* in [14]). Constraint 2 holds in classes `C1` and `C2` above, but is violated below:

```
    import io.takamaka.code.lang.Contract;
    import io.takamaka.code.lang.FromContract;

    public class C3 extends Contract {
      private C3 owner;

      public @FromContract(C3.class) C3() {
        owner = (C3) caller(); // ok
      }

      public @FromContract void m() {
        /* [....] */ owner.caller() /* [....] */; // error at deployment-
            time
      }
    }
```

Figure 10.6 reports our implementation of a check that verifies if a method satisfies constraints 1 and 2 above. The code has been simplified for readability: its complete version can be found in the repository of the distribution of our implementation of the runtime of Takamaka (see Section 10.3). Full understanding of the code in Figure 10.6 requires knowledge about Java bytecode and BCEL, which are outside the scope of this study. Nevertheless, it is possible to understand the structure of the code: the constructor of the check scans the stream of Java bytecode instructions of the method (`instructions()`), filters those that call a method named `caller` that returns a contract, and checks two conditions for each of them (with the two `if` statements inside the `forEach`): the method must be annotated as `FromContract` (Constraint 1 above) and the invocation must be immediately preceded by an `aload_0` bytecode instruction. The latter is Java bytecode for pushing `this` on the stack, as receiver of the call to `caller()` (Constraint 2 above). If any of the `if` statements is satisfied, an issue is generated, which will later reject the installation of the code in the blockchain.

10.3 Experimental Evaluation

We have implemented the on-chain verification for the Takamaka subset of Java, inside its runtime that works as a Tendermint application. It is an ac-

```
public class CallerIsUsedOnThisAndInFromContractCheck extends Check
    {

  public CallerIsUsedOnThisAndInFromContractCheck() {
    boolean isFromContract = annotations.isFromContract
    (className, methodName, methodArgs, methodReturnType);

    instructions()
    .filter(this::isCallToCaller)
    .forEach(ih -> {
      if (!isFromContract)
      issue(new CallerOutsideFromContractError(inferSourceFile(),
          methodName, lineOf(ih)));

      if (!previousIsLoad0(ih))
      issue(new CallerNotOnThisError(inferSourceFile(), methodName,
          lineOf(ih)));
    });
  }

  private boolean previousIsLoad0(InstructionHandle ih) {
    Instruction ins = ih.getPrev().getInstruction();
    return ins instanceof LoadInstruction && ((LoadInstruction) ins)
        .getIndex() == 0;
  }

  private final static String TAKAMAKA_CALLER_SIG = "()Lio/takamaka/
      code/lang/Contract;";

  private boolean isCallToCaller(InstructionHandle ih) {
    Instruction ins = ih.getInstruction();
    if (ins instanceof InvokeInstruction) {
      InvokeInstruction invoke = (InvokeInstruction) ins;
      ReferenceType receiver;

      return "caller".equals(invoke.getMethodName())
      && TAKAMAKA_CALLER_SIG.equals(invoke.getSignature())
      && (receiver = invoke.getReferenceType()) instanceof
          ObjectType
      && classLoader.isStorage(((ObjectType) receiver).getClassName
          ());
    }
    else
    return false;
  }
}
```

Figure 10.6: The on-chain check for a correct use of `caller()`.

tual blockchain running on Tendermint, that can be programmed with smart contracts written in Java. Our implementation is part of a larger project, called Hotmoka, whose long-term goal is to use the Takamaka language for programming both blockchains and IoT devices.

Environment Setup. All the experiments have been performed on a machine equipped with an Intel Core i3-4150 at 3.50 GHz and 16 GB of RAM memory running Ubuntu Linux 20.04.1 64bit, Oracle JDK version 13, and Go version 1.17.

We have created three scripts that request to install in the blockchain the examples from Section 10.2. We have also created a test that installs a smart contract and uses it to run many transactions, to check the scalability of the technique and evaluate the difference when on-chain verification is on or off. The implementation is available at [98]. That repository contains also the code of the 26 checks of on-chain verification (including that in Figure 10.6).

The first experiment starts a blockchain of a single node and runs a script that connects to the node and installs a jar containing class `C1` from Section 10.2. The result is successful:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C1 in the node... done (on-chain verification succeeded)
C1.jar installed at address ee848b5bc7fd8283ab01b5977970e71f548...
```

The subsequent experiment installs `C2` instead. The attempt to install the code in the blockchain will fail since on-chain verification fails:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C2 in the node...
Exception in thread "main" io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException: C2.java:8
caller() can only be used inside a @FromContract method or constructor
```

The third experiment performs the same operation with class `C3`. This attempt will fail since on-chain verification fails:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C3 in the node...
Exception in thread "main" io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException: C3.java:14
caller() can only be called on "this"
```

In order to evaluate the scalability of the technique, we have created a smart contract that creates and funds a pool of 500 externally-owned accounts and allows one to determine which is the *richest* among them (has the highest balance). We have written a JUnit test that installs that smart contract in blockchain and uses it to create and fund the 500 accounts, execute $1,000$ random money transfers between them and asks for the richest. This process is repeated ten times. The execution time of this test is 158.19 seconds. In total (including code installation and account creation) the test runs $10,020$ transactions, that is, it performs 63.34 transactions per second. By turning on-chain verification off, the same test runs in 156.95 seconds, that is, it performs 63.84 transactions per second. These numbers have been computed as an average of over five executions of the test. This shows that on-chain verification increases the execution time of the test by only 0.79%.

10.4 Evolution of Code Verification

This section shows that a change in the verification rules requires to re-verify all code installed in the blockchain and that this can be performed lazily, on-demand.

Section 10.1 stated that code verification is only performed when code is installed in the blockchain. However, that is true only under the unrealistic assumption that the verification module never changes. In practice, that module will be updated eventually, to include new verification rules or to improve the precision of already existing rules. When a new version is deployed, it becomes necessary to update all nodes to that version (or at least all validators), or otherwise, consensus might be lost. A change in the verification rules, if deployed on a subset of the network only, entails that the updated nodes might accept a request that the non-updated nodes might reject instead or vice versa.

All approaches to a network update can be used here. The novelty, however, is that some code that was successfully verified with the previous version of the verification module might be rejected with its current version or vice versa. Hence, there must be a mechanism that enforces that the execution of some code in the blockchain occurs only if that code passes the *current* verification rules. Conceptually, this means that an update of the verification module triggers a re-verification of all code previously successfully installed in the blockchain. In practice, this cannot be performed, since it would be extremely expensive and would hang the nodes for a long time. Our solution, which we are going to describe, is to lazily re-verify the code on-demand when it is asked to run. This amortizes the cost of re-verification. Moreover, [146] shows that only 0.05% of all contracts installed in Ethereum are involved in 80% of the transactions. Hence, a lazy approach avoids the re-verification of code that might actually never run again.

In order to implement this lazy re-verification approach, we expand the information in the *response* of a successful code installation *request* (Figure 10.4a). Namely, together with the installed code, *response* is enriched with a numerical tag $\tau(\textit{response})$, *i.e.*, the version of the verification module that has been used to verify the code inside *response*. The sequence diagram in Figure 10.7 shows the workflow for lazy code re-verification. Assume that a request arrives, that

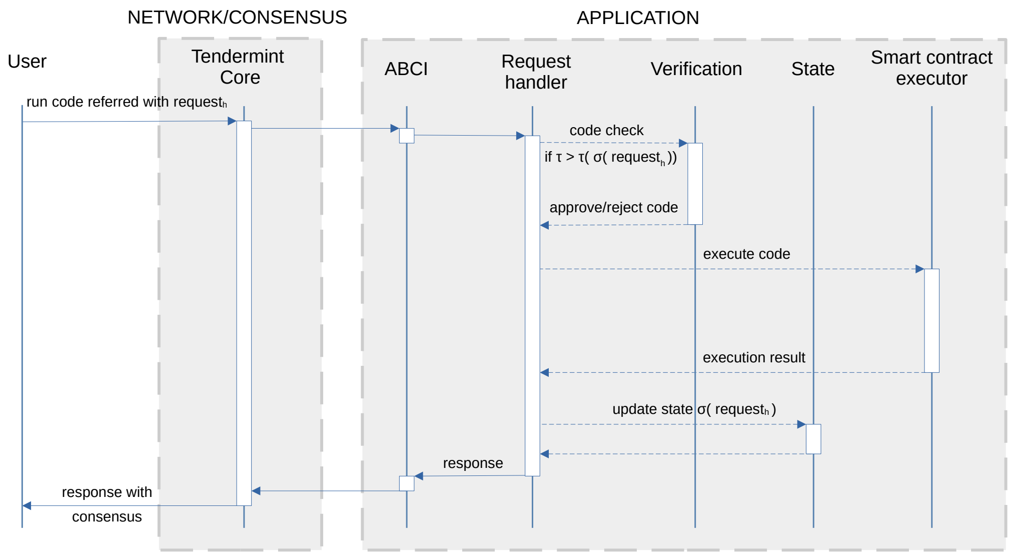


Figure 10.7: Sequence diagram for lazy code re-verification.

requires running code referred with the hash $request_h$ of a previous, successful code installation *request* (as in Figure 10.4c and 10.4d). The node finds out that $\sigma(request_h) = response$ has a verification tag $\tau(response)$ and compares it with the current version τ of the verification module. There are two possibilities:

1. $\tau = \tau(response)$: the code was verified with the current version of the verification module, it does not need re-verification and can be run immediately;
2. $\tau > \tau(response)$: the code was verified with an old version of the verification module; it must be re-verified before being run.

In the second case, the node verifies the code again, using the current version τ of the verification module. This is possible since *response* includes that code (Figure 10.4a). A new response $response'$ will be computed (successful, having τ as verification module version, or failed), and the application state is updated as $\sigma(request_h) = response'$. The use of $request_h$ in future requests will not re-verify the code until a newer version of the verification module is installed. The update is possible since it occurs in the state, not in the blockchain, whose blocks are immutable.

It is important to note that $response'$ might state that re-verification failed, because the old code passed the previous verification rules but not the new ones. In that case, the execution of the code will fail, since its classpath is not valid anymore. This means that a smart contract might work today, but might stop working tomorrow if updated verification rules reject its code. In theory, the converse is also possible: the same contract might be reactivated

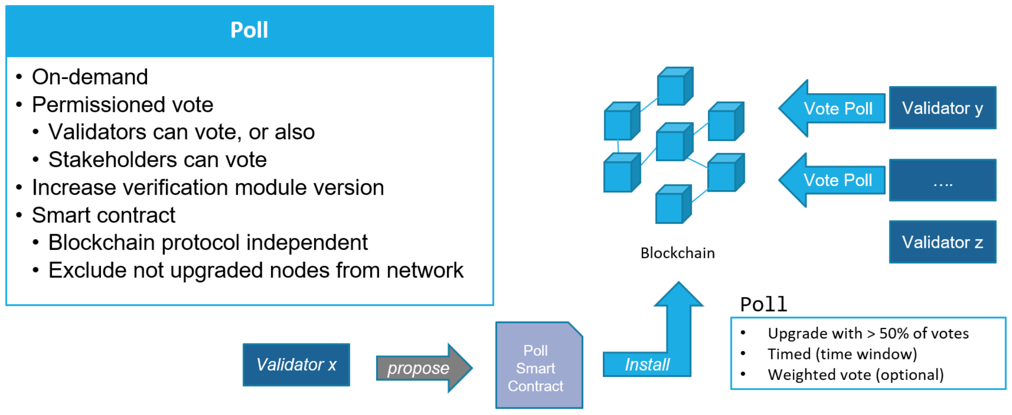

Poll
• On-demand
• Permissioned vote
• Validators can vote, or also
• Stakeholders can vote
• Increase verification module version
• Smart contract
• Blockchain protocol independent
• Exclude not upgraded nodes from network
Blockchain
Vote Poll
Validator y
Vote Poll
....
Validator z
Poll
• Upgrade with > 50% of votes
• Timed (time window)
• Weighted vote (optional)
Validator x
propose
Poll Smart Contract
Install

Figure 10.8: Schema for new verification rule upgrade.

after another change in the verification rules replaces a failed response with a successful response. However, we have decided to forbid this second scenario, since it might be surprising to users.

10.4.1 Governance for Tool Upgrade

New vulnerabilities are discovered every day and the verification tools need to be upgraded periodically to ensure sufficient security standards, as well as to improve the accuracy and performance of any existing analyses. However, in a decentralized and distributed system, there is a need for the majority of the network to agree. Different versions of tools may generate different results causing the consensus algorithm to conflict. In on-chain architecture, we govern this behavior through a poll consisting of a smart contract implementing a permissioned vote with custom attributes such as vote threshold, duration of time window, and optionally a weighted vote. In this way, the poll is *on-demand* and it allows one to be independent from blockchain protocol and verification tools because handled at the application level by the smart contract. Figure 10.8 shows a general schema of the process. The idea is that any validator or even stakeholder can propose an upgrade by deploying a poll in the blockchain. The contract can be voted on by other validators and stakeholders. If the threshold is exceeded, the upgrade will be made according to the terms of the contract, otherwise, at the expiry of the time window it will no longer be valid and the proposal will be rejected. In case of a positive poll, the current version of the verification module will be increased and the upgraded tools. Whoever does not carry out the operations will be excluded from the network.

10.5 Related Work

To our knowledge, this is the first work about on-chain verification. Similar techniques are related to continuous integration, which allow one to build and deploy code only if it passes all compilation and testing requirements. For instance, Marchesini et al. [130] describe a software development process called *Agile Block Chain Dapp Engineering* (ABCDE) to gather the requirements, to analyze, design, develop, test and deploy blockchain-oriented software. ABCDE complements the incremental and iterative development through boxed iterations, typical of agility, with more formal tools. Besides modeling interactions among BOS using UML, it also provides practices, patterns, and checklists to promote and evaluate the security of a DApp written in Solidity [129]. Furthermore, ABCDE was also applied in DApp development for HF [18]. However, the main difference with our approach is that smart contracts cannot be replaced or debugged once installed in the blockchain. In [127], there is a comparison between traditional SDLC models that highlights their inadequacy for blockchain software development, due to the immutability of blockchain data.

Even if it deviates slightly from the topic, it may be useful to mention the work of Beller and Hejderup [22] about how to use blockchain technology to solve continuous integration and package management in traditional software engineering. They sketch how this approach promises to solve fundamental issues plaguing software engineering such as quality, and trust.

Another concept often associated with blockchain verification architectures is that of *transparency* [146]. It is applied to some blockchains, including Ethereum, and consist in storing in blockchain the source code of the smart contracts, to guarantee that it actually compiles into their bytecode. This is only an optional technique that ensures that bytecode and source code match. However, there is no evidence or certainty that the code has been verified.

Lastly, about verification rule updates (Section 10.4), the specific updating technique related to the consensus layer is orthogonal to our work. In Cosmos, the government module supports such an update, with (dis-)incentives to minimize misconduct within the participants. In *Polkadot*, the stakeholders[6] are involved in periodic referendums to vote update proposals. Algorand [36] triggers a software update if a large majority of block proposers declare to be ready for that. Therefore, it does not require a predefined voting period.

[6] See `https://wiki.polkadot.network/docs/en/learn-governance`

10.6 Conclusions

In this chapter, we defined and provided a general architecture for the on-chain code verification of smart contracts. In this way, it is possible to make mandatory software verification and avoid untrusted smart contract executions. This architecture allows the same blockchain to reject the code that does not pass a set of checks. Therefore, the verification becomes part of the consensus mechanism, to ensure that all network nodes have reached the same verification result. A lazy re-verification approach is also proposed to re-check the code already deployed before its execution when the verification rules are updated.

Part III

Blockchain Software Optimization

Chapter 11

CODE OPTIMIZATIONS IN BLOCKCHAIN

This chapter introduces code optimization in blockchain software. Furthermore, it describes the case study which is optimized in the Chapter 12. Some contents of this chapter are also published in [51] and [50].

11.1 Blockchain Optimizations

According to Sedgewick [175], code optimization is the process of refactoring software to make some aspects of it work more efficiently or use fewer resources.

In blockchain software, the optimization goals are the same as traditional software, distributed and decentralized systems. In addition, specific to the blockchain context, there are also optimizations related to *gas* consumption.

The gas mechanism is a workaround to the problem of *non-termination* in the code execution. Gas is a resource that is typically declared prior to the execution of a smart contract. During the execution, the blockchain will consume units of gas for each instruction or set of instructions executed, depending on the gas model. The idea is that the code execution proceeds as long as gas is available. If the amount of gas is sufficient for termination, the execution will be completed successfully. Otherwise, the run will be aborted if there is not enough gas. It is also common for blockchains to place a limit on the maximum usable gas per transaction. Moreover, gas often corresponds to a cost in terms of crypto-currency. Therefore, the more the code is optimized with respect to the gas model, the cheaper execution will be depending on the gas model and gas limit.

It is also possible to perform *out-of-gas* attacks to provoke unwanted behavior in a victim's smart contract, e.g. wasting or blocking funds of the victim [90].

11.2 Optimize Code Migration and Translation

As argued in Section 3.2, in the enterprise scenario, it is common that the code is reused or migrated from one system to another. Code migration between different programming languages can be tricky, also for relatively simple code. There is no formal way that one can follow to perform such a translation. Therefore, the optimizations are based on re-engineering approaches and translation patterns. Languages might have different semantics for apparently similar constructs or might require different coding styles, for efficiency, which is more often the case if they compile towards different virtual machines. For instance, Vyper [189] and Solidity compile for the same EVM and the translation from Solidity to Vyper [197] is almost immediate. Differently, a translation from Solidity to Takamaka [188], since this latter compiles to Java bytecode for the Java virtual machine (JVM). In addition, some instructions could not be translatable from one language to another because they could be no equivalent semantics. Anyway, it is also possible that the target language for the translation provides data structures and algorithms, not present in the source language, able to improve the performance of blockchain software. Therefore, a literal translation does not necessarily lead to the best optimization.

11.3 Case of Study: Blockchain Token Standards

The purpose of a standard is to provide a reference model for the people to uniform a given data, method, activity, or process. In the context of blockchain, this also includes providing a secure and reliable way to exchange things between actors of the blockchain network, with the intention of making transactions less expensive and more secure at the same time.

What is exchanged is commonly referred to as a *token* [14, Chapt. 10], which is a blockchain-based abstraction that can be owned and that represents assets, currency, or access rights. A trend in blockchain is to apply standards for token interoperability, unchanged, from platform to platform, easing the design challenges with trusted and widely-used specifications.

In this section, we introduce the two main token standards and describe their most popular implementations, Chapter 12 deals with optimizations from Solidity to Takamaka.

A few standards have emerged for fungible and non-fungible tokens, that should guarantee correctness [156], accessibility, interoperability, management, and security of the smart contracts that run the tokens. Among them, the Ethereum Request for Comment #20 (ERC-20 [69]) and #721 (ERC-721 [65]) are the most popular fungible and non-fungible tokens, respectively, also outside Ethereum [104,105,116]. They provide developers with a list of rules required for the correct integration of tokens with other smart contracts and with applications external to the blockchain, such as wallets, block explorers, decentralized finance protocols, and games.

The most popular implementations of the ERC-20 standard are in Solidity, by OpenZeppelin [149], a team of programmers in the Ethereum community who deliver usefully and secure smart contracts and libraries, and by ConsenSys [43], later deprecated in favor of OpenZeppelin's. OpenZeppelin extends ERC-20 with snapshots, i.e. immutable views of the state of a token contract, that show its ledger at a specific instant of time. They are useful for investigating the consequences of an attack, for creating forks of the token and for implementing mechanisms based on token balances such as weighted voting. Snapshots are essential also to provide an immutable view of the ledger that can be queried by a client without the risk that it changes during the query, which would result in a race condition.

In the case of ERC-721, the standard implementation is in Solidity, again by OpenZeppelin [150]. That implementation does not provide a snapshot mechanism, despite the usefulness of such feature. The reason is that the already very tricky implementation in Solidity of snapshots for ERC-20 becomes intractable for the more complicated ERC-721 standard.

11.3.1 ERC-20 and its OpenZeppelin Implementation

The ERC-20 standard [69] defines an interface containing nine functions and two *events*, i.e. immutable marks saved in blockchain to attest some logical turning points. In Solidity, the owners of tokens are uniquely identified by *addresses*, which are untyped pointers to (i) *externally owned accounts* or to (ii) *contracts*. The firsts are a sort of bank accounts controlled by external applications and humans. The seconds are objects geared by their code. In principle, contracts can hold tokens. However, this could be problematic if their code is not programmed to deal with such tokens. In such a case, the tokens could remain stuck forever, since only the contract can transfer them but the code of the contract does not deal with token transfers. Therefore, it is normally assumed that only externally owned accounts own tokens, but the implementations of ERC-20 do not check this constraint and do not forbid to transfer tokens to contracts, even inadvertently. Section 11.3.2 will show that the same problem occurs for ERC-721 tokens, whose implementations have tried to solve the issue in a cumbersome and finally ineffective way.

The functions of the ERC-20 standard have three purposes:

1. Direct transfers
   - `totalSupply()` yields the integer total amount of tokens in circulation.
   - `balanceOf(address owner)` yields the amount of tokens that `owner` owns.
   - `transfer(address to, uint value)` transfers `value` tokens from the balance of the caller to the balance of `to` (`uint` is an unsigned integer of 256 bits). This function must emit a `Transfer` event.
2. Delegated transfers
   - `approve(address delegate, uint cap)` allows `delegate` to transfer up to `cap` tokens on behalf of the caller. It must emit an `Approval` event.
   - `transferFrom(address owner, address to, uint value)` transfers a `value` tokens from `owner` to `to`, but only if `owner` has `approve`d the caller to do so. This function must emit a `Transfer` event.
   - `allowance(address owner, address delegate)` yields the amount of tokens that `delegate` has been `approve`d to transfer on behalf of `owner`.
3. Optional info
   - `name()` yields the name of the tokens.
   - `symbol()` yields the symbol of the tokens.
   - `decimals()` yields the number of decimal digits of the tokens.

Figure 11.1 shows the main parts of the implementation provided by OpenZeppelin's team. The first part of this interface is just the API of a dynamic ledger of token balances. Not surprisingly, the code stores the user's balance in a field `_balances`[1] of type `mapping (address => uint)`, that binds each address to the amount of tokens it holds, and with an integer field `_totalSupply`, assigned at contract creation time.

The second part of the interface allows token owners to delegate, to other participants, the transfer of a capped amount of tokens. OpenZeppelin implements this through a field `_allowances` of type `mapping (address => mapping (address => uint))`: a map from each token owner to another map from each delegate to its allowed cap.

The third, optional part is just manifest information about the tokens. Both `transfer` and `transferFrom` use an internal function `_transfer`, that shifts the tokens from the `owner` to the destination `to`, calling the handler `_beforeTokenTransfer`. This does not do anything by default, but subclasses can redefine it to add extra functionalities to the contract. Function `_transfer` checks, defensively, for missing values (`address(0)`) that might arise from the

[1] It is customary in Solidity to start non-`public` properties with underscore.

incorrect use of the contract. Function `transferFrom` additionally checks if the `owner` of the tokens has actually delegated `msg.sender` (the caller of the function) to transfer at least `value` tokens on its behalf. This check occurs after the call to `_transfer`, which is fine since Solidity's functions do not commit their side-effects if they fail. The code of `transferFrom` ends with a call to `_approve` (not shown), which reduces the allowance. OpenZeppelin adds a `_mint` function that initializes the total supply of the token: it is internal since it is meant to be called from the constructors of subclasses that deploy actual instances of the contract. This function uses `address(0)` to represent the fact that minted tokens come *from nowhere*.

Snapshots of ERC-20 Ledgers

OpenZeppelin has subclassed its `ERC20` implementation (Section 11.3.1) to provide extra functionalities, for instance for tokens that can be (further) minted, burned, capped or paused. Among them, this section focuses on the `ERC20Snapshot` subclass only, which supports *snapshots*, shown in Figure 11.2. Namely, it adds a `_snapshot` function that performs a snapshot of the ledger and yields its progressive identifier (starting at 1). Then it overloads methods `balanceOf` and `totalSupply` from Figure 11.1 with variants that receive a snapshot identifier and yield the balance and the total supply *at the time of that snapshot* (Figure 11.2). For that, it stores the modification history of an integer variable by using the following data structure:

```
struct Snapshots {
  uint[] ids;
  uint[] values;
}
```

For instance, if a variable v is associated with a `Snapshots` structure with fields `ids={5,8,15}` and `values={6,7,20}`, then the value of v was 20 for snapshot identifiers from 9 to 15; it was 7 for snapshot identifiers from 6 to 8; it was 6 for snapshot identifiers from 1 to 5; for snapshot identifiers after 20, the value of v is v's current value in the ledger. A function `_valueAt` (not shown in Figure 11.2) reconstructs the value of a variable at a snapshot. There is one `Snapshots` instance for each address that takes part in the token, inside a new field `mapping (address => Snapshots) private _balancesSnapshots`, and for `_totalSupply`, with a new field `Snapshots private _totalSupplySnapshots`. Such structures are allocated and populated whenever a balance gets updated or the total supply changes (the latter situation occurs if mints or burns are allowed). This is achieved through the override of the internal function `_beforeTokenTransfer` (see Figure 11.2).

The code of `ERC20Snapshot`, that is very technical and consequently we omitted for sake of simplicity, has good computational complexity: it creates snapshots in $O(1)$; since the `ids` fields are sorted, it retrieves balances and total supply at each given snapshot in $O(\log n)$, by binary search, where n is the number of snapshots already performed. Nevertheless, it has some drawbacks:

- It is complex and tricky. We found it very hard to reach a sufficient trust in its correctness. It is so complicated and specific to ERC-20 that its extension from ERC-20 to ERC-721 tokens has never been done.
- It induces a significant overhead for the manipulation of the `Snapshots`, also because it needs the extra `_balancesSnapshots` map.
- All participants pay the overhead of the previous point when they transfer tokens, not just those who create snapshots. That is, if a participant creates a snapshot, then the other participants will later pay the overhead during transfers, even though they were not interested in the snapshot.
- If a large number of snapshots is generated, arrays `ids` and `values` might become so long that their manipulation exceeds the maximal gas (metering of code execution) allowed for Ethereum transactions, which is the perfect surface for a denial of service attack. That is why function `_snapshot` is internal: subclasses must implement some security policy to control its access.

11.3.2 ERC-721 and its OpenZeppelin Implementation

The ERC-721 standard [65] defines an interface with ten functions and three events. As for the ERC-20 standard, token owners can be both externally owned accounts and contracts, but contracts should be avoided, unless they have been explicitly programmed to deal with ERC-721 tokens. We will be back on this issue in a moment.

The functions of the ERC-721 standard are for:

1. Direct transfers
   - `balanceOf(address owner)` yields the amount of tokens that `owner` owns.
   - `ownerOf(uint tokenId)` yields the owner of the given token, if any.
   - `transferFrom(address from, address to, uint tokenId)` transfers the given token from `from` to `to`. In general, the caller of this function must coincide with `from`, or at least be authorized to transfer the given token on behalf of `from` (see later). This function does not even try to check that `to` is an externally owned account or a contract that will be able to deal with the token. If that is not the case, the token will be transferred to `to` and stuck forever. Because of that, this function is considered to be *unsafe*. This function must emit a `Transfer` event.

   – `safeTransferFrom(address from, address to, uint tokenId)` behaves like `transferFrom`, but additionally tries to ensure that `to` is an externally owned account or a contract able to deal with the token. In this sense, it is considered to be *safe*.
2. Delegation: function
   – `approve(address delegate, uint tokenId)` allows `delegate` to transfer the given token on behalf of the caller of the function, that must be the owner of the token or itself an authorized operator for the token. The previous delegate (if any) loses its delegation after this function has been called. This function emits an `Approval` event.
   – `setApprovalForAll(address operator, bool approved)` allows the `operator` to transfer all tokens owned by the caller of the function (if `approved` is true) or removes that right (if `approved` is false). It is possible to allow more operators per token owner. This function emits an `ApprovalForAll` event.
   – `getApproved(uint tokenId)` yields the delegate for the given token, if any.
   – `isApprovedForAll(address owner, address operator)` determines if `operator` has been authorized to transfer all tokens owned by `owner`.
3. Optional info:
   – `name()` yields the name of the tokens.
   – `symbol()` yields the symbol of the tokens.

OpenZeppelin's implementation of the ERC-721 standard is relatively long, so we only report a portion of the code in Figure 11.3. Most information is kept in four maps: `_owners` specifies who is the owner of each given token; `_balances` tells how many tokens each given owner owns; `_tokenApprovals` specifies which delegate has been authorized for each given token (if any); and `_operatorApprovals` yields the set of approved operators for each token owner. Note that `mapping (address => bool)` is actually a set of approved operators: Solidity has no set type, hence sets are encoded as their characteristic map.

Figure 11.3 shows that `transferFrom` calls an auxiliary function `_transfer` that decreases the balance of the sender, increases the balance of the receiver, and assigns the token to the receiver (`to`). There is no check on the fact that `to` is actually an externally owned account, or a contract, able to deal with the token it receives. This check exists for function `safeTransferFrom` (not shown in Figure 11.3). The idea is that contracts ready to receive ERC-721 tokens must be explicitly labeled by their programmer as implementing an `IERC721Receiver` interface, whose only method `onReceive` is called when the contracts receive an ERC-721 token. In general, it would be enough to check that `to instanceof IERC721Receiver` in order to be sure that the programmer was actually expecting the contract to receive ERC-721 tokens and to call `onReceive` in that case.

But this is not possible in Solidity, since that language lacks the `instanceof` operator and, in general, it misses any way to check the dynamic type of values. This is not just a missed feature: it is actually impossible to implement such a check in Solidity, since Ethereum implements data as unboxed values, so that their dynamic type is not available and no `instanceof` operator can ever be implemented. Because of this limitation, Solidity programmers use a very cumbersome technique, based on the ERC-165 standard [167], consisting in adding a function that yields a hash of the signatures of the methods implemented by a contract. By calling that function, it is possible, at run time, to guess the interfaces implemented by a contract. This technique (that we have highly simplified but is much more complicated than what we could express here) is very weak, since contracts are free to cheat and pretend to implement an interface that they actually do not implement. However, it is the best that a programmer can do in Solidity. There is an even weaker approach to cope with this problem. Namely, the ERC-223 token standard [61] requires to cast the token receiver to an interface `IERC223Recipient` and then call its `tokenReceived` method. If the receiver does not implement such method, the transaction fails. This is even weaker than ERC-165 since it makes no attempt to guarantee that the receiver was actually declared to implement `IERC223Recipient`: casts are unchecked in Solidity, they are pure decorations to make the compiler accept the code, but they are not verified at run time.

11.4 Conclusions

The code optimization of smart contracts leads to both the improvement of performance and the reduction of the waste of computational resources, but also to the reduction in terms of financial costs related to *gas* cost. In this chapter, we have covered some popular token standards that if optimized would benefit blockchain peers. In the next chapter, we deal with the migration from one blockchain platform to another, focusing on the optimization of tokens proposed in this chapter from Solidity to Takamaka language.

```
contract ERC20 is IERC20 {
  mapping (address => uint) private _balances;
  mapping (address => mapping (address => uint)) private _allowances;
  uint private _totalSupply;
  string private _name;
  string private _symbol;

  constructor(string name_, string symbol_) {
    _name = name_;  _symbol = symbol_;
  }

  function totalSupply() public view virtual override returns (uint) {
    return _totalSupply;
  }

  function balanceOf(address owner) public view virtual override
      returns (uint) {
    return _balances[owner];
  }

  function transfer(address to, uint value) public virtual override {
    _transfer(msg.sender, to, value);
  }

  function allowance(address owner, address delegate) public view
      virtual override returns (uint) {
    return _allowances[owner][delegate];
  }

  function transferFrom(address owner, address to, uint value) public
      virtual override {
    _transfer(owner, to, value);
    uint currentAllowance = _allowances[owner][msg.sender];
    require(currentAllowance >= value, "transfer excess");
    _approve(owner, msg.sender, currentAllowance - value);
  }

  function _transfer(address owner, address to, uint value) internal
      virtual {
    require(owner != address(0), "transfer zero address");
    require(to != address(0), "transfer to zero address");
    _beforeTokenTransfer(owner, to, value);
    uint senderBalance = _balances[owner];
    require(senderBalance >= value, "transfer excess");
    _balances[owner] = senderBalance - value;
    _balances[to] += value;
    emit Transfer(owner, to, value);
  }

  function _mint(address account, uint amount) internal virtual {
    require(account != address(0), "mint to zero address");
    _beforeTokenTransfer(address(0), account, amount);
    _totalSupply += amount;
    _balances[account] += amount;
    emit Transfer(address(0), account, amount);
  }

  function _beforeTokenTransfer(address from, address to, uint amount)
       internal virtual {
  }
}
```

Figure 11.1: A portion of OpenZeppelin's ERC-20 implementation in Solidity [152].

```
abstract contract ERC20Snapshot is ERC20 {
  Counters.Counter private _currentSnapshotId;
  struct Snapshots {
    uint[] ids;
    uint[] values;
  }

  mapping (address => Snapshots) private _balancesSnapshots;
  Snapshots private _totalSupplySnapshots;

  function _snapshot() internal virtual returns (uint) {
    _currentSnapshotId.increment();
    uint currentId = _getCurrentSnapshotId();
    // ...emit Snapshot event...
    return currentId;
  }

  function balanceOfAt(address account, uint snapshotId) public view
       virtual returns (uint) {
    (bool snapshotted, uint value) = _valueAt(snapshotId,
        _balancesSnapshots[account]);
    return snapshotted ? value : balanceOf(account);
  }

  function totalSupplyAt(uint snapshotId) public view virtual
      returns (uint) {
    (bool snapshotted, uint value) = _valueAt(snapshotId,
        _totalSupplySnapshots);
    return snapshotted ? value : totalSupply();
  }

  function _beforeTokenTransfer(address from, address to, uint
      amount) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    if (from == address(0)) {         // mint
      _updateAccountSnapshot(to);
      _updateTotalSupplySnapshot();
    } else if (to == address(0)) {    // burn
      _updateAccountSnapshot(from);
      _updateTotalSupplySnapshot();
    } else {                          // transfer
      _updateAccountSnapshot(from);
      _updateAccountSnapshot(to);
    }
  }

  // _valueAt, _updateAccountSnapshot, _updateTotalSupplySnapshot
      not shown
}
```

Figure 11.2: A portion of OpenZeppelin's Solidity ERC-20 contract with snapshots [153].

```
contract ERC721 is IERC721 {
  string private _name, _symbol;
  mapping(uint => address) private _owners; // Mapping from token ID to owner address
  mapping(address => uint) private _balances; // Mapping owner address to token count
  mapping(uint => address) private _tokenApprovals; // Mapping from token ID to approved
       address
  mapping(address => mapping(address => bool)) private _operatorApprovals; // Mapping from
       owner to approved operators

  constructor(string name_, string symbol_) {
    _name = name_;
    _symbol = symbol_;
  }

  function balanceOf(address owner) public view virtual override returns (uint) {
    return _balances[owner];
  }

  function ownerOf(uint tokenId) public view virtual override returns (address) {
    return _owners[tokenId];
  }

  function name() public view virtual override returns (string) {
    return _name;
  }

  function symbol() public view virtual override returns (string) {
    return _symbol;
  }

  function approve(address to, uint tokenId) public virtual override {
    address owner = ownerOf(tokenId); require(to != owner, "approval to current owner");
    require(_msgSender() == owner or isApprovedForAll(owner, _msgSender()), "caller is not
         owner nor approved");
    _approve(to, tokenId);
  }

  function getApproved(uint tokenId) public view virtual override returns (address) {
    return _tokenApprovals[tokenId];
  }

  function isApprovedForAll(address owner, address operator) public view virtual override
       returns (bool) {
    return _operatorApprovals[owner][operator];
  }

  function transferFrom(address from, address to, uint tokenId) public virtual override {
    require(_isApprovedOrOwner(_msgSender(), tokenId), "caller is not owner nor approved");
    _transfer(from, to, tokenId);
  }

  function _isApprovedOrOwner(address spender, uint tokenId) internal view virtual returns (
       bool) {
    address owner = ownerOf(tokenId);
    return spender==owner or isApprovedForAll(owner, spender) or getApproved(tokenId) ==
         spender;
  }

  function _transfer(address from, address to, uint tokenId) internal virtual {
    require(ownerOf(tokenId) == from, "transfer from incorrect owner");
    require(to != address(0), "transfer to the zero address");
    _beforeTokenTransfer(from, to, tokenId);
    _approve(address(0), tokenId); // Clear approvals from the previous owner
    _balances[from] -= 1; _balances[to] += 1; _owners[tokenId] = to;
    emit Transfer(from, to, tokenId);
  }

  function _approve(address to, uint tokenId) internal virtual {
    _tokenApprovals[tokenId] = to; emit Approval(ownerOf(tokenId), to, tokenId);
  }
}
```

Figure 11.3: A simplified portion of OpenZeppelin's ERC-721 implementation in Solidity [151].

Chapter 12

OPTIMIZATION OF TOKEN STANDARDS

OpenZeppelin's implementations of ERC-20 (Figure 11.1) and ERC-721 (Figure 11.3) are only around a few hundred non-comment lines of Solidity. Years of exposure to the open-source community and 35 Github contributors give some confidence in OpenZeppelin's code. This chapter presents a re-engineering of OpenZeppelin's implementation of the ERC-20 standard for fungible tokens on Takamaka. It starts with a literal translation from Solidity to Takamaka, but then describes a novel implementation for making snapshots of tokens, based on tree maps, that is possible in Java, but not in Solidity, and shows that it is much more efficient than the literal translation in Java from Solidity, within the Java Virtual Machine. Some contents of this chapter are also published in [51] and [50].

12.1 From Solidity to Takamaka

In this section, we propose a process to translate as literally as possible a program written in Solidity to the Takamaka language.

Takamaka compiles for the JVM and the translation from Solidity to Takamaka is more difficult. In many cases, different programming languages have specific solutions that cannot be translated literally: for instance, Java has an `instanceof` operator, hence it is pointless to translate the ERC-165-based technique used in Solidity to allow contracts to hold ERC-721 tokens only if they explicitly declare to implement a specific interface. Just use `instanceof` in Java

```
public class ERC20 extends Contract implements IERC20 {
  private final UnsignedBigInteger ZERO = new UnsignedBigInteger("0");
  private final StorageMap<Contract,UnsignedBigInteger> _balances = new StorageTreeMap<>();
  private final StorageMap<Contract,StorageMap<Contract,UnsignedBigInteger>> _allowances =
      new StorageTreeMap<>();
  private UnsignedBigInteger _totalSupply = ZERO;
  private final String _name, _symbol;

  public ERC20(String name, String symbol) { _name = name; _symbol = symbol; }

  public final @Override @View UnsignedBigInteger totalSupply() {
    return _totalSupply;
  }

  public final @Override @View UnsignedBigInteger balanceOf(Contract owner) {
    return _balances.getOrDefault(owner, ZERO);
  }

  public final @Override @FromContract void transfer(Contract to, UnsignedBigInteger value)
      {
    _transfer(caller(), to, value);
  }

  public final @Override @View UnsignedBigInteger allowance(Contract owner, Contract
      delegate) {
    return _allowances.getOrDefault(owner, StorageTreeMap::new).getOrDefault(delegate, ZERO)
        ;
  }

  protected final void transferFrom(Contract owner, Contract to, UnsignedBigInteger value) {
    _transfer(caller(), to, value);
    _approve(caller(), owner, allowance(owner, caller()).subtract(value, "transfer excess"))
        ;
  }

  protected void _transfer(Contract owner, Contract to, UnsignedBigInteger value) {
    require(owner != null, "transfer from null account");
    require(to != null, "transfer to the null account");
    require(value != null, "value cannot be null");
    _beforeTokenTransfer(owner, to, value);
    _balances.put(owner, balanceOf(owner).subtract(value, "transfer excess"));
    _balances.put(to, balanceOf(to).add(value));
    event(new Transfer(owner, to, value));
  }

  protected void _mint(Contract account, UnsignedBigInteger amount) {
    require(account != null, "mint to the null account");
    require(amount != null, "amount cannot be null");
    _beforeTokenTransfer(null, account, amount);
    _totalSupply = _totalSupply.add(amount);
    _balances.put(account, balanceOf(account).add(amount));
    event(new Transfer(null, account, amount));
  }

  protected void _beforeTokenTransfer(Contract from, Contract to, UnsignedBigInteger amount)
       {
  }
}
```

Figure 12.1: A portion of our ERC-20 implementation in Takamaka [99].

instead. Nevertheless, our investigation of both languages highlights some translation patterns from Solidity to Takamaka, as shown below.

Visibility modifiers. Solidity's `public` and `private` have direct Java equivalents. Solidity's `internal` corresponds to Java's `protected`, but the latter grants access also to code in the same package of the class `C` where `protected` is used, which is not the case for `internal` (Solidity has no packages). This might be dangerous since an attacker might place a new class in `C`'s package and get access to `C`'s methods that were meant to be `C`'s implementation details. To avoid this scenario, the verifier of Takamaka code, that Hotmoka runs before installing code in blockchain, rejects split packages, i.e. does not allow two classes in the same package to occur in different jars (Java archives) in the classpath (Java enforces the same constraint only from Java 9). Thanks to this constraint, `internal` can be safely translated into Java's `protected`. Solidity's `external` grants access to a function only to other contracts and, in this sense, it is used to specify the public API of a contract. There is no such visibility notion in Java. However, Takamaka introduces the `@FromContract` annotation, which restricts the callers of a method or constructor to contracts. Hence `external` can be translated into `public @FromContract`.

The following table summarizes the translation:

Solidity	**Takamaka** *(Java)*
`public`	`public`
`private`	`private`
`internal`	`protected`
`external`	`public @FromContract`

`view` *modifier.* In Solidity, this states that a function (such as `balanceOf` in Figure 11.1) has no side effects and can consequently be executed outside of transactions, in every single node of the blockchain. This translates into Takamaka's `@View` annotation, with the same semantics.

`override` and `virtual` modifiers. Solidity and Java take opposite approaches to non-`private` methods redefinition. Namely, methods can be redefined in Solidity only if they are marked with `virtual` and redefinitions must be marked with `override`. In Java, methods can always be redefined unless they are marked with `final` and redefinitions do not need any special syntactical mark, although the `@Override` annotation has become customary. Consequently, the translation of these modifiers from Solidity to Takamaka is the following:

Solidity	**Takamaka** *(Java)*
`virtual f(args) returns T`	`T f(args)`
`override f(args) returns T`	`@Override T f(args)`
`f(args) returns T`	`final T f(args)`

`uint` type. Solidity uses `uint` (short form of `uint256`) to represent unsigned, potentially very large integers (up to $2^{256}-1$). For instance, ERC-20 implemen-

tations use `uint` to represent token balances (Figure 11.1). This type suffers from (silent) underflows and overflows. To cope with this problem, Solidity code can use the SafeMath library that provides arithmetic functions with defensive checks against underflows, overflows, and divisions by zero. The latest versions of Solidity implement such checks in the language, natively, at an increased gas cost. Takamaka code can use `UnsignedBigInteger` for that, a wrapper of Java's `BigInteger` class, from Takamaka's support library, whose operations include defensive checks, with the extra advantage that they are unbounded unsigned integers, hence do not suffer from overflows.

`mapping` *type.* Solidity uses the `mapping` type for maps between values, as for field `_balances` in Figure 11.1. These are not data structures, but rather an algorithm that spreads the bindings of the mapping in the key/value store of Ethereum (with an *unlikely* risk of hash collision). Takamaka can use an actual, generic data structure `StorageTreeMap<Key,Value>` instead, an implementation of the interface `StorageMap<Key,Value>`, from Takamaka's support library. Solidity's maps default to 0, hence one must use `getOrDefault(index, 0)` calls on `StorageTreeMap` in Takamaka. If `mapping` is used in Solidity as a trick to implement a set (as in the codomain of `_operatorApproval` in Figure 11.3), then in Takamaka it is simpler and more efficient to use a `StorageTreeSet<Value>` instead, that is an implementation of the interface `StorageSet<Value>` from Takamaka's support library.

`msg.sender`. This Solidity expression refers to the contract that calls a function. In Takamaka, this corresponds to `caller()` inside a `@FromContract` method.

`address(0)`. This Solidity expression refers to a contract or account at address 0. It is assumed that nobody controls that contract or account. Hence, traditionally, it stands for a missing value or for the sign of missing information in a transaction request. In Takamaka, the same can be achieved with `null`.

Figure 12.1 shows our manual translation in Takamaka of the Solidity code for ERC-20 in Figure 11.1, by following the heuristics above. The translation is almost literal, with a few exceptions. For instance, function `transferFrom` in Figure 11.1 enforces a non-negative allowance through a `require` assertion. In Figure 12.1, that same check is moved inside the `subtract` method of the `UnsignedBigInteger` class.

Figure 12.2 shows our manual translation in Takamaka of the Solidity code for ERC-721 in Figure 11.3. Also, this translation is almost literal. We observe that the `_operatorApprovals` field uses a `StorageSet` in Takamaka, instead of the Solidity trick of using a map to represent a set. Token instances are represented as `BigInteger` in Takamaka, hence they are more general than in Solidity, where they are limited to be `uint`, hence 256 bits only. The `_balances` field uses

```
public class ERC721 extends Contract implements IERC721 {
  private final StorageMap<BigInteger,Contract> _owners = new StorageTreeMap<>();
  private final StorageMap<Contract,BigInteger> _balances = new StorageTreeMap<>();
  private final StorageMap<BigInteger,Contract> _tokenApprovals = new StorageTreeMap<>();
  private final StorageMap<Contract,StorageSet<Contract>> _operatorApprovals = new
       StorageTreeMap<>();
  private final String _name, _symbol;

  public ERC721(String name, String symbol) { _name = name; _symbol = symbol; }
  public final @Override @View BigInteger balanceOf(Contract owner) { return _balances.
       getOrDefault(owner, ZERO); }
  public final @Override @View Contract ownerOf(BigInteger tokenId) { return _owners.get(
       tokenId); }
  public final @View String name() { return _name; }
  public final @View String symbol() { return _symbol; }

  public @Override @FromContract void approve(Contract to, BigInteger tokenId) {
    Contract owner = ownerOf(tokenId); require(owner != to, "approval to current owner");
    Contract caller = caller();
    require(caller == owner or isApprovedForAll(owner, caller), "caller is not owner nor
         approved");
    _approve(to, tokenId); }

  public @Override @View Contract getApproved(BigInteger tokenId) { return _tokenApprovals.
       get(tokenId); }

  public @Override @View boolean isApprovedForAll(Contract owner, Contract operator) {
    StorageSet<Contract> approvedForAll = _operatorApprovals.get(owner);
    return approvedForAll != null && approvedForAll.contains(operator); }

  public @Override @FromContract void transferFrom(Contract from, Contract to, BigInteger
       tokenId) {
    require(_isApprovedOrOwner(caller(), tokenId), "caller is not owner nor approved");
    require(to instanceof ExternallyOwnedAccount or to instanceof IERC721Receiver,
    "transfer destination must be an externally owned account or implement IERC721Receiver")
         ;
    _transfer(from, to, tokenId); }

  protected boolean _isApprovedOrOwner(Contract spender, BigInteger tokenId) {
    Contract owner = ownerOf(tokenId);
    return spender == owner or isApprovedForAll(owner, spender) or getApproved(tokenId) ==
         spender;
  }

  protected void _transfer(Contract from, Contract to, BigInteger tokenId) {
    require(ownerOf(tokenId) == from, "transfer from incorrect owner");
    require(to != null, "transfer to null");
    _beforeTokenTransfer(from, to, tokenId); _approve(null, tokenId);
    _balances.put(from, balanceOf(from).subtract(BigInteger.ONE));
    _balances.put(to, balanceOf(to).add(BigInteger.ONE)); _owners.put(tokenId, to);
    if (to instanceof IERC721Receiver) ((IERC721Receiver) to).onReceive(this, from, to,
         tokenId);
    event(new Transfer(from, to, tokenId)); }

  protected void _approve(Contract to, BigInteger tokenId) {
    if (to == null) _tokenApprovals.remove(to); else _tokenApprovals.put(tokenId, to);
    event(new Approval(owner, to, tokenId)); }
}
```

Figure 12.2: A portion of our ERC-721 implementation in Takamaka [100].

`BigInteger` to represent the balance of each token holder. This is cheaper than `UnsignedBigInteger` and has been preferred in this case since the code of the contract guarantees such values to be non-negative, hence the run-time checks of `UnsignedBigInteger` are not useful here. Maps in Takamaka cannot use the handy indexing notation of Solidity and do not use `null` to represent a missing binding. This explains why the Takamaka code is sometimes a bit more verbose (see for instance the methods `isApprovedForAll` and `_approve`). In Takamaka, both methods `transferFrom` and `safeTransferFrom` have been collapsed into

```
contract ERC20 is IERC20 {
  mapping (address = > uint) private _balances;
  uint private _totalSupply;

  struct SnapshotImpl {
    immutable mapping (address => unit) public balances;
    immutable uint public totalSupply;
  }

  function snapshot() public returns (SnapshotImpl) {
    return (immutable clone of _balances, copy of _totalSupply) }
}
```

Figure 12.3: The pseudocode of an alternative implementation of snapshots in Solidity, that its compiler does not accept.

a single method `transferFrom` that safely checks if the receiver of the token is an externally owned account or a contract that implements `IERC721Receiver`. In this latter case, its `onReceive` method is called. The check on the type of the receiver is sound in Takamaka and doesn't need the tricky and fragile ERC-165 machinery, since Java has an `instanceof` operator that fails if the test is false.

12.1.1 Snapshot ERC-20 in Takamaka

We have translated in Takamaka the Solidity code from Figure 11.2. The result of this translation is at [101]. It works perfectly but suffers from the same issues highlighted above for its Solidity counterpart. Hence, it is interesting to investigate whether a better implementation of ERC-20 contracts with snapshots exists, at least in Takamaka, which is our target language. Moreover, it is interesting to see if that implementation can also work for ERC-721 tokens, currently missing the snapshot feature in Solidity.

12.2 An Efficient Algorithm for Snapshots

By looking at OpenZeppelin's code in Figure 11.1, it would be convenient to implement the `_snapshot` function in a way completely different from that described in Section 11.3.1: it should return an actual snapshot (not its identifier), i.e. a data structure containing an immutable view of the ledger. This new implementation does not increase the length of any array and can be safely `public`. In Solidity-like code, this would look like in Figure 12.3. However, this code cannot be written in Solidity. The main reason is that Solidity maps cannot be cloned, since they are not data structures, but just an algorithm for distributing

```
public class ERC20 extends Contract implements IERC20 {
  private UnsignedBigInteger _totalSupply = ZERO;
  private final StorageMap<Contract, UnsignedBigInteger> _balances =
       new StorageTreeMap<>();

  public final IERC20View snapshot() {

    class SnapshotImpl extends Storage implements IERC20View {
      private final UnsignedBigInteger totalSupply = _totalSupply;
      private final StorageMapView<Contract,UnsignedBigInteger>
          balance = _balances.snapshot();

      public @Override @View UnsignedBigInteger totalSupply() {
        return totalSupply;
      }

      public @Override @View UnsignedBigInteger balanceOf(Contract
          account) {
        return balances.getOrDefault(account, ZERO);
      }

      // the snapshot of a snapshot is itself
      public @Override @View IERC20View snapshot() {
        return this;
      }
    }

    return new SnapshotImpl();
  }
}
```

Figure 12.4: The `snapshot` method added to the code in Figure 12.1.

key/value pairs in the storage of Ethereum. Solidity maps do not even know their set of keys, whose iteration would at least allow a (very expensive) clone of the map. Moreover, at the time we conducted the analysis and experiments, Solidity functions could not return a `struct` (from Solidity v0.8, ABIEncoderV2 implements that feature). This is why we talk about pseudocode in Figure 12.3: it does not really compile.

Figure 12.4 shows that the corresponding code can well be written in Takamaka instead. The local inner class `SnapshotImpl` plays the role of the `struct` in Solidity. At creation time, it clones fields `_totalSupply` and `_balances` from the outer `ERC20` object. Class `SnapshotImpl` actually implements a new superinterface `IERC20View` of `IERC20`, that has only the read-only methods of ERC-20, i.e. `totalSupply` and `balanceOf`. Figure 12.5 shows the UML diagram of these interfaces and classes. It shows that there is no special class for ERC-20 contracts with snapshots anymore: all ERC-20 contracts can be snapshotted.

The core idea of this Java code is that, in Takamaka, an immutable clone of `_balances` is simply `_balances.snapshot()` (the `snapshot` method of `StorageMap`),

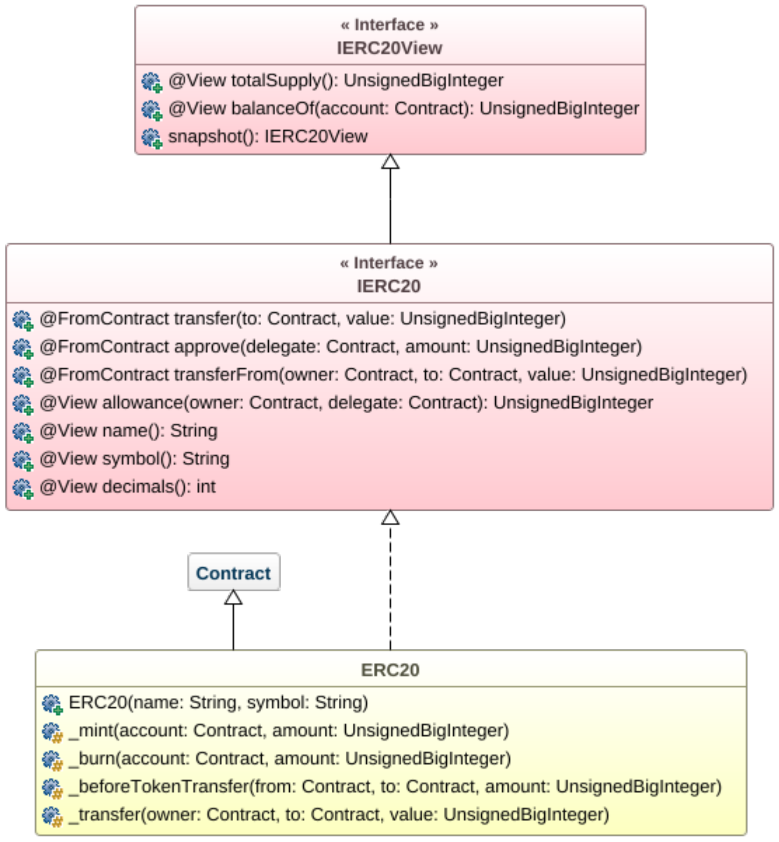

« Interface »
IERC20View
@View totalSupply(): UnsignedBigInteger
@View balanceOf(account: Contract): UnsignedBigInteger
snapshot(): IERC20View
« Interface »
IERC20
@FromContract transfer(to: Contract, value: UnsignedBigInteger)
@FromContract approve(delegate: Contract, amount: UnsignedBigInteger)
@FromContract transferFrom(owner: Contract, to: Contract, value: UnsignedBigInteger)
@View allowance(owner: Contract, delegate: Contract): UnsignedBigInteger
@View name(): String
@View symbol(): String
@View decimals(): int
Contract
ERC20
ERC20(name: String, symbol: String)
_mint(account: Contract, amount: UnsignedBigInteger)
_burn(account: Contract, amount: UnsignedBigInteger)
_beforeTokenTransfer(from: Contract, to: Contract, amount: UnsignedBigInteger)
_transfer(owner: Contract, to: Contract, value: UnsignedBigInteger)

Figure 12.5: The UML class diagram of the `IERC20View` and `IERC20` interfaces, implemented by the `ERC20` class. The `IERC20View` interface is a very abstract view of a ledger: it has methods for read-only access and for creating snapshots.

that runs in $O(1)$. Therefore, the problem is now to understand how the class `StorageTreeMap` and its `snapshot` method work. They exploit the same idea used, for instance, in the Git version control system and in the storage of Ethereum, allowing one to check out their full history of states, by simply swapping a root pointer. They favor the re-creation of immutable data structures instead of updates to mutable data structures. More in detail, in our case class `StorageTreeMap<K,V>` implements red/black trees [175], a special kind of balanced binary search trees that orders keys of type `K` by their *storage reference*, i.e. a machine-independent pointer to the keys in the memory of the blockchain [188]. Such references are 32 bytes long, i.e. 256 bits. Since a red/black tree is balanced, the length of a path from root to leaf is 256 at most and get and put operations run in $O(256)$, i.e. in $O(1)$. Figure 12.6a shows a `StorageTreeMap<Contract,UnsignedBigInteger> _balances` that implements the mapping with the following insertion order: 81af $\mapsto$ 14, 77b1 $\mapsto$ 18, da89 $\mapsto$ 14, 71a0 $\mapsto$ 19, fa31 $\mapsto$ 35 and 9100 $\mapsto$ 5 (for simplicity, this example assumes that storage references are only two bytes long, i.e. four hexadecimal digits or 16 bits). We remember that the O notation states a *worst-case* scenario. Namely, the cost for get and put is often smaller than 256 operations, being in general dependent on the number of elements in the tree. We are not stating that get and put cost always exactly 256 operations, which would need the Θ notation instead. What we are stating is that it is never higher than 256, which is the meaning of the O notation. The fact that get and put run in constant worst-case time is made possible by the choice of a particular kind of keys, whose size is fixed a priori. The situation here is similar to the use of Merkle-Patricia tries for implement the storage of Ethereum, whose get and put operations are considered to run in constant time as well since their cost increases with the size of the trie but is bounded from above by a constant [14, 33]. Also, in that case, constant worst-case time is possible since keys are Ethereum addresses, hence of fixed size.

Figure 12.6a shows also the computation of a clone of `_balances`: it is another `StorageTreeMap` whose root is the same root of `_balances`. The independence between `_balances` and its clones is obtained by making the nodes of the trees immutable data structures: destructive updates of the tree actually create new nodes instead of modifying old nodes. For instance, Figure 12.6b shows an update to `_balances`, that changes the value bound to da89, from 14 to 30. It shows that both nodes for 81af and da89 are recreated (darkened in the figure), and the root of `_balances` is updated. The clone's root remains unchanged instead and points to the old tree. Note that computing a clone means just creating a new root cell that points to the current root of the tree. Hence, a clone is computed in $O(1)$. The idea of creating independent clones of a tree by using immutable nodes and a new root pointer is not new. We have borrowed this idea from the way the Git version control system works internally. Git allows very inexpensive creation of branches of a repository in $O(1)$, since a branch is just a moving reference to the root of the repository.

The code in Figure 12.4 has the same asymptotical complexity as OpenZeppelin's ERC-20 contracts with snapshots, but overcomes all its drawbacks reported at the end of Section 11.3.1:

1. It is simple and intuitive. Class `StorageTreeMap` might look complex but comes with the support library of Takamaka and needn't be re-implemented.
2. It has no overhead because of snapshots and no `_balancesSnapshots` map exists anymore.

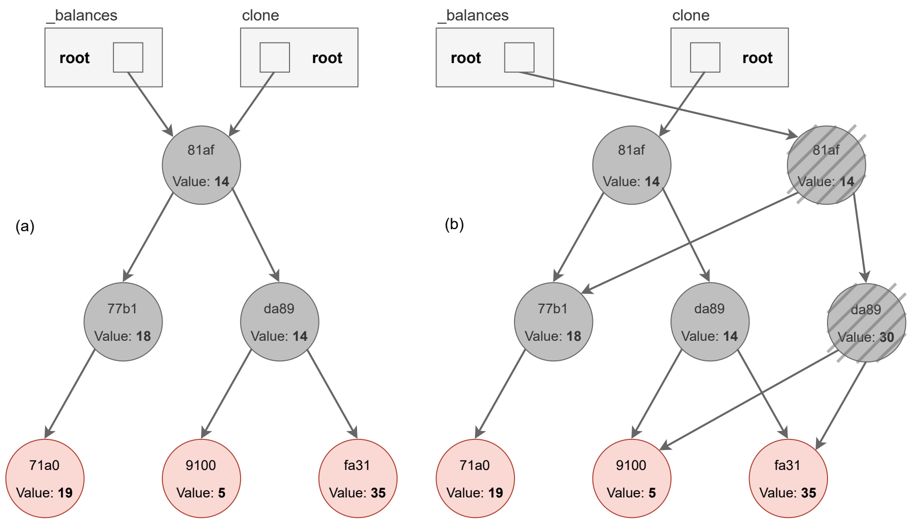


Figure 12.6: A red/black tree with immutable nodes, with snapshots in $O(1)$.

3. Who creates a snapshot pays gas. The other participants can transfer coins without paying any overhead because of that snapshot.
4. There are no arrays that grow in size when snapshots are created, hence a denial of service attack is not possible.

Moreover, the same technique can be used to implement a snapshot of an ERC-721 token ledger as well. There is no extra difficulty in comparison with ERC-20 ledgers. The only difference is that the snapshot must be performed for *two* maps this time: for the `_balances` and for the `_owners` maps of the implementation in Figure 12.2. The `snapshot` method added to the code in Figure 12.2 is shown in Figure 12.7. Also, in this case, there is an `IERC721View` interface that collects the read-only methods of `IERC721`.

The same technique can be applied to generate snapshots of other data structures. The idea is always that described above, based on the `snapshot` method of the underlying components of the data structure. For instance, we have implemented snapshots also for *shared entities* [23], that represent objects divided in many, dynamically changing shares, such as a private company, a DAO or the set of validators of a blockchain.

```
public class ERC721 extends Contract implements IERC721 {

  public final IERC721View snapshot() {

    class SnapshotImpl extends Storage implements IERC721View {
      private final StorageMapView<BigInteger,Contract> owners =
          _owners.snapshot();
      private final StorageMapView<Contract,BigInteger> balances =
          _balances.snapshot();

      public @Override @View BigInteger balanceOf(Contract owner) {
        return balances.getOrDefault(owner, ZERO);
      }

      public @Override @View Contract ownerOf(BigInteger tokenId) {
        return owners.get(tokenId);
      }

      // the snapshot of a snapshot is itself
      public @Override @View IERC721View snapshot() {
        return this;
      }
    }

    return new SnapshotImpl();
  }
}
```

Figure 12.7: The `snapshot` method added to the code in Figure 12.2.

12.3 Related Work

The ERC-20 standard [69] for fungible tokens was originally defined for initial coin offers and for the definition of new kinds of tokens supported by the underlying, native token of the blockchain. The ERC-721 standard [65] for non-fungible tokens has experienced an impressive success, mainly for the definition of NFTs for art and, in general, for representing things having a specific value. Their OpenZeppelin implementations [149] and [150], respectively, are currently the de facto standard implementations for Ethereum-like blockchains. The importance of such standards is growing with the progressive application of blockchain technology beyond its original context of cryptocurrency. Different application contexts also involve different programming languages, systems, and platforms, which must implement and support these standards. For instance, Hyperledger Fabric proposes some sample implementation in Java, Go, Javascript [104, 105]. Instead, for Cosmos, there are implementations written in Rust [109]. However, they limit themselves to proposing minimal versions of these standards by omitting valuable features, like snapshots, without offering improvements by exploiting the target languages.

12.4 Performance Evaluation

This section compares the performance of the literal translation into Takamaka of OpenZeppelin's ERC-20 contracts with snapshots (Section 11.3.1) against that of our implementation in Takamaka that uses a more efficient snapshot algorithm (Figures 12.1 and 12.4), which we call **Native**, in terms of gas consumed for code execution. Gas is the standard cost measure for smart contracts, since it reflects the actual number of resources (CPU cycles, RAM allocations, storage slots) that each node of a blockchain must consume. However, gas is a low-level and is a bytecode-specific measure. Solidity and Takamaka use two completely different bytecode languages. Because of that, what we are actually going to compare is OpenZeppelin's ERC-20 contract with snapshots translated in Takamaka (end of Section 11.3.1), that we call **OpenZeppelin**, against our **Native**. Both are written in Takamaka and both are compiled into Java bytecode. Hence, the comparison gives a measure of the relative efficiency of the two algorithmic solutions, which is what we are looking for. Instead, this is *not* a comparison between OpenZeppelin's Solidity code and our Takamaka code, or more generally between Solidity and Takamaka, that would be meaningless and that we cannot provide, since they compile into distinct bytecode languages, have different gas models and do not allow the same algorithmic solutions: maps can be cloned in Takamaka but not in Solidity.

We have written a JUnit test case that simulates a typical usage scenario for an ERC-20 contract: it creates the contract in blockchain, spreads its tokens among a set of investors (some random externally owned accounts), plays for some time with the ERC-20 contract (we assumed for ten days), performing random token transfers between them, burning some random tokens or minting new random tokens. At the end of each day, it takes a snapshot. The test case is implementation-agnostic: given an implementation of ERC-20 with snapshots (such as **OpenZeppelin** or **Native**), the test case will reproduce the scenario and report the gas consumption. Moreover, in order to be deterministic and fair, the test case uses a fixed seed for random choices. Hence its execution is exactly the same at each run, with both **OpenZeppelin** and **Native**. Similarly, the number and kind of transactions executed by the test case do not change. The implementation is available at [97].

Environment Setup. All the experiments have been performed on a machine equipped with an Intel Core i5-8259U 2,30/3,80 GHz and 16 GB of RAM memory running Ubuntu Linux 20.04.2 64bit, Oracle JDK version 13.

Table 12.8 shows the results. It reports the result of running our test that simulates ten days of interaction with an ERC-20 contract, performing a snapshot at the end of each day. *Implementation* is the implementation under test:

Implementation	Investors	Transfers	Mints	Burns	Txs	CPU	RAM	Storage	Time
Native	100	219	103	99	433	**2326293**	**3589999**	**66336137**	**1.98**
OpenZeppelin	100	219	103	99	433	**4020320**	**5808375**	**130334125**	**2.02**
Native	200	832	205	194	1243	**7636110**	**11627281**	**285906113**	**4.61**
OpenZeppelin	200	832	205	194	1243	**13766079**	**19617649**	**529992972**	**5.86**
Native	300	1776	302	316	2406	**15645862**	**23705622**	**655534336**	**9.82**
OpenZeppelin	300	1776	302	316	2406	**28372984**	**40299922**	**1216043831**	**12.43**
Native	400	3260	383	411	4066	**27995748**	**42184186**	**1272430439**	**16.22**
OpenZeppelin	400	3260	383	411	4066	**51587088**	**72881258**	**2332030574**	**24.02**
Native	500	5170	512	506	6200	**43846203**	**65859592**	**2086138985**	**27.68**
OpenZeppelin	500	5170	512	506	6200	**81836726**	**115260504**	**3801993384**	**43.42**
Native	600	7326	590	599	8527	**61657573**	**92597805**	**3064332633**	**44.20**
OpenZeppelin	600	7326	590	599	8527	**115871428**	**163144629**	**5600364411**	**68.47**
Native	700	10038	738	710	11498	**85337821**	**127833698**	**4327160882**	**68.61**
OpenZeppelin	700	10038	738	710	11498	**160102275**	**225029886**	**7815254989**	**107.20**
Native	800	12896	759	871	14538	**110260986**	**164858626**	**5673424050**	**98.56**
OpenZeppelin	800	12896	759	871	14538	**208781103**	**293035390**	**10340769047**	**160.49**
Native	900	15939	884	901	17736	**137069383**	**204568208**	**7154706461**	**144.09**
OpenZeppelin	900	15939	884	901	17736	**261476515**	**366375520**	**13058660548**	**231.03**
Native	1000	20390	939	1031	22372	**175274120**	**261148704**	**9282181878**	**223.00**
OpenZeppelin	1000	20390	939	1031	22372	**333622702**	**467160332**	**16925335716**	**344.23**

Figure 12.8: Result of test simulating ten days of interaction with proposed ERC-20 contract implementations.

native Takamaka with efficient snapshots or translated from OpenZeppelin into Takamaka. *Investors* is the number of accounts that invest in the ERC-20 contract. *Transfers*, *Mints* and *Burns* are the number of transfer, mint and burn transactions performed during the test, respectively. *Txs* is the total number of transactions performed by the test, including those for the creation and initialization of the ERC-20 contract and for the computation of its snapshots. *CPU*, *RAM*, and *Storage* are the gas units consumed for CPU execution, RAM allocation, and persistent storage in blockchain, respectively. *Time* is the time for the execution of the test, in seconds.

For instance, the test with 1000 investors generates 22372 transactions. With our **Native** contract, it consumes a total of 9718604702 units of gas (CPU+RAM+Storage) and takes 223 seconds. With the **OpenZeppelin** contract, it consumes 17726118750 units of gas (CPU+RAM+Storage, almost twice as **Native**) and takes 344 seconds.

This experiment shows that our **Native** solution with efficient snapshots (Figure 12.4) saves gas units (hence money) and reduces the overall time for the execution of the test case. This time reduction is more apparent when there are many investors, as the overhead of **OpenZeppelin**'s solution consequently grows.

12.5 Conclusions

This chapter proposed a translation process from the Solidity language to the Takamaka (Java) language, providing the implementation of specific standards for fungible and non-fungible tokens. Next, snapshot algorithm optimizations are applied to reduce the gas and time costs of our implementations within the JVM. These optimizations are based on maps with immutable clones, not available in Solidity but implementable in Java. Experimental results report an improvement compared to literal translations.

The same approach may be also applied to other blockchain languages and platforms, which can support the proposed data structures. Furthermore, the snapshot function may be included in other standards, such as our ERC-721 contracts, where snapshots were previously lacking in the Solidity version of OpenZeppelin.

Part IV

Conclusion

Chapter 13

FINAL CONCLUSIONS AND FUTURE DIRECTIONS

This thesis has investigated two challenging topics concerning blockchain: *software verification* and *code optimization*. We proposed generic approaches to analyze and optimize blockchain software. We implemented and applied them to real-world contexts.

Static Analyzers for Blockchain Software

We deal with the design and development, from scratch, of tools to define static analyses for the Go and Michelson languages. Although the challenges related to language modeling have been different, thanks also to the support of LiSA [72], it has been possible to reduce the technological gap and the development times. In this direction, we also proposed the first IR for LiSA to handle only stack-based languages exploiting SSA form and a symbolic stack. Moreover, we have empirically demonstrated for the first time the use of LiSA in the industrial field. In the future, we will improve the coverage of the analyzable code and in the case of Go, support new frameworks not necessarily related to the blockchain scenario. Furthermore, the development of these analyzers has allowed us to acquire skills for the creation from scratch of analyzers both on high and low-level languages, modeling GLPs and DSLs. A next step will be the creation of a new analyzer for other popular languages such as Python [10], Rust [9], etc. Moreover, given the peculiarity of LiSA, the topic of multi-language analysis between blockchain software layers with different languages will also be addressed.

Analyses for Blockchain Software

Regarding the analyses proposed for these analyzers. We described how to detect non-determinism issues exploiting information flow analyses. To the best of our knowledge, these techniques have never been applied so far to track non-determinism properties on blockchain software. Furthermore, they have allowed us to reduce the number of false positives compared to other tools used for the detection of the same problem. Moreover, we think the intuition of *"only those that affect the status or response of the blockchain can cause problems within the blockchain"* may also be used to improve other analyses for blockchain software. Furthermore, we investigate the issues related to UCCI, proposing an analysis based on the information flow that has allowed the detection of behaviors that lead to the execution of arbitrary code. Also in this case, as far as we know, it is the first implementation based on flow analysis for the detection of this issue in Go blockchain frameworks and smart contracts written in Michelson. For numerical issues, we describe a preliminary study regarding abstract numerical domains. Future work will improve the accuracy of the analysis and reduce false positive warnings, especially with regard to numerical issues. An additional idea regarding information analysis could be the implementation of a backward analysis for the reconstruction of the single source-sink paths as we have done in BackFlow [74] reconstructor. Indeed, the information flow analyses proposed for non-determinism and UCCI are *forward*: they start from a source and they trigger an alarm when the information flows into sinks. Hence, the end user will only know the sinks, but may not have clear evidence of how the information flowed at that point and from what source.

On-chain Verification Architecture

We introduced an alternative paradigm for blockchain verification, where the nodes of the blockchain verify the deployed code. That is, the same network, internally, runs a mandatory code verification step and rejects code that does not pass it. As a consequence, on-chain verification is a defensive, proactive technique that guarantees that all code executed in the blockchain has been successfully verified. On-chain verification must be efficient, in order not to block the nodes of the network. Our experiments show that the time of analysis is largely dominated by the time of block creation, also because smart contracts are typically small. In the future, we will also make other investigations on the proposed analyzers based on LiSA. Nevertheless, the on-chain application of powerful static analyses, such as those currently running, for instance, on Java desktop applications [186], seems challenging. Moreover, the re-verification of code already in the blockchain might not be the best choice, since it might disable some smart contracts already in the blockchain and lock their funds. A change in the verification rules might be opposed by a large number of users if it affects some highly popular contracts.

Smart Contract Code Optimization

We proposed a translation process from Solidity to Takamaka. Then, we provided the translation of ERC-20 and ERC-721 standards for fungible and non-fungible tokens, respectively. Next, snapshot algorithms have been applied to reduce the gas and time costs within the JVM. These algorithms use data structures that are not possible in Solidity but are implementable in Java, significantly optimizing the code. Currently, the translation and optimization are not automated. It will be considered in the future whether to automate the process at least in part.

REFERENCES

1. Aggarwal, S., Kumar, N.: Introduction to blockchain. In: The Blockchain Technology for Secure and Smart Applications across Industry Verticals, Advances in Computers, vol. 121, pp. 211–226. Elsevier (2021)
2. Alam, M.T., Chowdhury, S., Halder, R., Maiti, A.: Blockchain Domain-Specific Languages: Survey, Classification, and Comparison. In: 2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021. pp. 499–504. Ieee (2021). https://doi.org/10.1109/Blockchain53845.2021.00076
3. Ali, O., Ally, M., Clutterbuck, Dwivedi, Y.: The state of play of blockchain technology in the financial services sector: A systematic literature review. International Journal of Information Management **54**, 102199 (2020). https://doi.org/10.1016/j.ijinfomgt.2020.102199
4. Allen, C.: The Path to Self-Sovereign Identity (2016), `http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html` Accessed: 03/2022
5. Allen, F.E.: Control Flow Analysis. In: Proceedings of a Symposium on Compiler Optimization. p. 1–19. Association for Computing Machinery, New York, NY, USA (1970). https://doi.org/10.1145/800028.808479
6. Allombert, V., Bourgoin, M., Tesson, J.: Introduction to the Tezos Blockchain. In: 2019 International Conference on High Performance Computing and Simulation (HPCS). pp. 1–10 (2019). https://doi.org/10.1109/hpcs48598.2019.9188227
7. lisa analyzer: GoLiSA Analyzer - Non-Determinism Sources. `https://github.com/lisa-analyzer/go-lisa/blob/soap22/go-lisa/src/main/resources/for-analysis/nondeterm_sources.txt` Accessed: 10/2022 (2021)
8. lisa analyzer: GoLiSA Analyzer Repository. `https://github.com/lisa-analyzer/go-lisa` Accessed: 10/2022 (2021)
9. lisa analyzer: PyLiSA Analyzer Repository. `https://github.com/lisa-analyzer/pylisa` Accessed: 10/2022 (2021)
10. lisa analyzer: RustLiSA Analyzer Repository. `https://github.com/lisa-analyzer/rust-lisa` Accessed: 10/2022 (2021)
11. lisa analyzer: MichelsonLiSA Analyzer Repository. `https://github.com/lisa-analyzer/michelson-lisa` Accessed: 10/2022 (2022)
12. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings of the Thirteenth EuroSys Conference. EuroSys '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3190508.3190538

13. Antonopoulos, A.M.: Mastering Bitcoin: Programming the Open Blockchain. O'Reilly, 2nd edn. (2017)
14. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly (2018)
15. Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts (SoK). In: Maffei, M., Ryan, M. (eds.) Principles of Security and Trust. pp. 164–186. Springer Berlin Heidelberg, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
16. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001, special Issue on Second issue of experimental software and toolkits (EST)
17. Bakos, Y., Halaburda, H.: Tradeoffs in Permissioned vs Permissionless Blockchains: Trust and Performance. Ssrn (Feb 2021). https://doi.org/10.2139/ssrn.3789425
18. Baralla, G., Pinna, A., Corrias, G.: Ensure traceability in european food supply chain by using a blockchain system. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 40–47 (2019). https://doi.org/10.1109/WETSEB.2019.00012
19. Bashir, I.: Mastering blockchain. Packt Publishing Ltd (2017)
20. Bau, G., Miné, A., Botbol, V., Bouaziz, M.: Abstract Interpretation of Michelson Smart-Contracts. In: Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 36–43. Soap 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520313.3534660
21. Bayer, D., Haber, S., Stornetta, W.S.: Improving the Efficiency and Reliability of Digital Time-Stamping. In: Capocelli, R., De Santis, A., Vaccaro, U. (eds.) Sequences II. pp. 329–334. Springer New York, New York, NY (1993). https://doi.org/10.1007/978-1-4613-9323-8_24
22. Beller, M., Hejderup, J.: Blockchain-based software engineering. In: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results. p. 53–56. ICSE-NIER '19, IEEE Press (2019). https://doi.org/10.1109/ICSE-NIER.2019.00022
23. Benini, A., Gambini, M., Migliorini, S., Spoto, F.: Power and Pitfalls of Generic Smart Contracts. In: 2021 Third International Conference on Blockchain Computing and Applications (BCCA). pp. 179–186 (2021). https://doi.org/10.1109/bcca53669.2021.9657048
24. Berdik, D., Otoum, S., Schmidt, N., Porter, D., Jararweh, Y.: A Survey on Blockchain for Information Systems Management and Security. Inf. Process. Manag. **58**(1), 102397 (2021). https://doi.org/10.1016/j.ipm.2020.102397
25. Bernardo, B., Cauderlier, R., Claret, G., Jakobsson, A., Pesin, B., Tesson, J.: Making Tezos Smart Contracts More Reliable with Coq. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 60–72. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_5
26. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. In: Formal Methods. FM 2019 International Workshops. pp. 368–379. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_28
27. Bernardo, B., Cauderlier, R., Pesin, B., Tesson, J.: Albert, An Intermediate Smart-Contract Language for the Tezos Blockchain. In: Financial Cryptography and Data Security. pp. 584–598. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-54455-3_41
28. Blog, T.G.: Go Developer Survey 2021 Results (2021), https://go.dev/blog/survey2021-results, Accessed: 09/2022

29. Bosu, A., Iqbal, A., Shahriyar, R., Chakraborty, P.: Understanding the motivations, challenges and needs of Blockchain software developers: a survey. Empir. Softw. Eng. **24**(4), 2636–2673 (2019). https://doi.org/10.1007/s10664-019-09708-7
30. Bozzetti, M., Olivieri, L., Spoto, F.: Cybersecurity Impacts of the Covid-19 Pandemic in Italy. In: Italian Conference on Cybersecurity - ITASEC. vol. 2940, pp. 145–155. CEUR Workshop Proceedings (CEUR-WS.org) (2021), http://ceur-ws.org/Vol-2940/paper13.pdf Accessed: 10/2022
31. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis, University of Guelph (2016)
32. Buchman, E.: Byzantine Fault Tolerant State Machine Replication in Any Programming Language. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. p. 546. PODC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293611.3338023
33. Buterin, V.: Ethereum Whitepaper (2013), https://ethereum.org/en/whitepaper/ Accessed: 10/2022
34. Chabbi, M., Ramanathan, M.K.: A Study of Real-World Data Races in Golang. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 474–489. Pldi 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523720
35. Chatterjee, K., Goharshady, A.K., Pourdamghani, A.: Probabilistic Smart Contracts: Secure Randomness on the Blockchain. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 403–412 (2019). https://doi.org/10.1109/bloc.2019.8751326
36. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theoretical Computer Science **777**, 155–183 (2019). https://doi.org/10.1016/j.tcs.2019.02.001
37. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Programming Languages and Systems. pp. 3–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_2
38. Chess, B., West, J.: Secure programming with static analysis. Addison-Wesley Professional (2007)
39. Chittoda, J.: Mastering Blockchain Programming with Solidity: Write production-ready smart contracts for Ethereum blockchain with Solidity. Packt Publishing Ltd (2019)
40. Clarisó, R., Cortadella, J.: The Octahedron Abstract Domain. In: Static Analysis. pp. 312–327. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27864-1_23
41. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
42. Commercio.network: Commercio.network - White Paper (2022), https://commercio.network/project/ Accessed: 07/2022
43. ConsenSys: ConsenSys Tokens. https://github.com/ConsenSys/Tokens
44. Corporation, M.: Common Weakness Enumeration - CWE-829 (2020), https://cwe.mitre.org/data/definitions/829.html, Accessed: 10/2022
45. CosmWasm: CosmWasm Documentation, https://docs.cosmwasm.com/docs/1.0/ Accessed: 07/2022
46. Cousot, P.: Types as Abstract Interpretations. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 316–331. ACM Press (1997). https://doi.org/10.1145/263699.263744
47. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. Acm (1977). https://doi.org/10.1145/512950.512973

48. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
49. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
50. Crosara, M., Olivieri, L., Spoto, F., Tagliaferro, F.: Fungible and non-fungible tokens with snapshots in Java. Cluster Computing pp. 1–18 (2022). https://doi.org/10.1007/s10586-022-03756-3
51. Crosara, M., Olivieri, L., Spoto, F., Tagliaferro, F.: Re-engineering ERC-20 Smart Contracts with Efficient Snapshots for the Java Virtual Machine. In: 2021 Third International Conference on Blockchain Computing and Applications (BCCA). pp. 187–194 (2021). https://doi.org/10.1109/bcca53669.2021.9657047
52. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (oct 1991). https://doi.org/10.1145/115372.115320
53. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.): Structured Programming. Academic Press Ltd., Gbr (1972)
54. Database, N.V.: CVE-2018-10299 Detail. `https://nvd.nist.gov/vuln/detail/cve-2018-10299` Accessed: 10/2022 (2018)
55. Demange, D., Jensen, T., Pichardie, D.: A Provably Correct Stackless Intermediate Representation for Java Bytecode. In: Ueda, K. (ed.) Programming Languages and Systems. pp. 97–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_8
56. Denning, D.E.: A Lattice Model of Secure Information Flow. Commun. ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056
57. Denning, D.E., Denning, P.J.: Certification of Programs for Secure Information Flow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
58. Dennis, J.B., Horn, E.C.V.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966). https://doi.org/10.1145/365230.365252
59. Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., Hierons, R.: Smart contracts vulnerabilities: a call for blockchain software engineering? In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 19–25 (2018). https://doi.org/10.1109/iwbose.2018.8327567
60. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Not. **35**(6), 26–36 (jun 2000). https://doi.org/10.1145/352029.352035
61. Dexaran: ERC223 Token Standard. `https://github.com/Dexaran/ERC223-token-standard` Accessed: 10/2022 (2021)
62. Ding, M., Li, P., Li, S., Zhang, H.: Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. In: Evaluation and Assessment in Software Engineering. p. 321–328. EASE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3463274.3463351
63. Donovan, A.A., Kernighan, B.W.: The Go programming language. Addison-Wesley Professional (2015)
64. Emrath, P.A., Padua, D.A.: Automatic Detection of Nondeterminacy in Parallel Programs. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging. p. 89–99. Padd '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/68210.69224

65. Entrinken, W., Shirley, D., Evans, J., Sachs, N.: EIP-721: ERC-721 Token Standard, Ethereum Improvement Proposals, no. 721. https://eips.ethereum.org/EIPS/eip-721 Accessed: 10/2022 (2018)
66. Eos.io: EOS.IO White Paper, https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md Accessed: 07/2022
67. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean Formulas for the Static Identification of Injection Attacks in Java. In: Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 130–145. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_10
68. Ethereum: Solidity v0.8.0 Breaking Changes. https://docs.soliditylang.org/en/v0.8.2/080-breaking-changes.html Accessed: 10/2022 (2020)
69. F., V., Buterin, V.: EIP-20: ERC-20 Token Standard, Ethereum Improvement Proposals, no. 20. https://eips.ethereum.org/EIPS/eip-20 Accessed: 10/2022 (2017)
70. Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. Theor. Comput. Sci. **631**, 43–72 (2016). https://doi.org/10.1016/j.tcs.2016.04.001
71. Ferrara, P., Burato, E., Spoto, F.: Security Analysis of the OWASP Benchmark with Julia. In: Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy, January 17-20, 2017. CEUR Workshop Proceedings, vol. 1816, pp. 242–247. CEUR-WS.org (2017), http://ceur-ws.org/Vol-1816/paper-24.pdf Accessed: 10/2022
72. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static Analysis for Dummies: Experiencing LiSA. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6. Soap 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460946.3464316
73. Ferrara, P., Olivieri, L., Spoto, F.: Tailoring Taint Analysis to GDPR. In: Privacy Technologies and Policy. pp. 63–76. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-02547-2_4
74. Ferrara, P., Olivieri, L., Spoto, F.: BackFlow: Backward Context-Sensitive Flow Reconstruction of Taint Analysis Results. In: Verification, Model Checking, and Abstract Interpretation. pp. 23–43. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_2
75. Ferrara, P., Olivieri, L., Spoto, F.: Static Privacy Analysis by Flow Reconstruction of Tainted Data. Int. J. Softw. Eng. Knowl. Eng. **31**(7), 973–1016 (2021). https://doi.org/10.1142/s0218194021500303
76. Foschini, L., Gavagna, A., Martuscelli, G., Montanari, R.: Hyperledger Fabric Blockchain: Chaincode Performance Analysis. In: ICC 2020 - 2020 IEEE International Conference on Communications (ICC). pp. 1–6 (2020). https://doi.org/10.1109/icc40277.2020.9149080
77. Foundation, E.: EVM Documentation, https://ethereum.org/en/developers/docs/evm/opcodes Accessed: 07/2022
78. Foundation, E.: The Merge, https://ethereum.org/en/upgrades/merge/ Accessed: 07-2022
79. Foundation, P.S.: Python3 Documentation - Objects, https://docs.python.org/3/reference/datamodel.html#objects-values-and-types Accessed: 07/2022
80. Gabriel Barros, P.G.: EIP-1822: Diamonds, Multi-Facet Proxy, no. 1822. https://eips.ethereum.org/EIPS/eip-1822 Accessed: 10/2022 (2019)
81. Gagandeep Singh, Markus Püschel, M.V.: ETH Library for Numerical Analysis. http://elina.ethz.ch Accessed: 10/2022 (2018)
82. Gamage, H.T.M., Weerasinghe, H.D., Dias, N.G.J.: A Survey on Blockchain Technology Concepts, Applications, and Issues. SN Comput. Sci. **1**(2), 114 (2020). https://doi.org/10.1007/s42979-020-00123-0

83. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/sp.1982.10014
84. Goguen, J.A., Meseguer, J.: Unwinding and Inference Control. In: 1984 IEEE Symposium on Security and Privacy. pp. 75–75 (1984). https://doi.org/10.1109/sp.1984.10019
85. Goodman, L.: Tezos Whitepaper (2014), available at https://tezos.com/whitepaper.pdf
86. Google: Go Official Documentation - Integer Overflow, https://go.dev/ref/spec#Integer_overflow Accessed: 10/2022
87. Google: Gofuzz. https://github.com/Google/gofuzz Accessed: 10/2022 (2018)
88. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30**(3-4), 165–190 (1989). https://doi.org/10.1080/00207168908803778
89. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Tapsoft '91. pp. 169–192. Springer Berlin Heidelberg, Berlin, Heidelberg (1991). https://doi.org/10.1007/3-540-53982-4_10
90. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. Proc. ACM Program. Lang. **2**(Oopsla) (oct 2018). https://doi.org/10.1145/3276486
91. Guo, H., Yu, X.: A survey on blockchain technology and its security. Blockchain: Research and Applications **3**(2), 100067 (2022). https://doi.org/10.1016/j.bcra.2022.100067
92. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. Journal of Cryptology **3**, 99–111 (1991). https://doi.org/10.1007/bf00196791
93. Hassan, S., De Filippi, P.: Decentralized autonomous organization. Internet Policy Review **10**(2), 1–10 (2021). https://doi.org/10.14763/2021.2.1556
94. Honnef, D.: Staticcheck Website, https://staticcheck.io/, Accessed: 09/2022
95. Arrojado da Horta, L.P., Santos Reis, J., Pereira, M., Melo de Sousa, S.: WhylSon: Proving your Michelson Smart Contracts in Why3. arXiv e-prints (2020). https://doi.org/10.48550/arXiv.2005.14650
96. Hotmoka – Blockchain and IoT with Smart Contracts in Java. https://www.hotmoka.io (2021)
97. Hotmoka: Hotmoka ERC-20 Comparison Implementation. https://github.com/Hotmoka/hotmoka/tree/erc20-comparison Accessed: 10/2022 (2021)
98. Hotmoka: Hotmoka On-chain Verification Implementation. https://github.com/Hotmoka/hotmoka/tree/wtsc21 Accessed: 10/2022 (2021)
99. Hotmoka: Takamaka's ERC-20 Implementation. https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/tokens/ERC20.java Accessed: 10/2022 (2021)
100. Hotmoka: Takamaka's ERC-721 Implementation. https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/tokens/ERC721.java Accessed: 10/2022 (2021)
101. Hotmoka: Translation of ERC-20 with Snapshot in Takamaka. https://github.com/Hotmoka/hotmoka/blob/master/io-hotmoka-examples/src/main/java/io/hotmoka/examples/tokens/ERC20OZSnapshot.java Accessed: 10/2022 (2021)
102. Hyperledger: Hyperledger Fabric Documentation, https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html#what-is-hyperledger-fabric Accessed: 10/2022
103. Hyperledger: HyperLedger Fabric Documentation, https://hyperledger-fabric.readthedocs.io/en/release-2.4/ Accessed: 07/2022
104. Hyperledger: ERC-20 Token Scenario (2021), https://github.com/hyperledger/fabric-samples/tree/main/token-erc-20#erc-20-token-scenario Accessed: 10/2022
105. Hyperledger: ERC-721 Token Scenario (2021), https://github.com/hyperledger/fabric-samples/tree/main/token-erc-721#erc-721-token-scenario Accessed: 10/2022

106. Ignite: Ignite Documentation, https://docs.ignite.com/ Accessed: 07/2022
107. Inc, T.: What is Tendermint (2022), https://docs.tendermint.com/master/introduction/what-is-tendermint.html Accessed: 01/2022
108. Inc, T.: What is Tendermint: A Note on Determinism (2022), https://docs.tendermint.com/master/introduction/what-is-tendermint.html#a-note-on-determinism Accessed: 01/2022
109. InterWasm DAO: An ERC-20 Token Contract, https://github.com/InterWasm/cw-contracts/tree/e7b81397bd48a33711557c270abe53e1266baf04/erc20 Accessed 07/2022
110. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
111. Jiang, B., Liu, Y., Chan, W.: Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 259–269 (2018). https://doi.org/10.1145/3238147.3238177
112. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts. In: Network and Distributed System Security Symposium. pp. 1–12 (2018)
113. Kar, A.K., Navin, L.: Diffusion of blockchain in insurance industry: An analysis through the review of academic and trade literature. Telematics and Informatics **58**, 101532 (2021). https://doi.org/10.1016/j.tele.2020.101532
114. Khedker, U.P., Karkare, B.: Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In: Hendren, L. (ed.) Compiler Construction. pp. 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78791-4_15
115. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM **19**(7), 385–394 (jul 1976). https://doi.org/10.1145/360248.360252
116. Koscina, M., Lombard-Platet, M., Cluchet, P.: PlasticCoin: An ERC20 Implementation on Hyperledger Fabric for Circular Economy and Plastic Reuse. In: IEEE/WIC/ACM International Conference on Web Intelligence - Companion Volume. p. 223–230. WI '19 Companion, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3358695.3361107
117. Kwon, J.: Tendermint: Consensus without Mining (2014), available at https://tendermint.com/static/docs/tendermint.pdf Accessed: 10/2022
118. Kwon, J., Buchman, E.: Cosmos whitepaper (2019), https://v1.cosmos.network/resources/whitepaper Accessed: 02/2022
119. kzhry: Chaincode Analyzer, https://github.com/hyperledger-labs/chaincode-analyzerc Accessed: 12/2021
120. Lai, E., Luo, W.: Static analysis of integer overflow of smart contracts in ethereum. In: Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy. p. 110–115. ICCSP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377644.3377650
121. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (jul 1982). https://doi.org/10.1145/357172.357176
122. Lehmann, D., Pradel, M.: Finding the Dwarf: Recovering Pecise Types from WebAssembly Binaries. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 410–425. Pldi 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523449
123. Leng, J., Zhou, M., Zhao, J.L., Huang, Y., Bian, Y.: Blockchain Security: A Survey of Techniques and Research Directions. IEEE Transactions on Services Computing pp. 1–1 (2020). https://doi.org/10.1109/tsc.2020.3038641

124. Ligo: LIGO Documentation, https://ligolang.org/docs/intro/introduction Accessed: 07/2022
125. Liks: Liks GitHub Repository, https://github.com/LiskHQ/lisk-core#lisk-core Accessed: 07/2022
126. Lv, P., Wang, Y., Wang, Y., Zhou, Q.: Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis. In: IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021. pp. 1–7. Ieee (2021). https://doi.org/10.1109/iscc53001.2021.9631249
127. Mahdi H. Miraz, M.A.: Blockchain Enabled Smart Contract Based Applications: Deficiencies with the Software Development Life Cycle Models. Baltica Journal **33**, 101–116 (2020)
128. Malhotra, A., O'Neill, H., Stowell, P.: Thinking strategically about blockchain adoption and risk mitigation. Business Horizons **65**(2), 159–171 (2022). https://doi.org/10.1016/j.bushor.2021.02.033
129. Marchesi, L., Marchesi, M., Pompianu, L., Tonelli, R.: Security checklists for ethereum smart contract development: patterns and best practices. arXiv e-prints (2020). https://doi.org/10.48550/arXiv.2008.04761
130. Marchesi, L., Marchesi, M., Tonelli, R.: Abcde–agile block chain dapp engineering. Blockchain: Research and Applications **1**(1), 100002 (2020). https://doi.org/https://doi.org/10.1016/j.bcra.2020.100002
131. Martino, R., Cilardo, A.: Designing a SHA-256 processor for blockchain-based IoT applications. Internet of Things **11**, 100254 (2020). https://doi.org/doi.org/10.1016/j.iot.2020.100254
132. Merkle, R.C.: Protocols for Public Key Cryptosystems. In: 1980 IEEE Symposium on Security and Privacy. pp. 122–122 (1980). https://doi.org/10.1109/sp.1980.10006
133. Meyer, B.: Soundness and Completeness: With Precision. BLOGCACM, https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext Accessed: 02/2023 (2019)
134. mgechev: Revive, https://github.com/mgechev/revive Accessed: 12/2021
135. Min, T., Cai, W.: A Security Case Study for Blockchain Games. In: 2019 IEEE Games, Entertainment, Media Conference (GEM). pp. 1–8 (2019). https://doi.org/10.1109/gem.2019.8811555
136. Min, T., Wang, H., Guo, Y., Cai, W.: Blockchain Games: A Survey. In: 2019 IEEE Conference on Games (CoG). pp. 1–8 (2019). https://doi.org/10.1109/cig.2019.8848111
137. Miné, A., Ouadjaout, A., Journault, M.: Design of a modular platform for static analysis. In: 9th Workshop on Tools for Automatic Program Analysis (2018)
138. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
139. Mudge, N.: EIP-2535: Diamonds, Multi-Facet Proxy, no. 2535. https://eips.ethereum.org/EIPS/eip-2535 Accessed: 10/2022 (2020)
140. Mühle, A., Grüner, A., Gayvoronskaya, T., Meinel, C.: A survey on essential components of a self-sovereign identity. Computer Science Review **30**, 80–86 (2018). https://doi.org/10.1016/j.cosrev.2018.10.002
141. Murphy, G.: Secure Go Website, https://securego.io/, Accessed: 09/2022
142. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008), https://bitcoin.org/bitcoin.pdf Accessed: 10/2022
143. Neo: Neo White Paper, https://docs.neo.org/v2/docs/en-us/basic/whitepaper.html Accessed: 07/2022
144. Network, C.: Cosmos SDK (2020), https://cosmos.network/sdk Accessed: 10/2022
145. Nist: CVE-2021-41135 Detail (2021), https://nvd.nist.gov/vuln/detail/CVE-2021-41135 Accessed: 02/2022

146. Oliva, G.A., Hassan, A.E., Jiang, Z.M.: An Exploratory Study of Smart Contracts in the Ethereum Blockchain Platform. Empirical Software Engineering **25**(3), 1864–1904 (2020). https://doi.org/10.1007/s10664-019-09796-5
147. Olivieri, L., Spoto, F., Tagliaferro, F.: On-Chain Smart Contract Verification over Tendermint. In: 5th Wokshop on Trusted Smart Contracts (WTSC'21). Lecture Notes in Computer Science, vol. 12676, pp. 333–347. Springer (2021). https://doi.org/10.1007/978-3-662-63958-0_28
148. Olivieri, L., Tagliaferro, F., Arceri, V., Ruaro, M., Negrini, L., Cortesi, A., Ferrara, P., Spoto, F., Talin, E.: Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report. In: Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 23–29. SOAP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520313.3534658
149. OpenZeppelin: ERC-20 Docs. https://docs.openzeppelin.com/contracts/4.x/api/token/erc20 Accessed: 10/2022
150. OpenZeppelin: ERC-721 Docs, https://docs.openzeppelin.com/contracts/4.x/api/token/erc721 Accessed 07/2022
151. OpenZeppelin: OpenZeppelin's ERC-721 Implementation. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol Accessed: 10/2022 (2018)
152. OpenZeppelin: OpenZeppelin's ERC-20 Implementation. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol Accessed: 10/2022 (2019)
153. OpenZeppelin: OpenZeppelin's ERC-20 Implementation with Snapshot. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Snapshot.sol Accessed: 10/2022 (2019)
154. Parasaram, N.: Mythril Wiki Page. https://github.com/ConsenSys/mythril/wiki Accessed: 10/2022 (2020)
155. Parizi, R.M., Amritraj, Dehghantanha, A.: Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. In: Blockchain – ICBC 2018. pp. 75–91. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94478-4_6
156. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 912–915. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3264591
157. Parr, T.: ANTLR Website, https://www.antlr.org/ Accessed: 07/2022
158. Patrick, C.: Principles of Abstract Interpretation. MIT Press Academic (2021)
159. Polkadot: PolkaDot - Whitepaper (2020), https://polkadot.network/PolkaDotPaper.pdf Accessed: 10/2022
160. Popov, S., Lu, Q.: IOTA: feeless and free. IEEE Blockchain Technical Briefs (2019), https://blockchain.ieee.org/technicalbriefs/january-2019/iota-feeless-and-free Accessed: 10/2022
161. Popper, N.: A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. The New York Times (2016), june 17th
162. Porru, S., Pinna, A., Marchesi, M., Tonelli, R.: Blockchain-Oriented Software Engineering: Challenges and New Directions. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 169–171 (2017). https://doi.org/10.1109/icse-c.2017.142
163. Praitheeshan, P., Pan, L., Zheng, X., Jolfaei, A., Doss, R.: Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems. Information Sciences **579**, 150–166 (2021). https://doi.org/10.1016/j.ins.2021.08.007

164. Qasse, I.A., Abu Talib, M., Nasir, Q.: Inter Blockchain Communication: A Survey. In: Proceedings of the ArabWIC 6th Annual International Conference Research Track. ArabWIC 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3333165.3333167
165. Reis, J.S.: Tezla Analyzer Tests - Repository. `https://github.com/joaosreis/tezla/tree/main/tests` Accessed: 10/2022 (2021)
166. Reis, J.S., Crocker, P., de Sousa, S.M.: Tezla, an intermediate representation for static analysis of Michelson smart contracts. arXiv e-prints (2020). https://doi.org/10.48550/arXiv.2005.11839
167. Reitwießner, C., Johnson, N., Vogelsteller, F., Baylina, J., Feldmeier, K., Entriken, W.: EIP-165: ERC-165 Standard Interface Detection: Ethereum Improvement Proposals, no. 165. `https://eips.ethereum.org/EIPS/eip-165` Accessed: 10/2022 (2018)
168. Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathemathical Society **74**, 358–366 (1953). https://doi.org/10.1090/s0002-9947-1953-0053041-6
169. Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. Mit Press (2020)
170. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global Value Numbers and Redundant Computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 12–27. Popl '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/73560.73562
171. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). https://doi.org/10.1109/jsac.2002.806121
172. Saberi, S., Kouhizadeh, M., Sarkis, J., Shen, L.: Blockchain technology and its relationships to sustainable supply chain management. International Journal of Production Research **57**(7), 2117–2135 (2019). https://doi.org/10.1080/00207543.2018.1533261
173. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Verification, Model Checking, and Abstract Interpretation. pp. 25–41. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_2
174. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons (1998)
175. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley Professional, fourth edn. (2014)
176. Sedlmeir, J., Buhl, H.U., Fridgen, G., Keller, R.: The Energy Consumption of Blockchain Technology: Beyond Myth. Business & Information Systems Engineering **62**(6), 599–608 (2020). https://doi.org/10.1007/s12599-020-00656-x
177. Seijas, P.L., Thompson, S.J., McAdams, D.: Scripting smart contracts for distributed ledger technology. IACR Cryptol. ePrint Arch. **2016**, 1156 (2016)
178. Sharir, M., Pnueli, A., et al.: Two approaches to interprocedural data flow analysis. Courant Institute of Mathematical Sciences, New York University (1978)
179. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M. (ed.) Logic Based Program Synthesis and Transformation. pp. 71–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-45013-0_7
180. sivachokkapu: ReviveCC, `https://github.com/sivachokkapu/revive-cc` Accessed: 12/2021
181. SmartContractSecurity: Smart Contract Weakness Registy - SWC-112 (2020), `https://swcregistry.io/docs/SWC-112`, Accessed: 10/2022
182. SmartPy: SmartPy Documentation, `https://smartpy.io/docs/` Accessed: 07/2022
183. SmartPy: SmartPy Reference - Constants vs Expressions, `https://smartpy.io/reference.html` Accessed: 07/2022

184. Software, group Università Ca' Foscari in Venice, S.V.S.: LiSA's structure, https://unive-ssv.github.io/lisa/structure/ Accessed: 09/2022
185. SonarSource: SonarSource Go static code analysis, https://rules.sonarsource.com/go, Accessed: 09/2022
186. Spoto, F.: The Julia Static Analyzer for Java. In: 23rd Static Analysis Symposium (SAS'16). Lecture Notes in Computer Science, vol. 9837, pp. 39–57. Springer, Edinburgh, UK (September 2016). https://doi.org/10.1007/978-3-662-53413-7_3
187. Spoto, F.: Enforcing Determinism of Java Smart Contracts. In: 4th Wokshop on Trusted Smart Contracts (WTSC'20). Lecture Notes in Computer Science, vol. 12063, pp. 568–583. Springer, Kota Kinabalu, Malaysia (February 2020). https://doi.org/10.1007/978-3-030-54455-3_40
188. Spoto, F.: A Java Framework for Smart Contracts. In: Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11599, pp. 122–137. Springer (2019). https://doi.org/10.1007/978-3-030-43725-1_10
189. Team, V.: Vyper Documentation, https://vyper.readthedocs.io/en/stable/ Accessed: 07/2022
190. Tendermint: What is Tendermint (2020), https://docs.tendermint.com/master/introduction/what-is-tendermint.html Accessed: 10/2022
191. Tendermint: Tendermint Core - Determinism. https://github.com/tendermint/spec/blob/master/spec/abci/abci.md#determinism Accessed: 10/2022 (2020)
192. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. p. 9–16. WETSEB '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3194113.3194115
193. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A Survey of Smart Contract Formal Specification and Verification. ACM Comput. Surv. **54**(7) (jul 2021). https://doi.org/10.1145/3464421
194. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: effective taint analysis of web applications. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 87–97. Acm (2009). https://doi.org/10.1145/1542476.1542486
195. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 67–82. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243780
196. Varela-Vaca, A.J., Quintero, A.M.R.: Smart Contract Languages: A Multivocal Mapping Study. ACM Comput. Surv. **54**(1) (jan 2021). https://doi.org/10.1145/3423166
197. Vyper ERC20 Implementation. https://github.com/vyperlang/vyper/blob/master/examples/tokens/ERC20.vy Accessed: 10/2022
198. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in ethereum smart contracts. Proc. ACM Program. Lang. **3**(OOPSLA), 189:1–189:29 (2019). https://doi.org/10.1145/3360615
199. Wang, X., Li, J., Zhang, X.: A semantic-based smart contract defect detection general platform. In: 2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA). pp. 34–37 (2022). https://doi.org/10.1109/AEECA55500.2022.9918903

200. WebAssembly: WebAssembly GitHub - Non determinism, https://github.com/WebAssembly/design/blob/main/Nondeterminism.md Accessed: 07/2022
201. WebAssembly: WebAssembly GitHub - Number types, https://webassembly.github.io/spec/core/syntax/types.html#number-types Accessed: 07/2022
202. WebAssembly: WebAssembly Introduction, https://webassembly.github.io/spec/core/intro/introduction.html Accessed: 07/2022
203. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular Specification and Verification of Go Programs. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 367–379. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_17
204. Yakovenko, A.: Solana: A new architecture for a high performance blockchain v0.8.13, https://solana.com/solana-whitepaper.pdf Accessed: 07/2022
205. Yamashita, K., Nomura, Y., Zhou, E., Pi, B., Jun, S.: Potential Risks of Hyperledger Fabric Smart Contracts. In: 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 1–10 (2019). https://doi.org/10.1109/iwbose.2019.8666486
206. Ye, C., Li, G., Cai, H., Gu, Y., Fukuda, A.: Analysis of Security in Blockchain: Case Study in 51%-Attack Detecting. In: 5th International Conference on Dependable Systems and Their Applications. pp. 15–24 (2018). https://doi.org/10.1109/dsa.2018.00015
207. Zhang, X., Guo, X., Zeng, Z., Liu, W., Guo, Z., Chen, Y., Chen, S., Yin, Q., Yang, M.: Argus: A Fully Transparent Incentive System for Anti-Piracy Campaigns. In: The 40th Int. Symp. on Reliable Distributed Systems (SRDS) (2021), https://www.microsoft.com/en-us/research/publication/argus-a-fully-transparent-incentive-system-for-anti-piracy-campaigns/ Accessed: 10/2022
208. Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., Gu, D.: Smartshield: Automatic smart contract protection made easy. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 23–34 (2020). https://doi.org/10.1109/SANER48275.2020.9054825
209. Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: A survey. International journal of web and grid services **14**(4), 352–375 (2018). https://doi.org/10.5555/3292946.3292948
210. Zou, W., Lo, D., Kochhar, P.S., Le, X.B.D., Xia, X., Feng, Y., Chen, Z., Xu, B.: Smart Contract Development: Challenges and Opportunities. IEEE Transactions on Software Engineering **47**(10), 2084–2106 (2021). https://doi.org/10.1109/tse.2019.2942301