Darwinian Code Optimisation

Michail Basios

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

January 18, 2019

I, Michail Basios, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Programming is laborious. A long-standing goal is to reduce this cost through automation. Genetic Improvement (GI) is a new direction for achieving this goal. It applies search to the task of program improvement. The research conducted in this thesis applies GI to program optimisation and to enable program optimisation. In particular, it focuses on automatic code optimisation for complex managed runtimes, such as Java and Ethereum Virtual Machines.

We introduce the term Darwinian Data Structures (DDS) for the data structures of a program that share a common interface and enjoy multiple implementations. We call them Darwinian since we can subject their implementations to the survival of the fittest. We introduce ARTEMIS, a novel cloud-based multi-objective multi-language optimisation framework that automatically finds optimal, tuned data structures and rewrites the source code of applications accordingly to use them. ARTEMIS achieves substantial performance improvements for 44 diverse programs. ARTEMIS achieves 4.8%, 10.1%, 5.1% median improvement for runtime, memory and CPU usage.

Even though GI has been applied succesfully to improve properties of programs running in different runtimes, GI has not been applied in Blockchains, such as Ethereum. The code immutability of programs running on top of Ethereum limits the application of GI. The first step of applying GI in Ethereum is to overcome the code immutability limitations. Thus, to enable optimisation, we present PROTEUS, a state of the art framework that *automatically* extends the functionality of smart contracts written in Solidity and makes them upgradeable. Thus, allowing developers to introduce alternative optimised versions of code (*e.g.,* code that consumes less gas), found by GI, in newer versions.

Impact

The main impact of this thesis is the introduction of two novel optimisation frameworks (ARTEMIS and PROTEUS) that *automatically* improve the performance (execution time, memory consumption, and CPU usage) and enable program optimisations for users' programs with minimal effort. We also provide both frameworks as open-source services (`http://www.darwinianoptimiser.com`), allowing the research community to interact with them. We show how the code optimisation, a process time-consuming, brittle, expensive with unclear benefits for many projects can be done automatically using Genetic Improvement.

With this thesis, we extend the existing literature in different ways. First, we introduced the term "Darwinian Data Structures" and we formalised the Darwinian data structure selection and optimisation problem $\mathbf{DS^2}$. We showed that it is possible to improve the performance of large code bases automatically by discovering and optimising sub-optimal parts of the code cheaply, fast and without affecting the functionality of the system.

Second, we tackled and provided a solution to the problem of code immutability of programs running on distributed environments, such as the Blockchain, by proposing upgradeable smart contracts. We introduced PROTEUS, the first framework, to best of our knowledge, that provides automatic transformation of Ethereum smart contracts such that they can be upgraded after published on the Blockchain. Thus, enabling the possibility of program optimisation and the application of Genetic Improvement techniques to improve the performance of Blockchain systems. By applying GI on smart contracts, we can improve the "Gas" consumption on the Ethereum Blockchain by providing alternative code implementations, similarly to

the approach that ARTEMIS follows. PROTEUS also allows other GI techniques to be applied such as automatic bug fixing automatic test generation and prioritisation.

Broader usage of PROTEUS on the Blockchain, and subsequently the application of GI, has the potential to lead to faster contract execution and less number of duplicate contracts on the Ethereum Blockchain. This means that the size of the Blockchain will be smaller and the transactions can be processed faster. A smaller and faster Blockchain has a significant impact on the global energy consumption because this means that the overall energy consumption of the network is going to be smaller as well; an Ethereum network may contain thousands of nodes across different countries. Similarly, ARTEMIS can be applied to optimise different metrics to a variety of programs and systems, not limited to Blockchain. Thus, ARTEMIS can have broader applicability, and many can benefit by its optimisations.

Finally, we provide clear research ideas of how our work can be extended and the problems that still need to be solved for future work. We also provide information on how new researchers can start working on their research by using our research findings and our open-source optimisation frameworks.

Acknowledgements

I would like to thank all the people that helped me during my PhD, either directly with my research or indirectly with their advice and support.

First, I thank my supervisor Earl Barr for his guidance, support, and for having an excellent collaboration. I would also like to thank my second supervisor Ilya Sergey for his suggestions. Special thanks to all the colleagues from UCL for their generous feedback and fun discussions we had during our meetings and Microsoft for the support of this research.

I am very grateful to Graham Barrett for his help and guidance. I would also like to especially thank my wonderful friends Dr. Leslie Kanthan, Dr. Lingbo Li, Dr. Fan Wu, Dr. David Martinez, Dr. Wei Chen, Petros Kyrkillis, Rhyan Barret and Aaliaan Khan for making my experience in London great. I would also like to thank my Edinburgh supervisor Stratis Viglas for his guidance and my amazing colleagues Mihaela Dragomir, Aurora Constantin, Fabian Nagel, Andreas Chatzistergiou and Maria Nadejde. I cannot ignore my great undergraduate colleagues and friends Alexandros Spathoulas, Panagiotis Karavidas, Nikos Anyfantis, Konstantinos Chimos, Giorgios Christopoulos, Panagiotis Papantonakis and Antonis Antonis for their true friendship and help.

Last, I am very thankful to my parents, for their unconditional support to my decisions and my beautiful sister for always being there for me.

Publications

- Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. "Darwinian Data Structure Selection". In Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)
- Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. "Optimising darwinian data structures on Google guava." In International Symposium on Search Based Software Engineering, pp. 161-167. Springer, Cham, 2017. Best Challenge Paper Award.
- Michail Basios, Ilias Sergey, Earl T. Barr. "Proteus: A framework for upgradeable Ethereum Smart Contracts". Under Review.

Contents

1 Introduction **17**

1.1 Objectives of the Research 19

1.2 Contributions 20

1.3 Organisation of the PhD Thesis 22

2 Literature Review **24**

2.1 Search-Based Software Engineering (SBSE) 25

2.1.1 Requirements Engineering 25

2.1.2 Effort Estimation 26

2.1.3 Software Product Line 26

2.1.4 Software Testing 27

2.1.5 Other Areas 27

2.1.6 SBSE Industrial Applications 28

2.2 Genetic Improvement 29

2.2.1 Code Representation 32

2.2.2 Search-Based Parameter Tuning 34

2.2.3 Improvement using Genetic Programming 39

2.2.4 Patch-Based Genetic Improvement 40

2.3 Data Structure Selection and Tuning 41

2.3.1 Java Virtual Machine 43

2.3.2 Java Collection Framework 44

2.3.3 Tradeoffs in Collection Implementations 45

2.3.4 Empirical Rigorous Performance Evaluation 47

2.3.5 Data Structure Optimisation and Bloat 48
2.4 Blockchain 51
2.4.1 Consensus Layer 54
2.4.2 Types of Blockchain Systems 55
2.4.2.1 Permissionless Blockchain 55
2.4.2.2 Permissioned Blockchain 56
2.4.2.3 Hybrid Blockchain 57
2.4.3 Ethereum Blockchain 57
2.4.3.1 Ethereum Account Types 59
2.4.3.2 A Smart Contract Example. 60
2.4.3.3 Ethereum Virtual Machine 61
2.4.3.4 Contract State Transitions / Transactions 64
2.4.3.5 Message calls 65
2.4.3.6 Function Dispatch 65
2.4.3.7 Types of Message Calls 66
2.4.4 Security issues with Solidity smart contracts 69
2.4.5 Research Findings on Ethereum Smart Contracts 71

3 Darwinian Data Structure Selection 74
3.1 Motivating example 77
3.2 Darwinian Data Structure Selection and Tuning 78
3.3 Artemis 80
3.3.1 Darwinian Data Structure Store 81
3.3.2 Discovering Darwinian Data Structures 82
3.3.3 Code Transformations 83
3.3.4 Search Based Parameter Tuning 85
3.3.5 Deployability 86
3.4 Evaluation 86
3.4.1 Corpus 87
3.4.2 Experimental Setup 90
3.4.3 Research Questions and Results Analysis 91

3.4.4 Optimising Google Guava library using ARTEMIS 102
3.5 Threats to Validity 104
3.6 Summary 105

4 Upgradeable Ethereum Smart Contracts 106

4.1 Immortal Bugs 112
4.2 Motivating Example 113
4.3 Approach 116
4.3.1 Delegating Calls to Other Contracts 118
4.3.2 Solidity Syntax 122
4.3.3 Proteus Rewriting Rules 123
4.3.4 Mutable Implementation Mode 124
4.3.4.1 Trampoline Rewriting Rules 125
4.3.4.2 Rewriting Rules for Contract `C'` 130
4.3.4.3 Mutable Implementation Code Transformation Example 133
4.3.5 Mutable Interface Mode 135
4.3.5.1 Rewriting Rules for Mutable Interface 138
4.4 Implementation 141
4.4.1 Deployability 143
4.4.2 Deployment Process 143
4.4.3 Summary 144

5 Conclusions and Future Work 145

5.1 Conclusions 145
5.2 Future Work 147

Bibliography 149

List of Figures

2.1 Flowchart of a Genetic Algorithm. 29

2.2 Example of the crossover genetic operation. Sections of trees' structure of the selected parents are swapped to create children [1]. . . . 32

2.3 Selected mutation operators by Deep Parameter Optimisation framework [2]. 37

2.4 Java Collection Framework. 45

2.5 Memory layout of an `ArrayList` with integers for a 32-bit JVM. . 46

2.6 When an `ArrayList` is used with default parameters, it allocates memory for 10 entries. 47

2.7 An instance illustrating the problem of reporting misleading metrics: the 'best' method is shown on the left and the empirical rigorous method is shown on the right (Figure borrowed from [3]). 50

2.8 Blockchain is a chain of blocks of transactions linked by hash pointers. 51

2.9 Transaction Example: In blockchain, an account with a key pair (public key and private key) is representing a wallet for submitting transactions. 52

2.10 Cryptocurrency Market Capitalisation. Source: `www.coinmarketcap.com`, 27/10/2017. 53

2.11 Comparison between public, consortium and private Blockchain. Table taken from [4]. 56

2.12 Account types on Ethereum [5]. External account is controlled by a private key and does not contain code. Contract is controlled by EVM code. 58

2.13 Number of Ethereum addresses in 31 October 2017. (Source: `https://etherscan.io/chart/address`) 59
2.14 When a transaction is accepted on the Blockchain, each mining node change their Blockchain state. 61
2.15 The EVM is a simple stack-based architecture [5]. 62
2.16 Internal fields of a transaction. 64
2.17 Function dispatch example in smart contracts. 66
2.18 Visualisation of function calls between contracts. 68

3.1 Example of how ARTEMIS maps the extracted darwinian data structures and its parameters to a search-based optimisation problem. . . 78
3.2 System Architecture of ARTEMIS. 80
3.3 DDS in the Java Collections API. 82
3.4 Process of generating the uniformly selected at random Github corpus. First, a Github project is selected randomly, then if it contains a maven build system, it compiles and its tests run succesfully it is added in the corpus. 88
3.5 **Answers *RQ2***. Description. 93
3.6 **Answers *RQ2***. Description. 95
3.7 Best execution time of uniformly selected GitHub programs. The median value is 95.4% and mean is 94.7%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.8. 96
3.8 Best memory consumption of the uniformly selected GitHub programs. The median value is 89.1% and mean is 86.8%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.6. 97
3.9 Best CPU usage of the uniformly selected GitHub programs. The median value is 5.1% and mean is 8%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.5. 98

3.10 Optimal solutions with large improvement in at least one measure. . 102

4.1 State of the art update mechanism for smart contracts. 107
4.2 Different types of clients that interact with Ethereum Smart Contracts. 108
4.3 MultiSig bug. The contract owner is meant to be defined using the `init` function only by the contract owner. However, the hacker managed to change the owner of the contract by calling the `init(owners)` function of the `AbstractWallet` contract, using the fallout function of the `Wallet` contract. 112
4.4 Smart contract transformation. 116
4.5 Trampoline and Contract interaction for the mutable implementation mode. After the `Trampoline` is deployed, its code is immutable. Each call to a function of the `Trampoline` is delegated to one of the upgradeable contracts. The user can deploy a new contract that will have the same function signatures, but the internal code can be modified. 125
4.6 EVM uses an intenal immutable virtual function table as a lookup table of functions for resolving function calls in a dynamic/late binding manner. 136
4.7 Dynamic virtual function mechanism used by the `Trampoline`. The logic of how EVM dispatches functions internally is exposed to the developer through the code contained in the `Trampoline`. 138
4.8 System Architecture of Proteus. 142
4.9 Contract Deployment. The Trampoline contract and the initial user's contract is given as an input by the user. PROTEUS generates two transactions: a) The first one publishes the Trampoline contract on the Blockchain b) the second one updates the address that the Trampoline should forward the calls. 144

List of Tables

3.1 Data structure groups. 83
3.2 DaCapo projects. #Star, #Loc are the number of stars and line of codes respectively. All these subjects are retrieved from the official Dacapo Benchmark page on 11th Jan 2017. 88
3.3 Subject projects studied in this research. #Star, #Loc, #Test, and Coverage(%) are the number of stars, line of code, number of tests, and the line coverage ratio, respectively. All these subjects are retrieved through GitHub on 11th Jan 2017. 89
3.4 Hardware characteristics. 90
3.5 DDS changes for optimal solutions across all measures. 100

4.1 Table with mutability modes of PROTEUS. Mutable Implementation 124
4.2 Trampolinify Contract Rule. PROTEUS injects a set of meta variables and helper functions in the body of the Trampoline contract. 127
4.3 Constructor Injection rule. Initialise meta variables when contract is created. 128
4.4 Upgradeable Functions rule. **public** and **external** functions are forwarding the call to the upgradeable contracts. 129
4.5 Upgradeable Functions with Returns. **public** and **external** functions are forwarding the call to the upgradeable contracts. The return value is assigned to the corresponding meta-variable. 130
4.6 Unexposed Functions rule (F1). **internal** and **private** functions are not exposed through the Trampoline. Those functions can be updated with the new version of contract C. 131

4.7 The same set of of meta-variables that were added in the `Trampoline` are also added in the code of contract `C'`. 131
4.8 PROTEUS allows calls to the contract C' only from the Trampoline address. This rule forbids any other calls. 132
4.9 PROTEUS allows calls to the contract `C'` only from the `Trampoline` address. This rule forbids any other calls. 132
4.10 Table with mutability modes of PROTEUS. Mutable Interface. 136
4.11 Dynamic Function Dispatch Rule. PROTEUS injects a set of meta variables and helper functions in the body of the Trampoline contract. 139
4.12 Constructor Expansion rule. The `vtable` is populated with the default function definitions of contract `C`. 140
4.13 Constructor Expansion rule. The `vtable` is populated with the default function definitions of contract `C`. 141

Chapter 1

Introduction

> *"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."*
>
> — Donald E. Knuth [6]

Under the immense time pressures of industrial software development, developers are heeding one part of Knuth's advice: they are avoiding premature optimisation. Indeed, developers appear to be avoiding optimisation altogether and neglecting the "critical 3%". When selecting data structures from libraries, in particular, they tend to rely on defaults and neglect potential optimisations that alternative implementations or tuning parameters can offer. This, despite the impact that data structure selection and tuning can have on application performance and defects. A similar pattern is observed in the development of programs (smart contracts) for Blockchain systems (e.g., Ethereum). Developers tend to rely on existing default smart contract templates published on the Blockchain, without having the possibility of upgrading and using more gas-efficient (gas is the execution fee compensating the computing resources of miners for running smart contracts) or with fewer bugs alternative implementations.

Considering four examples. Selecting an implementation that creates unnecessary temporary objects for the program's workload [7]. Selecting a combination of Scala data structures that scaled better, reducing execution time from 45 to 1.5 minutes [8]. Avoiding the use of poor implementation, such as those in the Oracle bug database that leak memory [9]. Selection of under-optimised smart contracts that cost more gas than necessary, overcharging its creators and users [10].

Optimisation is time-consuming and challenging, especially on large code bases with multiple conflicting non-functional properties (such as execution time and memory consumption). Manually optimising non-functional properties, while keeping the functional behaviour of software is challenging because of the enormous space of alternative solutions. Optimisation is also brittle. An optimisation for one version of a program can break or become a de-optimisation in the next release. Another reason developers may avoid optimisation are development fads that focus on fast solutions, like "*Premature Optimisation is the horror of all Evil*" and "*Hack until it works*" [11]. In short, optimisation is expensive and its benefits unclear for many projects. Developers need automated help.

A long-standing goal is to reduce the optimisation cost through automation. We use recent findings in the area of Search-Based Software Engineering as a generic technique for automatically finding optimal solutions for the optimisation problems we define in this thesis. For providing a more scalable and faster framework, we then focus on a more specific subarea of Search-Based Software Engineering that uses Genetic Improvement (Section 2.2). Genetic Improvement (GI) is a new direction for achieving this goal. It has recently received notable awards, demonstrating its acceptance and success within the software engineering community [12, 2, 13, 14, 15, 16, 17]. GI uses optimisation and machine learning techniques, mainly search based, to change and improve existing software automatically. This thesis applies GI to program optimisation and to enable program optimisation. In particular, it focuses on automatic code optimisations for complex managed runtimes, such as Java and Ethereum Virtual Machines.

Most of the existing GI-based optimisation approaches rely on the Plastic

Surgery Hypothesis, which assumes that the solutions exist in the code base. However, good solutions can also be found from external code repositories. To ensure scalability of their approaches, existing frameworks usually modify programs at the 'line' level of granularity. Recently, finer level modifications (at the 'parameter' level of granularity) have shown that better solutions can be found [18]. However, sometimes optimising in a 'higher' granularity level than parameters (for example in 'data structure' level or algorithm level) can be more rewarding than just tuning the parameters.

In this thesis, we investigate how GI can improve existing software by applying code transformantions in a higher than parameter granularity level; *e.g.,* alternative data structures or libraries. Our approach introduces solutions not only from the code base of the subject program, but also from external code repositories. We also provide automatic code transformation such that GI can be applied in systems that limit code upgradeability. More specifically, we investigate the usage of search-based approach for improving *automatically* non-functional properties (execution time, memory consumption, CPU usage and gas-consumption) of users' code and we show its applicability on multiple languages (Java, C++, and Solidity) and different managed runtime systems (Java Virtual Machine or Ethereum Virtual Machine).

Next, we provide an introduction to the objectives of this thesis. We then describe our contributions to the scientific community and provide an overview of the structure of this thesis.

1.1 Objectives of the Research

The primary objective of this research is to help developers perform optimisations and automatic code transformations cheaply and efficiently. We aim to provide two frameworks, one that solves the Darwinian Data Structure Selection and Tuning problem and one that enables program optimisation on Ethereum Blockchain by introducing the notion of upgradeable smart contracts. We also want to provide the foundations of building a third framework that will automatically apply GI to optimise non-functional properties of Blockchain systems. The detailed goals and

objectives of this thesis are as follows:

- Investigate the possibility of automatic extraction of Darwinian Data Structures and its parameters from large code bases.
- Formally define the Darwinian Data Structure selection and tuning problem as a search-based optimisation problem and show its generalisability across different domains and languages.
- Perform a thorough statistical rigorous evaluation of the proposed frameworks.
- Identify existing optimisation limitations of Blockchain systems that provide managed runtimes for executing user's code (smart contracts).
- Transform automatically Ethereum smart contracts such that they can be upgraded, after being published on the Blockchain. This is the crucial first step that enables program optimisation on Ethereum Blockchain. Thus, allowing the use of GI and its significant findings to be applied to the Blockchain.
- Propose a new research area of applying GI to improve non-functional properties of Blockchain distributed systems automatically; *e.g.,* gas consumption, automatic bug fixing or test generation and prioritisation.

1.2 Contributions

The main contribution of this thesis is the introduction of two novel optimisation frameworks (ARTEMIS and PROTEUS) that *automatically* improve the performance (execution time, memory consumption, and CPU usage) and enable program optimisations for users' programs with minimal effort. We also provide both frameworks as open-source services (`http://www.darwinianoptimiser.com`), allowing the research community to interact with them. We show how the code optimisation, a process time-consuming, brittle, expensive with unclear benefits for many projects can be done automatically using Genetic Improvement.

With this thesis, we extend the existing literature in different ways. First, we introduced the term "Darwinian Data Structures" and we formalised the Darwinian

data structure selection and optimisation problem $\mathbf{DS^2}$. We showed that it is possible to improve the performance of large code bases automatically by discovering and optimising sub-optimal parts of the code cheaply, fast and without affecting the functionality of the system.

To provide evidence to the effectiveness of our framework, we conducted an extensive empirical study on 43 projects; a) 30 uniformly selected at random projects, b) 8 popular well-written projects and c) a Benchmark from Dacapo proposed and accepted by the scientific community. We also used Random Testing to further test and validate the effectiveness of the proposed framework. For all 43 subjects, ARTEMIS can successfully find variants that outperform the original for all three objectives. On extreme cases, ARTEMIS discovered 31% improvement on execution time, 70.68% improvement on memory consumption, and 78.86% improvement on CPU usage.

Second, we tackled and provided a solution to the problem of code immutability of programs running on distributed environments, such as the Blockchain, by introducing upgradeable smart contracts. PROTEUS, is the first framework, to best of our knowledge, that provides automatic transformation of Ethereum smart contracts such that they can be upgraded after published on the Blockchain. Thus, enabling the possibility of program optimisation and the application of Genetic Improvement techniques to improve the performance of Blockchain systems. By applying GI on smart contracts, we can improve the "Gas" consumption on the Ethereum Blockchain by providing alternative code implementations, similarly to the approach that ARTEMIS follows. PROTEUS also allows other GI techniques to be applied such as automatic bug fixing automatic test generation and prioritisation.

A broader usage of PROTEUS on the Blockchain and subsequently, the application of GI has the potential to lead to faster contract execution and less number of duplicate contracts on the Ethereum Blockchain. This means that the size of the Blockchain will be smaller and the transactions can be processed faster. A smaller and faster Blockchain has a huge impact on the global energy consumption because this means that the overall energy consumption of the network is going to be smaller

as well; an Ethereum network may contain thousands of nodes across different countries.

Finally, we provide clear research ideas of how our work can be extended and the problems that still need to be solved for future work. We also provide information on how new researchers can start working on their research by using our research findings and our open-source optimisation frameworks.

1.3 Organisation of the PhD Thesis

Initially, we provide an introduction to the code optimisation problems that this thesis tries to solve and point out their importance. Then we focus on the literature review and identify the related scientific work that tried to provide solutions to those problems and what are their advantages and disadvantages.

We introduce ARTEMIS, a novel cloud-based multi-objective optimisation framework that automatically finds optimal, tuned data structures and rewrites applications to use them. We describe the system architecture and present experiments that show how the proposed approach outperforms other provided solutions. Then we focus on code optimisation on top of Blockchain systems, and we introduce PROTEUS, a framework that *automatically* extends the functionality of smart contracts code written in Solidity. We discuss the evaluation of the generated smart contracts and conclude by showing how PROTEUS can be used in practice. We further focus on the Blockchain and present the foundations for a new optimisation framework that improves the gas consumption of Ethereum smart contracts by *automatically* transforming their code to use more efficient implementations of costly external libraries. Finally, we provide a conclusion with the findings of this thesis and potential future research work.

The structure of the rest of the thesis is organised as follows:

Chapter 2: Literature Review: Briefly surveys state-of-the-art related work in genetic improvement, code optimisation, data structure bloat and Blockchain technology.

Chapter 3: Darwinian Data Structure Selection: Presents ARTEMIS, a multi-

objective code optimisation framework that *automatically* selects and optimises the data structures of a project. The Darwinian Data Structure Selection problem is formulated and experiments that show the efficiency of the approach are presented.

Chapter 4 Upgradeable Smart Contracts: Introduces PROTEUS, a framework that *automatically* solves the problem of mutability on Ethereum smart contracts. The architecture of the framework is presented as well as the different transformation modes that are supported. Finally, the transformation rules are defined and potential security issues are discussed.

Chapter 5 Conclusions and Future Work: Concludes the thesis with a discussion on the threats to validity and suggestions for potential future work directions.

Chapter 2

Literature Review

In this chapter, we review the related research areas and findings necessary to establish a foundation for the research undertaken in this thesis. Our research focuses mainly on two areas: 1) code optimisation by automatic selection and tuning of data structures and 2) enabling of program optimisation on programs running on top of Blockchain through program transformation.

First, we present the literature review related to SBSE and its applicability for automatic code optimisatiom. We analyse the data structures of the programming languages (Java, C++, Solidity) that we optimise in this thesis. We then detail the characteristics of their managed runtimes and how developers' code practices can affect the performance (execution time, memory consumption, CPU usage) of a program. We dive into the Java Virtual machine's details and show how the usage of its collection API can affect the performance of a program. Next, we focus on how to correctly measure the performance of complex programs running on top of a managed runtime using statistical rigorous performance evaluation methods.

Our research focuses on data structure selection and tuning. To understand why the optimisation framework selects various data structures against others and how it tunes them, we need to present the related research work on Data Structure optimisation and code performance improvement (Section 2.3). To evaluate correctly the performance of the optimised code and the statistical importance of our experiments, we present the related work on the area of rigorous statistical Java performance evaluation and various profiling techniques (Section 2.3.4).

Then, we describe the essential characteristics of Blockchain systems, and more particularly, the Ethereum Blockchain. We investigate the properties of Ethereum Virtual Machine (EVM) and its programming language (Solidity) and analyse how they can affect the applicability of automatic code optimisation.

2.1 Search-Based Software Engineering (SBSE)

In Software engineering, we can view many activities as optimisation problems where the goal is to find better solutions based on one or multiple evaluating functions. Search-Based Software Engineering (SBSE) [19, 20] is a sub-field that contains that body of work that uses search-based optimisation algorithms (i.e. Hill Climbing [21], Simulated Annealing [22], Random Search, Genetic Algorithms [23]) to solve a software engineering problem. It has been a sub-field of Software Engineering since 2011 [24] with very successful and generic applicability. Previous work has applied SBSE in a variety of software engineering problems, such as requirements engineering [25, 26, 27], software effort estimation [28, 29, 30], performance and energy optimisation [2, 31], system architecture design [32], software testing [33], code transplantation [12], bug fixing and maintenance [34].

2.1.1 Requirements Engineering

Requirements engineering is an essential part of the Software Engineering process [35]. It refers to the process of defining, documenting and maintaining requirements [36] in the engineering design process. A problem that project managers come during this process is what to select among requirements and how to prioritise them. The aim is to meet the demands of stakeholders (*e.g.,* minimise the cost of the budget) and maximise the value of the delivered software product [37] (*e.g.,* maximise the revenue). We can formulate both the selection and prioritisation of requirements as search problems. Thus, we can apply SBSE successfully to automatically provide good and robust solutions which allow more natural adaption of the software process to requirement changes. SBSE can optimise *multiple* conflicting objectives and find solutions that will find a good trade-off between them. As a result, SBSE helps the project managers to take better and more informed decisions when designing the

next release.

2.1.2 Effort Estimation

When developing software, it is essential for an organisation to predict accurately the effort required for the project to be delivered under the detailed budget [38, 29]. Underestimation may lead to not successful delivery of the project while overestimation may lead to unnecessary bigger budget allocation [39, 40]. Existing research [29, 30, 41, 42] has shown that we can utilise SBSE for the effort estimation problem. By applying a multi-objective evolutionary approach that tries to maximise the estimation accuracy but at the same time minimise uncertainty, it is possible to build more robust estimation models than humans, who tend to provide over-optimistic estimations [43]. In a similar context, we can apply SBSE for estimating the cost of a software project. Kirsopp *et al.* [44] improve the cost estimations by using search to predict unknown attributes of a project; they search near neighbour projects that share similar values for the known attributes.

2.1.3 Software Product Line

Another area that we can apply SBSE successfully is the Software Product Line (SPL) [45]. The idea is to identify and extract differences in features from a given number of software products that provide some common functionalities. We can consider SPL problems as complex search optimisation problems because of the large search space that the high variability of various products (expressed by their feature models) create. The goal of SPL engineering is to optimally manage the extraction, analysis, evolution and application of the feature model and the SPL architecture combined with the products constructed from them [46]. After converting the SPL problem to an optimisation problem, SBSE can identify such features and combine them to construct similar software products that can benefit from the commonalities shared by all features. SBSE finds optimal (or near optimal) choices of products. SBSE is useful for the SPL problem also because of its multi-objective optimisation nature; many of the problems need to find a right balance of multiple competing and conflicting software engineering concerns.

2.1.4 Software Testing

Software testing is a critical component of software development because it provides information about the quality of the system and how well it is developed to meet the required specifications. A project may contain various levels of testing, such as Unit Testing, Integration Testing, Acceptance Testing and System Testing. Because of this variety of levels of testing and the different requirements, many techniques have been applied (including SBSE) for generating, executing and verifying tests [47, 48, 49]. The two most popular applications of SBSE in testing involve the optimisation of the test data generation [50, 51] and the optimisation of the test data selection and prioritization [52].

Optimising the test data generation is a process difficult and slow because of the vast search space that the large pool of possible software inputs generates. A good search strategy is needed to find a combination of inputs that will simultaneously satisfy specific coverage criteria. SBSE is applied successfully in the test generation problem because it can find near-optimal solutions in a vast search space.

Optimising test data selection and prioritisation is necessary because sometimes it is not possible to execute all the given tests of an application. The goal is to select a set of test cases that achieve the same (or nearly the same) level of test adequacy as the entire set. SBSE is applied to prioritise the tests and select that subset such that it minimises the cost of testing while maximising the coverage criteria.

In this thesis, we use Unit Testing and Regression Testing to verify the correctness of the altered program that the proposed frameworks (ARTEMIS) optimise. Regression testing is useful as it provides confidence that changes to the source code do not break the existing behaviour of the program. By using regression testing, which has been shown to be effective in many software testing practices [53], we minimise the threat to validity and correctness of a program after the proposed framework alters it.

2.1.5 Other Areas

Other work has applied SBSE to automatically search for re-factoring solutions that may improve various code quality metrics, such as readability, efficiency, adapt-

ability and extensibility [54]. SBSE has also been used during the design phase of software engineering, for architecture design, software clustering and software re-factoring [55], with different objectives and formulations. Except for software processes, SBSE has been applied successfully to optimise the software itself. Meta-heuristic search has been applied to search for optimisation sequences in source code [56]. Other work used SBSE to parallelise better tasks on supercomputers [57].

Souza *et al.* [58] studied thoroughly the effectiveness and efficiency of SBSE and how it compares with human-provided solutions. For different problems such as next release, test case selection and the work-group formation, those studies showed the superiority of SBSE solutions. More specifically, professional software engineers compared the quality between SBSE solutions and human-provided solutions and showed that the quality of SBSE solutions is, in most cases, better or similar to the human-provided ones [58]. Also, the SBSE solutions are usually more consistent, and there is much less effort to obtain them.

2.1.6 SBSE Industrial Applications

All those applications of SBSE have made it mature enough that many industrial applications have started to appear [59, 60] and other new promising tools are under open-source development and testing. EvoSuite [61] is a real-world tool, used by many open-source projects and companies like Google, that automatically generates test suites, satisfying a coverage criterion for programs written in Java. Sapienz [33] is another industrial application, used by Facebook, that automatically generates tests for Android applications. Sapienz generates tests that optimise multi-objective goals; minimising the length of test sequences while maximising coverage and fault revelation. μSCALPEL [12] is a tool that aims to automatically migrate one piece of code from one system into another, entirely unrelated, system (a process name “software transplantation"). μSCALPEL successfully auto-transplanted the H.264 video encoding functionality from the x264 system to the VLC media player automatically, showing that code transplantation is possible and can have wider adaption. Another tool implemented by Li *et al.* [25] that handles uncertainty during the optimisation of the next release problem has been adopted by Microsoft Visual

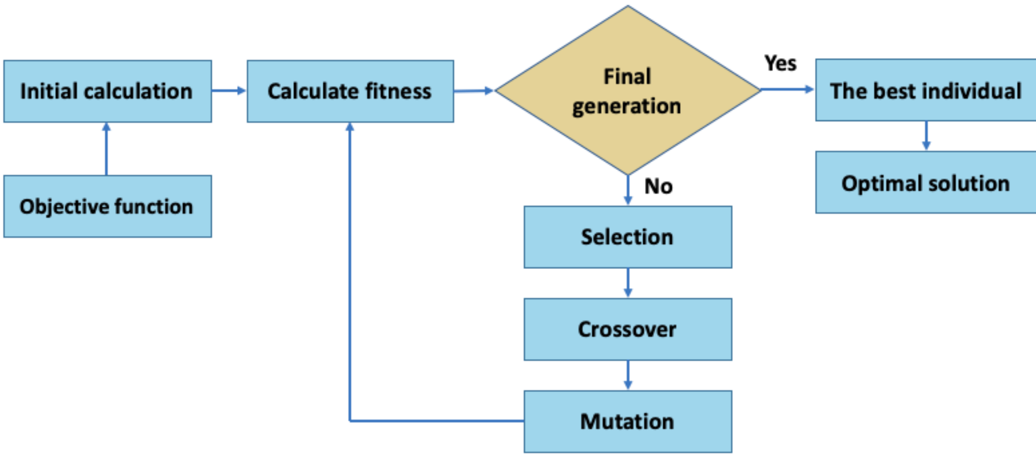


Figure 2.1: Flowchart of a Genetic Algorithm.

Studio, under closed beta test and has shown promising results. The key strength of those proposed tools that makes them successful is the fact that they can *automatically* solve very complex problems that are difficult to find the exact solution in reasonable time deterministically.

2.2 Genetic Improvement

Genetic Improvement (GI) is a new area of SBSE that focuses on improving one or more properties of software by using Genetic/Evolutionary Algorithms [62]. A genetic algorithm (GA) is a metaheuristic method inspired by the process of natural selection. Genetic algorithms belong to the larger class of evolutionary algorithms (EA) and can be applied to solve constrained and unconstrained optimisation problems. Genetic algorithms can provide high-quality solutions to optimisation and search problems by relying on bio-inspired operators such as mutation, crossover and selection [63]. In this thesis, we use genetic improvement to improve the performance of the software automatically and to scale on larger code bases.

In a genetic algorithm, given a population of candidate solutions (called individuals) to an optimisation problem, the intention is to evolve that population towards better solutions. Each candidate solution has a set of properties (its chromosomes) which can be mutated and modified [23]. Typically, the algorithm starts with a randomly produced population of individuals, which is called a generation. Then,

it evolves the population iteratively (each iteration is a generation) until specific conditions are met (as shown in Figure 2.1). For every generation, the algorithm evaluates a fitness function for each individual; the fitness represents the value of the objective function in the optimisation problem being solved (i.e., the execution time of a program). The algorithm then stochastically selects the fittest individuals from the current population and alters them (through mutation or crossover) to form a new generation. The new generation of potential candidate solutions produces the population in the next iteration of the algorithm. The algorithm finishes when it finds the desired solution specified by the user or when it has produced the maximum number of generations.

Although Genetic Improvement has a long history that traces back to the foundations of computer science, the original explicit usage of Genetic Improvement term was in 1995 (as surveyed by Petke *et al.* [62]) by Ryan and Walsh [64], who used genetic programming to convert serial programs to parallel ones automatically. The GI area became very popular when Harman *et al.* [65] introduced the GISMOE challenge to the research community. With this challenge, the authors noticed the demand and need in the software engineering area of optimising non-functional and functional properties of software in a multi-objective setting. They also pointed out that some SBSE techniques [19] can automatically or semi-automatically fulfill this demand. Additionally, they noted that GI has the potential for a broad adaption because of it is generality and adaptability. At the same time, they mentioned open challenges such as "human in the loop" optimisation and high dimensional Pareto surfaces visualisation.

GI has had a wider adaption because it can find better solutions in a vast search space, which comes from relaxing the restrictions on program correctness. Optimisation of software indicates that some of its aspects should be changed to improve some functionality and some should remain the same, such that the software does not work differently. For example, GI may automatically fix a bug but should guarantee that it does not introduce new bugs because of its changes. To ensure that the changes in the program do not break any functionality of the original pro-

gram, existing work [66] used only semantic-preserving transformations. However, semantic-preserving transformations limit the search space and the likelihood of finding better solutions. Besides, this is not guaranteed always to provide correct solutions, as Orlov *et al.* [66] showed; when improving the existing Java bytecode by using a semantic-preserving crossover, they found out that GI generated faulty individuals.

We can apply GI successfully with outstanding optimisation results, prominently in problems that can trade-off constraints on the functional attributes for better performance. Such an illustration is the GI-modified shader simplification software that obtained 67% reduction in runtime by having slightly lower image fidelity [67]. Still, such use cases that allow this tradeoff are limited. Thus, we need techniques to check the preserving of equivalent functionality when making changes to a program.

Most of the empirical work in GI uses software testing to verify if the same software functionalities remain after applying GI. The concept is that if the set of test cases is all the possible test cases, then the functional equivalence and test equivalence is similar. However, this is not possible as the number of test cases can be unlimited in theory. Thus, most of the current research relaxes the notion of equivalence to a finite set of test cases. It assumes that if the modified, by GI, software passes those test cases then the functionality does not change.

Genetic Improvement is not limited to problems that optimise non-functional properties [68, 69, 70] of software such as execution time, memory consumption, CPU usage, and energy consumption. It has also been applied successfully to optimise the functionality or correctness of the software; finding or fixing the bugs automatically [71, 72, 73]. Most of these approaches rely on the Plastic Surgery Hypothesis [74], which assumes that the solutions exist in the code base. More specifically, Plastic Surgery Hypothesis states that we can use fragments that exist already in the source to construct new code. So, in practice, there is no need to generate new code, but to effectively cut and paste the code from other parts of the program.

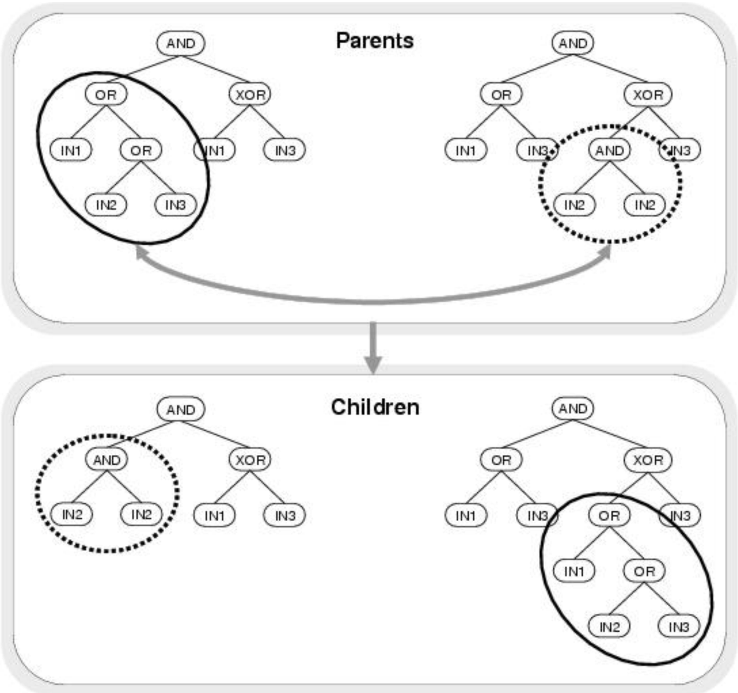


Figure 2.2: Example of the crossover genetic operation. Sections of trees' structure of the selected parents are swapped to create children [1].

2.2.1 Code Representation

The way the source code is represented or how a problem is formulated dictates what options are available when we apply GI on a program under optimisation. Usually, developers face a challenge when deciding the format that they will represent the code because this choice may limit or improve the possibility of finding better solutions. It is not clear to the developer which representation to follow, and a solid understanding of the problem is necessary. The most common representations of source code used when applying GI are: 1) abstract syntax trees, 2) machine code and 3) patches.

When evolving the source code, developers usually will keep copies of the code with the changes they want to make and then evaluate it based on the chosen objective functions. The copies that improve the most the non-functional or functional properties of interest are given as the final solutions to the problem. Typically, those changes happen to the abstract syntax tree (AST) representation of the code. Each node of an AST is either a function with child nodes or a terminal node. GI makes

changes to the AST by changing or rearranging the nodes of the AST (Figure 2.2). Then it parses back the AST to its source code. A mutation happens by changing a single node with an equivalent node that preserves the syntax. Syntactically equivalent nodes have the same input (from children nodes) and output (to their parent nodes) data types. A crossover occurs when two or more nodes are rearranged together to form new nodes; in the simplest case, a crossover selects a node in both branches of the AST and swaps them (Figure 2.2).

Other work has used GI to evolve programs represented directly as machine code [66, 75]. Orlov *et al.* [66] evolved the programs directly in their Java bytecode format, without any intermediate representation. Their approach relies on the notion of compatible crossover, which provides correct programs by conducting compatibility checks on the source and target bytecode sections. By evolving sequences of annotated bytecode, they claim that they can apply more straightforward crossover operators than using branches or types. Schulte *et al.* [75] also applied GI to software represented as assembly code to repair defects while maintaining the desired behaviour. They illustrated their approach on Java bytecode and x86 assembly representation of programs that target embedded software; the aim is to optimise energy and memory efficiency. They evolved the programs by permutating, erasing and repeating the set of instructions in the compiled code, which each represents a single statement from the source code. They also stated the benefits of assembly-level repair over source code level repairs and pointed out the ability to repair applications written in multiple different languages, and the ability to repair bugs that were before intractable, even on non-trivial programs.

Applying GI to improve the software may cause many code changes, and sometimes the optimisation process may be slow and not scalable. That is especially true when GI uses the complete program representation because it generates a huge search space. Thus, in most cases, it cannot scale to more extensive programs. Typically, just a few changes to the software may improve the software significantly. Much work focused on how GI can converge and try to find solutions only on predefined parts of the code, minimising the number of statements that it needs to search.

Essentially, GI can follow a process to evolve a program which is similar to the approach developers follow to produce patches. Developers usually provide patches with small fixes on particular functionalities of the code. By applying such evolving patches to software, we only need to keep a single copy of the original program, resulting in significant memory reduction. Accordingly, GI can apply automatic patch fixing, a method which is called Patch-Based Genetic Improvement.

Barr et al. [74] claimed that we could further improve the optimisation process by using existing code bases to generate many such patches automatically. GenProg [76] followed such a patch-based approach, instead of a full AST representation. They represent patches as a sequence of edits to the AST, where each edit is a tuple consisting of an operation and node numbers. When estimating the cost of automatic bug fixing on the cloud, the authors observed that ASTs copies of the code that had the bugs fixed were of much bigger size than human-written patches and also very memory consuming; 1.7 GB for each node in the cloud. In their experiments, the human-written patches were no more than 50 lines, meaning that two variants were identical in all other parts of the code except those 50 lines.

There is a diverse collection of applications that have used Genetic Improvement. Examining the genetic improvement approaches used in those applications, we categorise them into the three following categories: a) Parameter tuning for software improvement, b) genetic programming and c) patch-based genetic improvement, *etc*.

2.2.2 Search-Based Parameter Tuning

Another area of SBSE that has attracted much attention is the Search Based Parameter Tuning. The idea is that, usually, software systems come with various parameters exposed which can be tuned to improve the system's performance for different evaluation metrics. Manually finding those parameters is a time-consuming process with no guaranteed results because of the huge number of combinations that the developer has to try. Hence, we can improve the system's performance by searching for the optimal values of those parameters automatically. To further enhance the performance of the system and identify optimisation opportunities, researchers applied several techniques to discover more implicit parameters and tune them

altogether.

Previous work has applied successfully automatic parameter tuning to compilers. Parameter tuning can be applied relatively easily to a compiler because it provides a well-defined set of parameters/flags and their selection/tuning can have immediate performance results. Depending on the objective functions that we want to optimise and the hardware that our application runs, we may choose a specific set of parameters. For example, we may choose a set of parameters that minimises power consumption, when compiling the application to run on embedded systems; similarly to the action that Apple took to deliberately use different optimisation settings for applications running on older phones [77, 78].

Manually finding a right combination of such parameters is a time consuming and laborious task for the developer because of the large number of parameter settings that compilers provide. For example, GCC has at least 54 different optimization techniques that have been implemented into the compilation routines [79]. This yields 2^{54}, or nearly 10^{15} different possible settings. Thus, automatic and efficient solutions are necessary. Stephenson *et al.* [80] used evolutionary algorithms to search the space of compiler heuristics automatically. They obtained an average speedup of 23% (up to 73%) for the applications in their test suite. Furthermore, they created useful generic heuristics by evolving a compiler's heuristic over several benchmarks. Boussaa *et al.* [81] introduced NOTICE, a framework that achieves optimal performance by automatically tuning compiler optimisation options based on user-specified properties. The proposed framework uses multi-objective optimisation techniques to provide solutions with trade-offs between various non-functional properties.

Automatic parameter tuning has also been applied extensively to enhance the performance of Machine Learning models [82, 83, 84, 85]. Tantithamthavorn et al [86] used automated parameter tuning techniques to improve the performance of their Machine Learning algorithms. They managed to improve by 40% the ability of their application to discover faults. Lam *et al.* [82] tuned both the parameters and the structure of neural networks by using an enhanced version of a genetic algorithm;

they implemented a faster version of GA by using a representation with floating-point numbers instead of binary numbers. Tsai *et al.* [87] also tuned the parameters and structure of a neural network by introducing a new hybrid Taguchi-genetic algorithm (HTGA). HTGA approach combines the traditional genetic algorithm (TGA), which has a robust global search capability, with the Taguchi method [88], which can utilise the optimum offspring. Both methods showed, by being tested on a sunspot forecasting application, that the proposed neural networks trained with the proposed GA provide better results than those of traditional feed-forward neural networks, regarding accuracy (fitness values).

Other work [68, 89, 90, 91, 92] showed how parameter tuning can be used to automatically improve the energy consumption of applications. Schulte *et al.* [89] introduced a general post-compilation approach called Genetic Optimisation Algorithm (GOA), which aims to improve measurable non-functional properties of programs that compile to x86 assembly. Their algorithm combines meta-data from profile-guided optimisation, evolutionary computation and mutational robustness and searches for program variants that retain the required functional behaviour while improving non-functional behaviour. They evaluated their proposed framework on eight benchmark applications and achieved 20% less energy consumption on average while 7 out of 8 benchmarks retained the required functional behaviour. Bruce *et al.* [68] used GI to provide more energy efficient mobile applications. More specifically, they reduced energy consumption by up to 25% by using GI to automatically find more energy efficient versions of the MiniSAT Boolean satisfiability solver for three applications.

Often, tuning the parameters exposed by a system does not provide the desired performance improvement. Hence, much work has focused on discovering and extracting implicit parameters from the programs under optimisation and expose them as tunable parameters. We can then use search-based techniques to tune both the explicit and implicit parameters.

The Software Tuning panel for Autonomic Control (STAC) [93] identified and extracted a limited number of implicit parameters by using transition flows of the

Mutation Operators	Changes Between
CRCR – Constant replacement	constants, 0, 1, −1
OAAN – Arithmetic operator	+, -, *, /, %
OAAA – Arithmetic assignment	+=, -=, *=, /=, %=
OCNG – Logical context negation	*expr*, !*expr*
OIDO – Increment/decrement	++x, --x, x++, x--
OLLN – Logical operator	&&, ||
OLNG – Logical negation	*x* op *y*, *x* op !*y*, !*x* op *y*, !(*x* op *y*)
ORRN – Relational operator	>, >=, <, <=, ==
OBBA – Bitwise assignment	&=, |=
OBBN – Bitwise operator	&, |

Figure 2.3: Selected mutation operators by Deep Parameter Optimisation framework [2].

program that were also similar for the shallow parameters. STAC was not fully automatic as it requires initial human effort to characterise shallow parameters. Also, it was limited as it can find only a subset of deep parameters automatically.

Hutter et al. [94] tuned both explicit and implicit parameters of the SPEAR SAT solver. They exposed almost all tunable variables possible, and, they created a huge search space. However, exposing every possible parameter means that some of them may not provide any performance benefits when tuned. Their approach, in the current form, has scalability limitations and to overcome that we need to exclude some of those parameters. One commonly used approach to eliminate such parameters is sensitivity analysis.

Hoffmann et al. [95] used parameter tuning to improve output quality and non-functional properties of applications, whose behaviour changes based on the input workload; a quite typical scenario for many dynamic applications nowadays. They fed their application with various datasets and tuned their parameters based on those datasets. They proposed *PowerDial*, a framework that traces the Pareto-best candidates produced during this training process and interchanges them respectively during the application running. For instance, if for some reason there is a resource shortage, the application dynamically selects another set of parameters (from the Pareto-best ones) that can decrease the quality of the output but simultaneously enabling the application to run and not crash.

Wu et al. [18] proposed the Deep Parameter Optimisation approach, a mutation-

based method [96] that automatically exports "deep" parameters from the program under optimisation, leading to a stream of research work in this area [97, 98, 99, 100, 101, 102, 103]. The proposed framework identifies a number of constant definitions in the source code and turns them to tunable parameters. Those constant definitions are usually fixed-numerical values that either the developers pick randomly or by applying some domain knowledge; 'magic numbers' as named by White *et al.* [104]. The constant definitions are not restricted to primitive data types [105]. Wu et al. [18, 106] added nine C and Java fundamental operator classes (Figure 2.3) to increase the number of exposed tunable parameters. To deal with the huge search space and the scalability constraints, Wu et al. [18] applied a mutation-based sensitivity analysis to automate the process of finding candidate deep-parameters fully. They next applied NSGA-II to seek for optimal values for those parameters that balance the non-functional properties of interest.

Though the idea of exposing additional tunable parameter is similar to our proposed framework, ARTEMIS, their approach did not optimise data structure selection, which can sometimes be more rewarding than just tuning the parameters. Moreover, they applied their approach to a memory management library to benefit that library's clients. The extent of improvement usually depends on how much a program relies on that library. In contrast, ARTEMIS directly applies to the source code of the program, making no assumptions about which libraries the program uses, affording ARTEMIS much wider applicability. Also their approach relies on the Plastic Surgery Hypothesis [74], which assumes that the solutions exist in the code base. Our approach, on the other hand, does not rely on the hypothesis but relies on a set of transformation rules. Our approach can automatically generate these transformation rules from the library code or library documentation exhaustively, therefore the approach guarantees a comprehensive set of transformation rules.

In other work, Manotas et al. [31] extracted implicit data structure parameters and later tuned them to improve the energy consumption of Java applications. Bruce *et al.* [107] exposed deep parameters to improve face detection software that used the Viola-Jones algorithm in OpenCV, by allowing a trade-off between execution

time and classification accuracy. Their results showed that execution time could be decreased by 48% if a 1.80% classification inaccuracy is allowed (compared to 1.04% classification inaccuracy of the initial, unmodified algorithm).

2.2.3 Improvement using Genetic Programming

Many of the work in the Genetic Improvement focuses on automatic source code modifications of the software with the intent to enhance functional and non-functional properties of it. The usual process followed is that we represent the source code in a more understandable and structured format such that we can modify it easily and quickly. Next, we apply Genetic Programming (GP) to guide the selection and creation of code variations. GP can be considered as a hyper-heuristic [108] search methodology. It searches in the space of program variants, and it composes functions or models that solve specific problems that depend on input test cases. The usage of Genetic Programming is essential to address scalability problems that other approaches such as exhaustive search face when the search space is huge.

Langdon *et al.* [109] noted how Genetic Programming could be used successfully to many real-world problems such as machine learning and image processing benchmarks, and program quality improvement while reducing their size. Besides, they presented experiments and revealed how to speed up the search for optimal solutions, by choosing as a seed the current version of the program. They also demonstrated that, in some use cases, a small set of test cases could be good enough to generate the optimal solutions.

Other work has used Genetic Programming for bug fixing. Gao et al. [110] automatically detected and repaired bugs associated with memory leaks. They modelled the memory behaviour (trace memory allocations and usage) of the program by expressing it as a Control Flow Graph. This simplified representation of the program was suitable for this optimisation problem as the focus was only on memory leak related bugs. They assessed their approach on 15 benchmarks that contained memory leaks. They identified 89 memory leaks and managed to fix 28% of them automatically. Goues et al. [76] also researched the bug fixing problem and introduced a more generic approach. They represented the program as an array of statements and

did not adopt the simplified Control Flow Graph representation. They used Genetic Programming for searching the space and guided the bug fixing process by using test cases. After the GP finds a solution (bug fix), they apply a post-processing step (by using a hill-climbing clean-up process) to remove the unnecessary changes to the code.

White et al. [111] used a multi-objective GP algorithm to produce both energy efficient and good quality pseudo-random number generators. In their experiments, they revealed that only $1/4$ of the test cases is enough to generate the Pareto-optimal solutions of their optimisation problem.

Arcuri and White et al. [112, 113] showed that extra techniques, combined with a good seed to guide the search, can be applied to improve the optimisation process of multi-objective algorithms. They recommended the usage of co-evolved test cases to support the maintenance of the programs' semantics. They evaluated their proposed approach on 8 benchmarks and showed how they succeeded to discover a non-trivial optimisation that compilers did not find. Arcuri *et al.* [112, 113] used the same approach of co-evolution for bug fixing. They used genetic programming to evolve a buggy program simultaneously with a set of test cases. The authors claimed that this strategy has no restrictions on the type of bugs that it can fix, under the assumption that the input comes with the source code and a formal specification.

2.2.4 Patch-Based Genetic Improvement

Ackling et al. [114] applied GI to transform buggy applications and produce patches. Before the evolutionary optimisation process begins, the genetic algorithm takes as input a modification table with the possible permitted modifications on the AST representation of the program. They use this guided evolutionary process to generate patches, and next they evaluate them on a set of predefined test cases. In their evaluation, they showed that this approach is much quicker in finding patches than random search; the found patch fixes contained only 1 to 8 alterations on average

Langdon et al. [115] used a similar approach to automatically improve the execution time of complex systems with large code bases. They represented the program as an array of statements, rather than using a predefined table with possible

modifications on the AST representation. They performed simple modifications on the program such as plain inserts, deletes or swapping of statements. They further used a hill climbing process for removing unnecessary changes to the code after the optimisation process finished. In their evaluation, they presented impressive results. Not only their optimised program was 70 faster than the initial one, but also they achieved small semantic improvements.

Petke et al. [116] likewise used code from other SAT solvers (code donors) to obtain a SAT solver that will tackle a specific class of problems. They also noted that GI could automatically improve the code faster than a developer that would provide the best possible code improvement. Plus, they showed that growing the code base affected the final improved version of the SAT solver positively. Bruce et al. [68] used a similar strategy to evolve patches that improve both the execution time and energy consumption. Their evaluation on three medium size programs showed 25% improvement in the energy consumption.

2.3 Data Structure Selection and Tuning

In this thesis, we concentrate on improving the performance (execution time, memory consumption and CPU usage) of programs, by utilising search-based software engineering techniques. More specifically, we focus on enhancing the performance of programs by optimising the process of selection and tuning of their data structures. Hence, we initially describe the essential properties of data structures that affect the most the performance of a program. Then, we present the approach that one should follow to measure correctly the different measures of the program and how to conduct statistical rigorous experiments.

Nowadays, the memory available to the runtime of a programming language has increased significantly; a program can have access to Giga bytes of memory. Programs store a massive amount of data directly in their memory space through the use of data structures. For instance, a program may store the most frequently accessed data in a large cache in memory; usually implemented by a map or set. Handling such notable amounts of data indicates that the program will spend a

significant amount of resources on data management. When investigating what influences the performance of numerous programs written in Java, C#, C++ or other programming languages, we observed that the data structure types and their particular implementations were a vital factor.

The amount of available memory and the approach used to store and access those data affects the performance of an application profoundly. Less memory available to the application may impact its speed negatively. If the application has limited memory and wants to access a significant amount of data, then it will spend much time in reading and writing data from the disk; the disk is slower than accessing data from memory. In other cases, the application may allocate a large amount of memory, but may not use it and thus it remains unutilised, not allowing other applications that could benefit from using it. Hence, we need to find a proper trade-off between memory consumption and other requirements, such as execution time.

Nearly every modern programming language provides the developer a collection framework library with abstract data types for handling groups of data (*e.g.,* Lists, Maps, Trees), hiding the details of the underlying data-structure implementation. Typically, the collection framework is abstract, well-defined, and it gives the developer the possibility to choose the appropriate collection implementation and its default parameters. Finding the appropriate collection and its parameters for every occurrence in the code is a difficult, laborious and often infeasible task [117, 118]. A large number of collection occurrences in programs and its significant number of different parameters construct a big search space.

Manual exploration of the search space is time-consuming and challenging with no guaranteed results. This task is considered even more difficult because of the complexity of understanding what and how a data structure implementation and its parameters affect the performance of the application. Often, the developer's assumption about the advantages of one implementation over the other can be misleading and point to inefficient choices, even in cases that the developer has domain knowledge. Thus, developers need help so they can have access to insights

about the performance impact of their data structure selections as well as automatic help for better selection and tuning.

2.3.1 Java Virtual Machine

To demonstrate how we can improve the performance of a program using GI through ARTEMIS, we used applications written in Java, as it is a language that provides a well-defined set of collections with the most commonly used data structures. Hence, for the rest of this chapter, we will analyse the most common characteristics of collections that affect the performance of applications written in Java. These collections, such as ArrayList, HashMap and HashSet, are known as the Java Collection Framework (JCF).

Java is a general-purpose, object-oriented programming language which was designed with the primary goal to be platform independent. A Java program is compiled to Java bytecode and can run on different platforms, without the need of re-compilation. The compiled bytecode is executed on top of a Java Virtual Machine (JVM), which provides operating system style functionalities. Example of such provided functionalities are memory management through garbage collection, automated exception handling, synchronisation, threading and just in time compilation.

Java was initially considered a slow programming language, when compared to languages that compile to native code, but eventually gained a lot of performance improvement with the introduction of Just-in-time compilation (JIT) and other dynamic-based performance optimisations. By using the JIT compiler, the Java Virtual Machine continuously analyses the code being executed and identifies the hottest parts (such as function calls or loops) of the code. It then compiles those parts to native code when the speedup that the compilation provides outweighs the overhead of compiling that code. After the JIT compiler has recompiled the code, it usually runs faster within that specific platform.

Identifying how data structures affect the performance of a program becomes very challenging when it runs on a complex managed runtime environment, such as the Java Virtual Machine (JVM). JVM performs various complex optimisations and features which are not obvious to the developer. Thus, there are a number of

sources of non-determinism in JVM (*e.g.,* JIT, thread scheduling, garbage collection) that affect the overall performance of a program and make it difficult to properly benchmark it; in contrast to compiled programming languages such as C++ with more stable performance. Hence, it may sometimes not be apparent to the developer how a particular code that he/she wrote affects the performance of the program.

Because of this lack of easy understanding of the performance impact that data structures have, developers tend to use only a subset of the available data structures that they feel familiar with. Also they use data structures that are considered to work well in most general use cases, without however thinking the potential benefits that other data structures can have. In real-world Github open-source applications (as we describe in Section 3.4), we noticed the extensive usage of one implementation over the others; *e.g.,* `ArrayList`, which is considered to work very well in most cases, has a much bigger use when compared with `LinkedList`. We observed that there are many optimisation opportunities by using other data structures and by tuning their parameters which are neglected or not discovered.

2.3.2 Java Collection Framework

The Java collections framework (JCF) is a set of classes and interfaces that implement commonly reusable container-like data structures [119]. The framework (Figure 2.4) contains a) interfaces: abstract data types that represent collections, b) implementations of the collection interfaces and c) polymorphic algorithms: for useful computations on objects that implement collection framework; *e.g.,* sorting, searching, composition, shuffling and so on.

Collections do not need to be assigned a certain capacity when instantiated, since they come with some default values; *e.g.,* the initial size of an `ArrayList` is 10. However, most collections provide parameters that allow the developer to change the default capacity value. A collection can grow and shrink in size automatically when items are added or removed; *e.g.,* an `ArrayList` will increase its size by 1.5 if the number of items exceeds the available pre-allocated memory.

There are 4 main types of Java collections available `List`, `Map`, `Set` and `Queue` (Figure 2.4). For each of those data structures there exist usually more than one

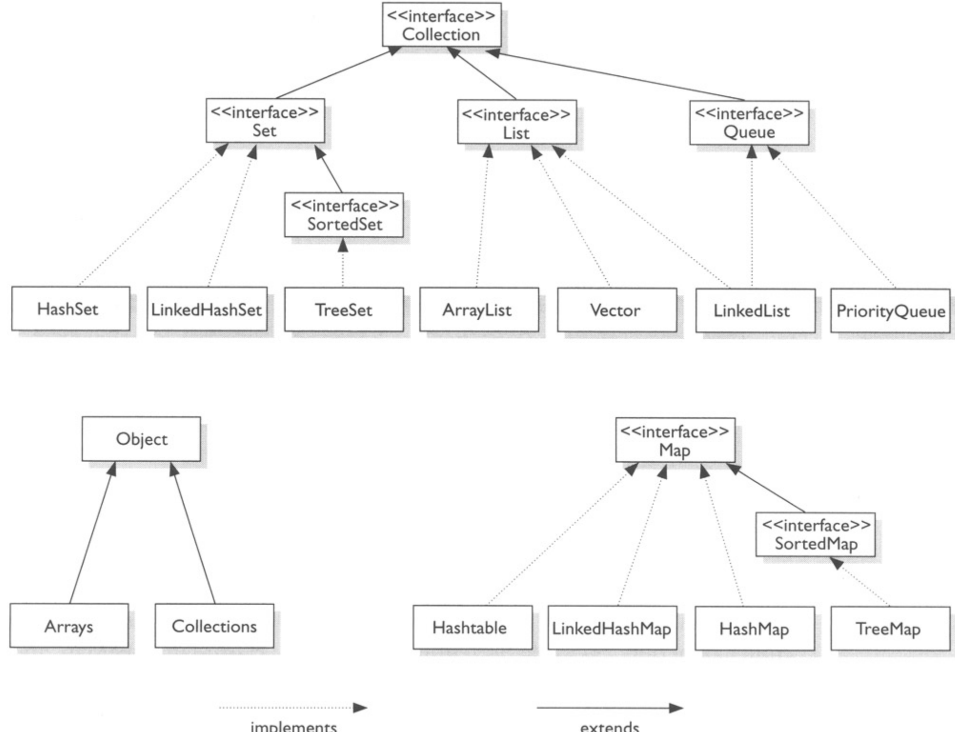


Figure 2.4: Java Collection Framework.

available implementation under the corresponding interface; *e.g.,* for the `List` data structure, the developer can use either an `ArrayList`, a `LinkedList` or a `Vector`. Some of the implementations are thread-safe (*e.g.,* `Vector`) and some are not (*e.g.,* `ArrayList`). Some of them store data sorted (*e.g.,* `TreeMap`) and some do not contain duplicate elements (*e.g.,* `TreeMap`).

2.3.3 Tradeoffs in Collection Implementations

Choosing a suitable collection implementation is a process that needs careful analysis and understanding of the internal details.

Time Usually developers choose a collection based on the asymptotic time complexity of the operations on them. In most cases, this approach works for large datasets, but it is not a good measure when the collection contains a small number of items. For small sizes, the constant values matter [118]. Also, using only the asymptotic cost is not enough because the performance of a collection depends on other factors such as the data locality of items (as shown in Figure 2.5), the cost and the approach used to increase or decrease its size, the cost of computing a hash

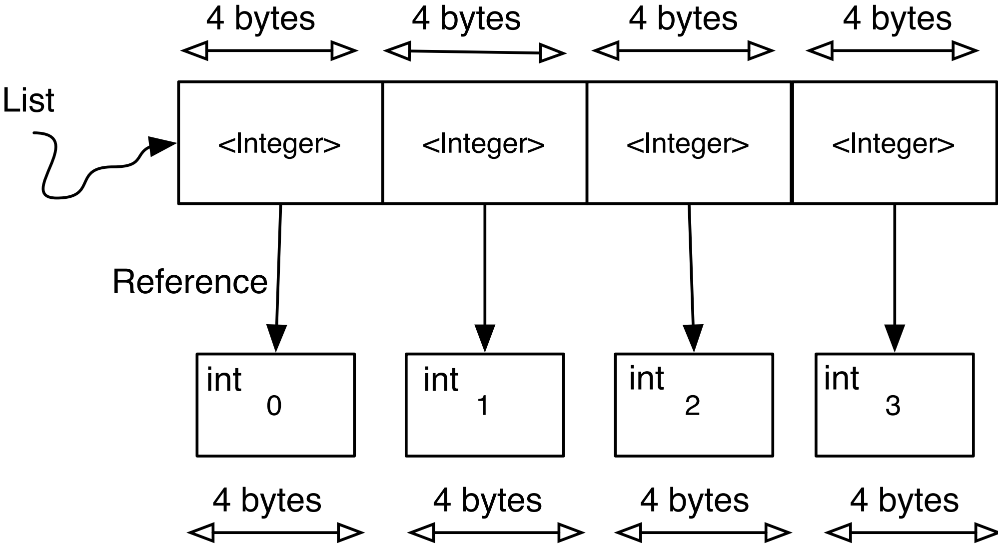


Figure 2.5: Memory layout of an `ArrayList` with integers for a 32-bit JVM.

function *etc.*

Memory Consumption Collections differ significantly between them also on the space that they consume and the data they store. In Java, all data are stored in collections as references to objects. Thus, someone has to calculate not only the size of the data itself but also the additional space consumed by the reference (*e.g.,* extra 4 bytes for each entry in an `ArrayList`) to estimate the memory consumption of the implementation that wants to choose. In Figure 2.5 we see the memory layout of an `ArrayList` implemented in Java. Except for the apparent increase in memory because of using references instead of accessing data directly, there is an additional time overhead as well. This overhead comes from the need of converting primitive types into objects and the reverse and the extract step for accessing data through references and not directly. In our example, if the developer chooses a `LinkedList` instead, each object stores a reference to the actual element, and two references to the next and previous entries in the list adding extra memory overhead ($2*4$ bytes for each entry stored).

Time/Memory Tradeoff It is significant to mention that when choosing an implementation of a particular collection, there is usually a tradeoff between execution time and memory consumption. For example, we can initially define an `ArrayList`

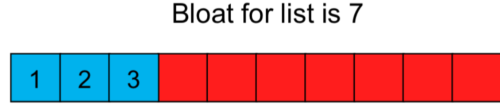


Figure 2.6: When an `ArrayList` is used with default parameters, it allocates memory for 10 entries.

with large size, such that we do not need to resize it when we insert new data. This way we can improve the execution time, but we will waste potentially much memory; *e.g.,* in Figure 2.6 space for 10 entries is pre-allocated, but we insert only three items in the `ArrayList`. On the other hand, we can pre-allocate less memory, but that will impact the execution time negatively because of the need to dynamically resize each time the `ArrayList` is full. Similarly, selecting a `LinkedList` over `ArrayList` would save memory space but would make update operations slower.

2.3.4 Empirical Rigorous Performance Evaluation

Our optimisation framework, ARTEMIS, supports multi-objective optimisation and, more specifically, aims to optimise execution time, memory consumption and CPU usage. Getting correct measurements for those objectives affects the optimisation process significantly. Usually, one of the main concerns when benchmarking Java applications is on how to do the benchmark correctly, because there exist non-determinism at run-time and many factors that affect performance, as we mentioned previously in Section 2.3.1.

A lot of research work [120, 121, 122, 123, 124, 125] has pointed out the difficulty of quantifying correctly managed runtimes. More specifically, recent work on Java performance benchmarking [126, 127] pointed out the significance of conducting a careful, well-chosen and well-motivated experimental design. The authors also mentioned that results presented in every Java performance study are subject to the available benchmarks, the implementation of the Virtual Machine, the hardware and their benchmark inputs. Thus, when reporting performance results, we need to appropriately describe the exact details of how we obtained them and follow suggested research methodologies. Otherwise, the results that we report may present a skewed view. Mytkowicz *et al.* [120] presented such surprising skewed

results by showing that systems researchers can easily make the wrong conclusions from an experiment and introduce measurement bias because of seemingly harmless aspect in the experimental setup. They further pointed out the significance of such measurement bias in leading to a performance analysis that may present wrong conclusions and how the existing research papers (survey of 133 recent papers) ignore and do not report measurement bias.

There exist several performance benchmarking methodologies used in the many research papers that try to provide robust benchmark methodologies to address the issues mentioned previously [3, 126]. Some of them suggest to run the same experiments for many executions and report mean and median confidence intervals and effect sizes. Other methodologies report the best performance and some other report the worst. In some experiments the same experiment is run multiple times within a single VM invocation in some others they include for every experiment a VM invocation as well.

2.3.5 Data Structure Optimisation and Bloat

A body of work [128, 117, 129, 130, 131, 132, 103] has attempted to identify bloat [1] arising from data structures. In 2009, Shacham et al. [118, 133] introduced a semantic profiler that provides online collection-usage semantics for Java programs. They instrumented Java Virtual Machine (JVM) to gather the usage statistics of collection data structures. Using heuristics, they suggest a potentially better choice for a data structure for a program. Though developers can add heuristics, if they lack sufficient knowledge about the data structures, they may bias the heuristics and jeopardise the effectiveness of the approach. Also, the proposed approach is specific to the JVM the authors use, meaning that this approach is not transferable to other non-JVM programming languages.

The proposal of this thesis, ARTEMIS directly uses the performance of a data structure profiled against a set of performance tests to determine the optimal choices of data structures. Those performance tests are selected from a well accepted

[1] A program is bloated when execution time and memory consumption is high compared to what the program actually accomplishes.

performance benchmark (Dacapo) and from a set of randomly selected Github projects that follow ARTEMIS's constraints. Therefore, ARTEMIS does not depend on expert human knowledge about the internal implementation and performance differences of data structures to formulate heuristics. Instead. ARTEMIS relies on carefully-chosen performance tests to minimse bias. Furthermore, ARTEMIS directly modifies the program instead of providing hints, thus users can use the fine-tuned program ARTEMIS generates without any additional manual adjustment.

Other frameworks provide users with manually or automatically generated selection heuristics to improve the data structure selection process. JitDS [134] exploits declarative specifications embedded by experts in data structures to adapt them. CollectionSwitch [135] uses data and user-defined performance rules to select other data structure variants. Brainy [136] provides users with machine learning cost models that guide the selection of data structures. ARTEMIS does not require expert annotations, user-defined rules or any machine learning knowledge. Storage strategies [137] changes VMs to optimize their performance on collections that contain a single primitive type; ARTEMIS rewrites source code and handles user-defined types and does not make VM modifications.

In 2014, Manotas et al. [31] introduced a collection data structure replacement and optimisation framework named *SEEDS*. Their framework replaces the collection data structures in Java applications with other data structures exhaustively and automatically select the most energy efficient one to improve the overall energy performance of the application. Conceptually, ARTEMIS extends this approach to optimise both the data structures and their initialization parameters. ARTEMIS also extends the optimisation objectives from single objective to triple objectives and used Pareto non-dominated solutions to show the trade-offs between these objectives. Due to a much larger search space in our problem, the exhaustive exploration search that used by *SEEDS* is not practical, therefore we adopted meta-heuristic search.

Furthermore, ARTEMIS directly transforms the source code of the programs whilst *SEEDS* transforms the bytecode, so ARTEMIS provides developers more intuitive information about what was changed and teaches them to use more efficient

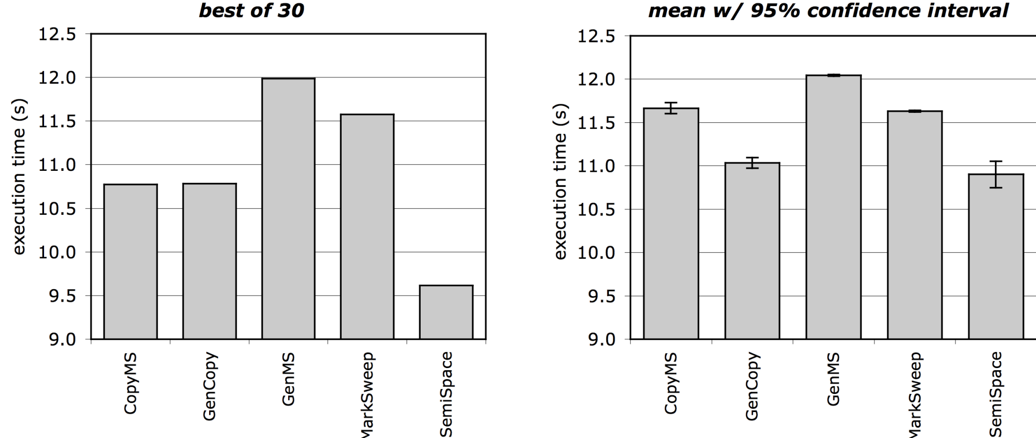


Figure 2.7: An instance illustrating the problem of reporting misleading metrics: the 'best' method is shown on the left and the empirical rigorous method is shown on the right (Figure borrowed from [3]).

data structures. Moreover, ARTEMIS can be more easily applied to other languages as it does not depend on language specific static analysers and refactoring tools such as WALA [138] and Eclipse IDE's refactoring tools. In order to support another language we just need the grammar of that language and to implement a visitor that extracts a program's Darwinian data structures. We note that ANTLR, which ARTEMIS uses, currently provides many available grammar languages [2].

Apart from the novelties mentioned above, we conduct the largest empirical study to our knowledge compared to similar work. In the studies mentioned above, only 4 to 7 subjects were included in the experiments. Our study included the DaCapo benchmark, 30 sampled Github subjects and 8 well-written popular subjects to show the effectiveness of ARTEMIS, therefore our results are empirically more meaningful. We also applied ARTEMIS in C++.

To mitigate instability and incorrect results we followed state of the art evaluation methods [3, 139, 140, 141]. Initially, as those methodologies suggest, before we run the experiments, we differentiate VM start-up and steady-state. We run each experiment for 30 runs, and we compute effect sizes and mean and median interval confidences; reporting best or worst values may be misleading as the example in Figure 2.7 shows. We followed all the suggested techniques and methodologies to

[2] https://github.com/antlr/grammars-v4/

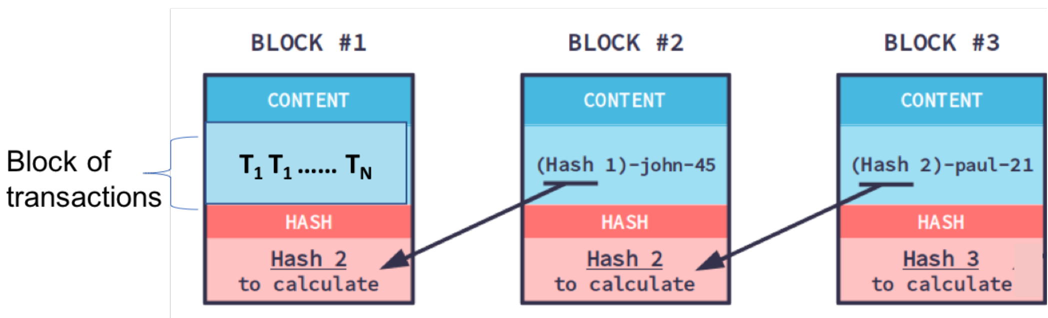


Figure 2.8: Blockchain is a chain of blocks of transactions linked by hash pointers.

guarantee that our results are robust and correct and our methodology is statistically rigorous.

2.4 Blockchain

In this section, we focus on the related research findings regarding the Blockchain technology. We give an introduction to the most popular Blockchain platform (Ethereum [142]) that allows developers to build decentralised applications. We describe Ethereum Virtual Machine (EVM) and its execution model. We then introduce Solidity, a high-level Javascript-like language to write applications (smart contracts) that run on top of EVM. We discuss the current characteristics of Solidity that limit the applicability of automatic search-based optimisation techniques. We then present the notion of Gas which is a unit used for paying miners for executing code in Ethereum. Finally, we discuss how search-based techniques can be used to minimise the amount of gas a smart-contract consumes. Then, we point out how significant impact such optimisation can have in an open, decentralised network, with thousands of nodes, which consumes a vast amount of electricity.

In recent years, Blockchain technology has become very popular as the decentralised technology used in Bitcoin [143] and other cryptocurrencies. Industrial sectors, led by the Financial sector, realised that they could use the Blockchain technology for diverse applications far beyond cryptocurrencies. In academia, Blockchain attracted interest because of the research problems that arise when having a single real-world system that combines so many different research areas, such as cryptography, distributed database management, programming languages and economic

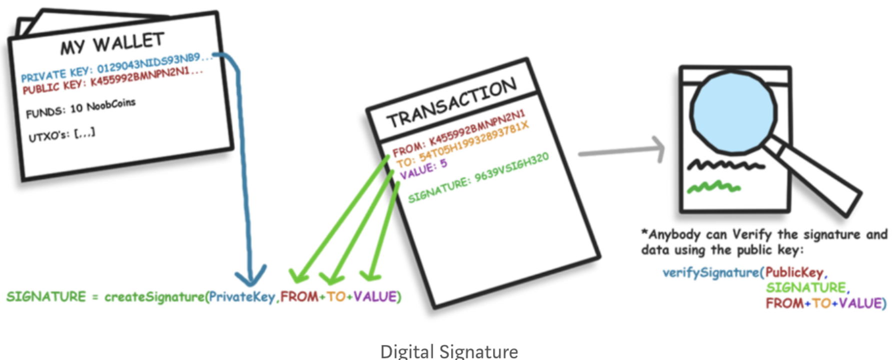
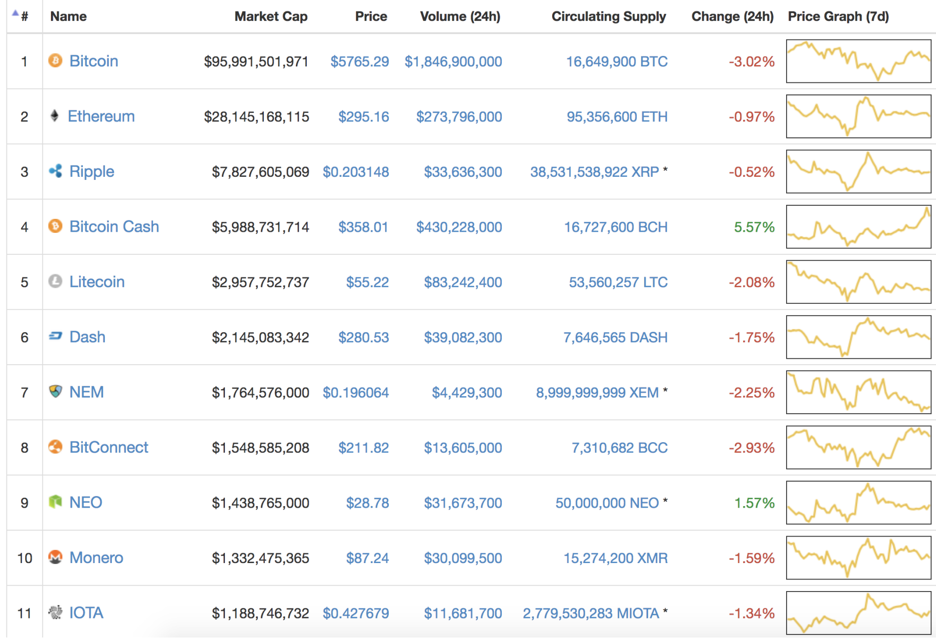


Figure 2.9: Transaction Example: In blockchain, an account with a key pair (public key and private key) is representing a wallet for submitting transactions.

incentives.

Bitcoin is the first real-world public implementation of Blockchain with significant success. Bitcoin was first proposed in 2008 [143] on the cryptography mailing list at `metzdowd.com` by an anonymous author called Satoshi Nakamoto. The described Blockchain by Nakamoto is a continuously growing public distributed ledger that stores committed transactions (see Figure 2.9) in a chain of blocks (Figure 2.8). The main characteristics of the Blockchain technology are: persistent data (once data are written in the Blockchain they cannot be removed or lost), decentralised environment (any user can join the network), audibility (chain blocks include all history of transactions) and anonymity (users participate in the network by using private and public key cryptography). To allow transactions take place in a decentralised environment with no trusted authorities, Blockchain uses several essential technologies such as asymmetric cryptography for digital signatures (a user wallet consists of a public and private key), cryptographic hashes, and distributed consensus mechanisms (Proof of Work, Proof of Stake).

Apart from cryptocurrencies, Blockchain has been used (or is under development to be used) in various services such as digital assets (store and track diamond purchases on the Blockchain), energy trading (connect energy grid with a Blockchain grid for payments), automotive industry (cars connected into a Blockchain can automatically pay toll fees) and other online payments use cases [144, 145, 146, 147]. Additionally, Blockchain is considered a promising technology for the Internet in-



#	Name	Market Cap	Price	Volume (24h)	Circulating Supply	Change (24h)	Price Graph (7d)
1	Bitcoin	$95,991,501,971	$5765.29	$1,846,900,000	16,649,900 BTC	-3.02%	
2	Ethereum	$28,145,168,115	$295.16	$273,796,000	95,356,600 ETH	-0.97%	
3	Ripple	$7,827,605,069	$0.203148	$33,636,300	38,531,538,922 XRP *	-0.52%	
4	Bitcoin Cash	$5,988,731,714	$358.01	$430,228,000	16,727,600 BCH	5.57%	
5	Litecoin	$2,957,752,737	$55.22	$83,242,400	53,560,257 LTC	-2.08%	
6	Dash	$2,145,083,342	$280.53	$39,082,300	7,646,565 DASH	-1.75%	
7	NEM	$1,764,576,000	$0.196064	$4,429,300	8,999,999,999 XEM *	-2.25%	
8	BitConnect	$1,548,585,208	$211.82	$13,605,000	7,310,682 BCC	-2.93%	
9	NEO	$1,438,765,000	$28.78	$31,673,700	50,000,000 NEO *	1.57%	
10	Monero	$1,332,475,365	$87.24	$30,099,500	15,274,200 XMR	-1.59%	
11	IOTA	$1,188,746,732	$0.427679	$11,681,700	2,779,530,283 MIOTA *	-1.34%	

Figure 2.10: Cryptocurrency Market Capitalisation. Source: `www.coinmarketcap.com`, 27/10/2017.

teraction systems of the future, such as smart contracts [148], public services [149], Internet of Things (IoT) [150, 151] and health services [152].

The most popular Blockchain product (October 2017), as we mentioned previously, is Bitcoin with tremendous success (capital market of 91-92 billion dollars in October 2017 according to coinmarketcap[3]). Then Ethereum [153] was introduced as the first successful public implementation of a Blockchain that supports programmable Smart Contracts and a capital market of 28 billion dollars in October 2017. The hype continued with other public products that leverage the Blockchain technology, such as Monero [154], ZCash [155] that provide anonymous transactions, Ripple that aims to become the cryptocurrency that financial institutions will use to do payments and IOTA [156], a product that connects Internet of Things devices using a distributed network. Figure 2.10 shows other popular public Blockchain projects. Last but not least, there exist some commercial products that aim to provide private Blockchain solutions such as IBM Hyperledger [157], R3 Corda [158] or Ethereum Alliance.

Technically, a Blockchain is a distributed peer-to-peer shared ledger (fully

[3] `https://coinmarketcap.com`

replicated between all participants or so-called miners) that allows transactions between several members in a verifiable and permanent way [143, 153], without the need of a trusted third party, *e.g.,* banks, credit card companies, *etc.* The Blockchain maintains a consistent view in a secure and fault-tolerant way in a permission-less environment accessible to everyone. Users smoothly join and leave the system without having any global knowledge of the number of participants and yet the Blockchain network operates smoothly.

Users communicate with the Blockchain through transactions which are verified and accepted by its miners. The term “transactions" implies that either that completes and is accepted as part of the Blockchain or not done at all. An accepted transaction in the Blockchain is immutable, and no other transaction can modify it. Every transaction is always cryptographically signed by the sender (creator), ensuring that only the participant holding the keys can make such a transaction. Conceptually, we can consider Blockchain as a transaction-based state machine, and each transaction updates its state.

In a database context, we can see a Blockchain as a distributed database where each node has a full copy of the data and transactions happen using cryptographic methods; instead of using some sql-like query language. Miners are continually monitoring the transactions broadcasted in the peer-to-peer network, pick a number of them and construct a block of transactions which they then propose for inclusion in the shared ledger. Each block contains a list of transactions and some metadata, such as timestamp, a Merkle hash of the transactions, the hash to the parent block. If all other miners accept a miner’s block, then it is added in the Blockchain, and usually, the miner gets a reward.

2.4.1 Consensus Layer

A consensus protocol includes the rules that determine how to add a block in the Blockchain and how to reward miners. The Bitcoin and Ethereum blockchain use the proof of work consensus protocol, in which a miner has to solve a puzzle before suggesting a block in the Blockchain. The lead miner announces the block to all other miners. Other miners verify that the proposed block follows specific predefined

rules and constraints, *e.g.,* preventing “double spending". If the block is valid, then miners include it in their copy of the Blockchain. To resolve issues where blockchain splits into multiple chains, miners follow the chain with the longest length.

Most of the existing Blockchain systems rely on the proof of work (PoW) [159] consensus mechanism. PoW works by providing each peer voting rights based on their “computing power", by solving proof of work puzzles and building the relevant hash blocks. In Bitcoin, the PoW requires the brute-force finding of hash-value that has specific characteristics. More specifically, a miner has to find a nonce value, such that when merged with the block parameters (e.g., the hash of the previous block and a Merkle hash), the value of the hash has to be smaller than the current hash value. The miner that will find first such a nonce creates the new block and transmits it to the Blockchain network; it takes approximately 10 minutes to find such a block on Bitcoin. The other peers of the network verify the specific properties of the block and include the block in their copy of the Blockchain.

Proof-of-stake (PoS) [160] was proposed to provide an alternative faster version of PoW. In PoS-based cryptocurrencies, the creator of the next block is determined based on various combinations of random selection and wealth or age (i.e., the stake). The idea is that instead of solving the Proof-of-Work, the miner that creates a block has to give a proof that it owns a specific amount of coins before being accepted by the network. Creating a block includes sending coins to oneself, which confirms the ownership. The network Blockchain network specifies the number of target coins by adjusting a difficulty mechanism, similar to PoW. The difficulty ensures that the Blockchain will add new blocks in some approximate constant time.

2.4.2 Types of Blockchain Systems

Based on who has access to the Blockchain and how it adds new blocks, there exist Permissionless, Permissioned and Hybrid Blockchains (Figure 2.11) .

2.4.2.1 Permissionless Blockchain

A permissionless Blockchain is public, decentralised, anonymous and open to everyone. The environment is untrusted, and everyone can participate in validating

Property	Public blockchain	Consortium blockchain	Private blockchain
Consensus determination	All miners	Selected set of nodes	One organization
Read permission	Public	Could be public or restricted	Could be public or restricted
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralized	No	Partial	Yes
Consensus process	Permissionless	Permissioned	Permissioned

Figure 2.11: Comparison between public, consortium and private Blockchain. Table taken from [4].

transactions and mining new blocks. Bitcoin and Ethereum are the two most famous examples of permissionless blockchain systems. The consensus mechanism used in permissionless blockchain is one that allows the network to agree on a shared state, even though untrusted participants may try to trick the network; under the assumption that the number of trusted participants is at least 51%. The most popular consensus algorithms used in permissionless Blockchains are Proof of Work and Proof of Stake. However, their scalability and transaction throughput is low when compared to traditional distributed database systems.

2.4.2.2 Permissioned Blockchain

Permissioned Blockchains attracted attention after Bitcoin and Ethereum became popular. Financial and other sectors started looking at how to use Blockchain technology to solve problems between participants that may not trust each other, how to share data securely and how to avoid third-party intermediaries. A permissioned Blockchain is one that restricts the number of participants in the consensus algorithm. In particular, the existing peers need to approve a new participant before joining those that can propose and validate transactions in the Blockchain. Usually, the number of participants in a permissioned Blockchain system is small (between 10 to 100), there is no notion of a currency and the consensus algorithm used is not proof-based. Because all the participants are known, faster consensus algorithms can

be used such as PBFT [161] or Paxos [162] allowing higher transaction throughput and better scalability. Example of permissioned Blockchains, popular to the financial industry, are IBM Hyperledger Fabric [157], Private Ethereum Consortium [163] (a version of Ethereum which runs on top of different consensus algorithm and limited to specific participants only) and Corda R3 [164].

2.4.2.3 Hybrid Blockchain

Other work has proposed Hybrid Blockchain solutions [165, 166, 167] to address scalability issues of permissionless Blockchains. In particular, Hybrid Blockchains try to combine public and private Blockchains into a single network. In a hybrid Blockchain, a specific number of supernodes (usually fast servers with significant processing power) are considered miners and allowed to validate transactions, but anyone can participate in the network. The consensus protocol used is also a version of PBFT [161] because the supernodes (miners) are few and known to each other. Note that users have to trust the specific number of supernodes, thus there is some kind of centralisation in the network. Other solutions try to address scalability issues by storing some of the data on-chain and some off-chain [168, 169, 170]

2.4.3 Ethereum Blockchain

The focus of our paper is on Ethereum Blockchain, the first public implementation of a Blockchain that supports programmable smart contracts. Ethereum is considered a more generalised Blockchain platform that combines the notion of public economic consensus via proof of work (or eventually proof of stake) with the abstraction power of a stateful Turing-complete virtual machine. Ethereum allows developers to generate applications that benefit from the decentralisation and security properties of Blockchains, bypassing the need for building a new Blockchain for each new application. Members use Ethereum for simple transactions, similarly to Bitcoin, but additionally to write decentralised applications using smart contracts.

The success of Ethereum as a public Blockchain led to a private permissioned Ethereum Blockchain version used by various (financial mostly) institutions. More specifically, cloud providers, such as Microsoft Azure, provide Blockchain as a

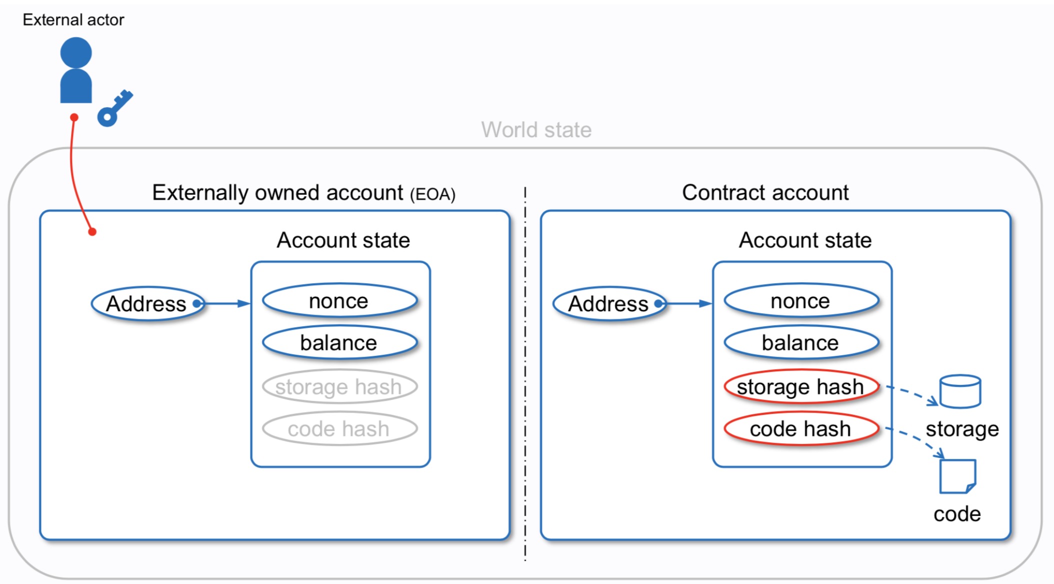


Figure 2.12: Account types on Ethereum [5]. External account is controlled by a private key and does not contain code. Contract is controlled by EVM code.

service that allows participants to create their private Blockchain network. Then, they can easily run their decentralised applications on top of it. Institutions are more interested in the immutability aspect of the Blockchain as they can use it as a proof to regulators that particular transactions happened; without someone having the ability to remove those transactions. Also, institutions aim to combine legal contracts with programmable smart contracts, with the aim to minimise the cost by removing third-party legal intermediaries [144].

Another reason to use private Ethereum Blockchain is the fact that participants can share data easily because the Ethereum protocol will keep the data consistent automatically on the background. Last, for private Blockchains, the participants would like to be more flexible, when they develop decentralised applications, and to use more traditional software deployment and versioning mechanisms, which currently do not exist. Thus, the participants are more willing to agree on upgrading existing smart contracts with new features and not rely on immutable code that they cannot change in case of bugs.

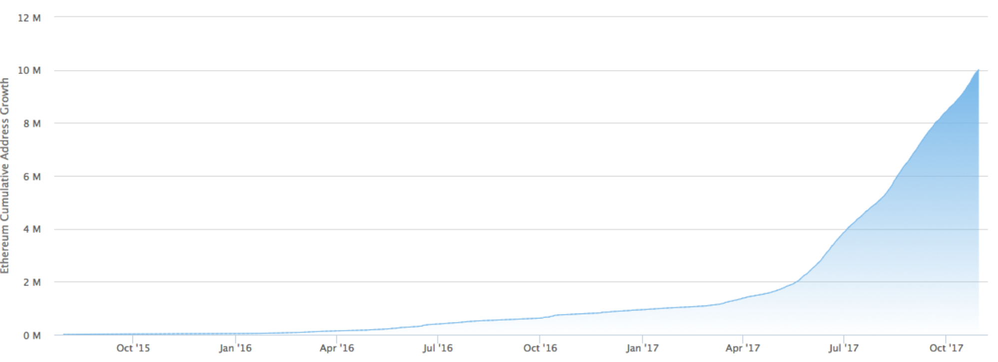


Figure 2.13: Number of Ethereum addresses in 31 October 2017. (Source: `https://etherscan.io/chart/address`)

2.4.3.1 Ethereum Account Types

Contrary to Bitcoin that has only user accounts that hold coins, Ethereum provides two types of accounts. Both those accounts share the same address space. Ethereum has traditional external accounts (user wallets that hold the Ether coin) that use asymmetric public and private key pairs. It also has contract accounts (smart contracts) that are managed by code saved in them and run on top of Ethereum Virtual Machine (EVM) (Figure 2.12). Users can send Ether coins between them using standard transactions, but also they can add more complicated logic in programmable contracts, which can also commit transactions.

The address of an external account is its public key, and the address of a contract account is automatically assigned when created; the address of the contract is generated from the address of the contract creator and the number of transactions sent from that address (“nonce"). EVM handles equally both types of accounts with the only distinction that the external accounts do not have storage and code associated with them. The way a user sends Ether from his wallet to another user’s wallet is similar to sending Ether to a smart contract. The user enters the receptient’s address that he/she wants to send the Ether, without necessarily knowing if that is an external owner or a smart contract account.

The smart contracts provided in Ethereum are quite attractive to users as they can run distributed applications (Dapps). Dapps can include storage, payments, and cryptographic services all in the context of a contract script. Users have used smart

contracts to build a wide range of distributed application including applications that can exchange automatically financial instruments (money, content, currencies, financial derivatives, savings wallets, wills, etc), autonomous governance applications [171] or even gambling applications [172]. Figure 2.13 shows the number of available accounts (both user account addresses and smart contract addresses) deployed on Ethereum between January 2016 and October 2017. We can see that there are more than 10 million accounts in Ethereum and from those addresses, more than 1 million are smart contract accounts.

A deployed Ethereum smart contract is public, and anyone can interact with it through transactions. Once a smart contract is deployed, its code is immutable. That means that we can access the functions the contract has, but we can not add more functions to it. In particular, because Blockchain is running in a trustless environment, the idea is that participants should trust only the code deployed and not have trust on some other entity that has the right to update the contract. Thus, developers should test well the functionalities of a smart contract, before they deploy it on the Blockchain; they will not be able to fix bugs if the contract has any.

The code of an Ethereum contract is written in a low-level, stack-based bytecode language referred to as Ethereum virtual machine (EVM) code. Usually, developers define contracts using higher-level programming languages (*e.g.,* Solidity, a Javascript-like language), which are compiled to EVM code. To invoke a contract at a specific address, users send a transaction to the contact's address. A transaction usually includes payment (to the contract) for the execution (in Ether) and input data for invocation.

2.4.3.2 A Smart Contract Example.

Listing 2.1 is a straightforward simplified contract, written in Solidity, which can be used to issue new tokens/currencies inside the Ethereum ecosystem; a highly used contract by different organisations to raise funding in Ether. When the creator of the MyToken contract publishes it on the Blockchain, the EVM executes the code inside the constructor first. The constructor assigns an initial amount of tokens to the contract creator. The contract uses a mapping called `balances` that keeps track

```
1  contract MyToken {
2   mapping (address => uint256) public balances;
3   constructor (uint256 supply) {
4    balances[msg.sender] = supply;
5   }
6   function transfer(address _to, uint256 _value) {
7    require(balances[msg.sender] >= _value);
8    require(balances[_to] + _value >= balances[_to]);
9    balances[msg.sender] -= _value;
10   balances[_to] += _value;
11  }
12 }
```

Listing 2.1: "A simple contract in Solidity that can be used to issue a new token."

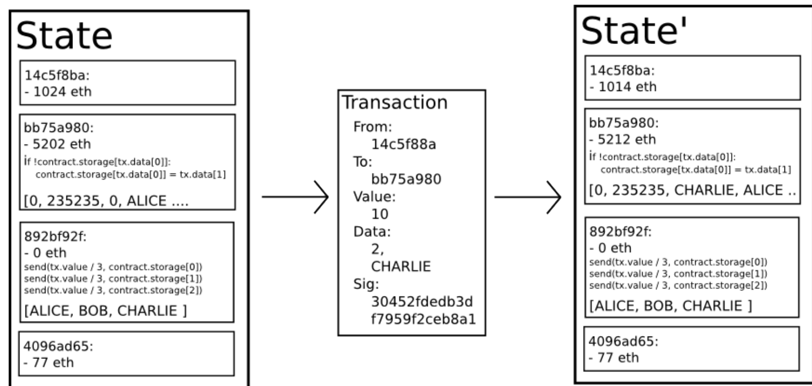


Figure 2.14: When a transaction is accepted on the Blockchain, each mining node change their Blockchain state.

users and their number of tokens. A user can transfer the token to any other user or contract by sending a transaction that calls the `transfer` function in Line 6. The transaction should contain the information of the sender, the value (amount of Ether sent to the contract), the gas to be spent and the included data of the invocation transaction function; all those data are stored in variable name called `msg` which is part of the Ethereum instruction set.

2.4.3.3 Ethereum Virtual Machine

The EVM runtime environment is sandboxed and runs in isolation, meaning that the code inside EVM has no access to network, filesystem or other processes. EVM

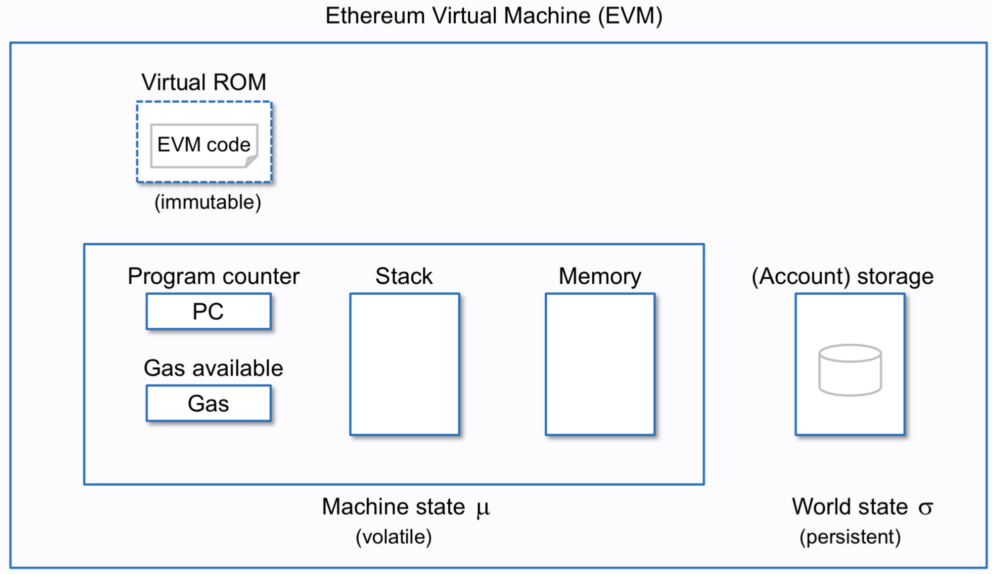


Figure 2.15: The EVM is a simple stack-based architecture [5].

maps addresses (160-bit words) to account states. An account state contains code, persistent private storage, nonce and the balance (see Figure 2.15). EVM also has access to several global parameters (*e.g.,* the current number of Ether, the gas price, the current block difficulty or current block gas limit), but the accounts store the most states. The Ethereum Virtual Machine has three areas where it can store items a) storage, b) memory and c) stack. During execution, EVM maintains an infinitely expandable byte-array termed "memory", the "program counter" pointing to the current instruction, and a stack of 32-byte values. At the start of execution, memory and stack are empty, and the program counter is zero.

Ether Ether is the cryptocurrency that comes with the Ethereum Blockchain platforms. When Ethereum first went public the total number of Ether was 90 million. New Ether are generated continously with a predefine rate, which are used as payment to the miners that find and add new blocks in the Blockchain. Ether is considered a necessary element, a fuel, for running the Ethereum on a fully open decentralised setting. It is a method of reimbursement made by the consumers of the platform to the machines executing the requested transactions. Note that Ether can be ignored, if desired, on private Ethereum networks.

Gas Ethereum supports the code execution in an open distributed system; each

miner of the Blockchain executes the code. To pay miners for the resources they use to execute the code, Ethereum introduced a measure which is called “gas". The Blockchain network pays the miners in gas based on the amount of work they did. Gas is also essential for putting an upper limit on how much work a transaction can execute; this is necessary for an open environment where some participants may deliberately consume many resources. When a user creates a transaction, he/she associates a default amount of gas with it. While the EVM is executing a transaction, the gas is gradually consumed based on what operations it executes. The creator of the transaction sets the gas price (in Ether) upfront and the maximum amount that wants to pay. Usually, miners will prioritise those transactions that pay the highest price of gas. If the transaction executes successfully, the remaining gas returns to the sending account. If the transaction uses all the gas but has not yet finished, then the EVM throws an out of gas exception error and reverts all the modifications made by the transaction, without refunding the gas.

Storage Each account has a storage area where all state variables reside. EVM implements storage as a key-value store of 256-bit words and is persistent among function calls. Operations for reading and modifying storage are quite expensive to use; obtaining a new storage cell costs $20,000$ gas, transforming an occupied cell costs $5,000$ gas, reading a cell costs 200 gas. It is not feasible to enumerate storage, and a contract cannot read/write other contract's storage apart from its own. We can resemble the storage with a hard drive; once the code runs, the EVM records everything, and in the next contract call, we will have access to all the earlier acquired data.

Memory Each account can also use the memory, of which a contract receives a freshly cleared instance for each message call. Memory is used to hold temporary values; *e.g.,* function arguments, local variables and storing return values. We can compare memory with RAM; when the computer (in our case the EVM) is switched off, it erases all the data that reside in memory. Structurally, memory is a linear byte array. Initially, its size is zero, and it expands by words of 256-bits. It costs only three gas items to read and record one digital word. As for the memory extension,

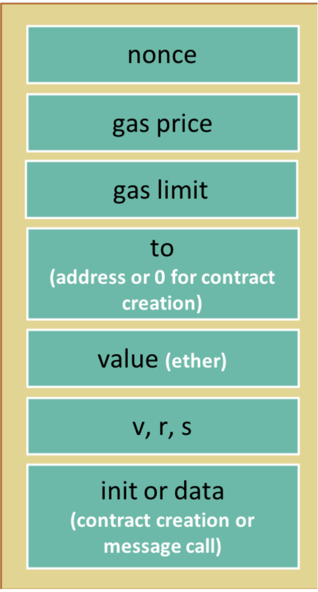


Figure 2.16: Internal fields of a transaction.

it becomes more expensive depending on the current size. The storing of several Kbytes will be cheap enough, but 1Mbyte will already cost millions of gas because the price grows quadratically.

Stack The EVM is a stack-based machine with a maximum size of 1024 elements of 256 bits each. It is used to conduct all the EVM calculations, and costs as much to use as the memory. Access to the stack is limited to the top end and data are moved from stack to storage or memory. It is not feasible to access arbitrary elements deeper in the stack without first removing the top of the stack. If a Stack overflow happens, the contract implementation stops and EVM throws an exception.

2.4.3.4 Contract State Transitions / Transactions

An external account can launch two types of transactions: a sending transaction and a contract-creating transaction. A sending transaction is a standard transaction, containing a receiving address, an ether amount, a data byte-array and some additional parameters, and a signature from the private key associated with the sender account (Figure 2.16). A contract creating transaction looks like a standard transaction, except the receiving address is blank. When the Blockchain accepts a contract-creating transaction, the data byte-array in the transaction represents the EVM code, and the

value returned by that EVM execution is the code of the new contract; by default, the EVM executes the code defined in the constructor of the contract. The address of the new contract is deterministically computed based on the sending address and the number of times that the sending account has made a transaction before (called the account nonce). When the called account is an external account, a simple balance transfer occurs. Otherwise, when the called account is a contract, after the balance transfer, the called contract's code is executed.

A transaction belongs to a block. A block is a unit of agreement among Ethereum nodes. EVM has special instructions that read the block number and the cryptographic hash values of some previous blocks. Since a block defines a prior block but not a unique successor, blocks in the network sometimes may form a tree, but, as far as the states of EVM are concerned, only one branch in the tree matters. Because of this, we can think of EVM as a sequentially executed machine.

2.4.3.5 Message calls

A contract can call other contracts, send Ether to regular accounts or even create other new contracts, by using message calls. Message calls are quite similar to transactions as they both have a source address, a target address, data payload (`calldata`), Ether, gas and return data. Each contract has access to the `calldata` payload. The payload of `calldata` is an immutable, non-persistent area where the function identifier and its arguments are stored, and has a similar structure to memory. After a message call has finished execution, it can return data and store them at a location in the caller's memory preallocated by the caller. Calls are limited to a depth of 1024, meaning that for more complex operations, developers should favour loops over recursive calls.

2.4.3.6 Function Dispatch

To execute a particular function of a contract, the user must specify the name of the function in the transaction's `calldata` (`msg.data`). The EVM uses an identifier from the `calldata` (function definition and its parameters) to load the corresponding code. In particular, EVM contains an immutable internal mapping (vtable) with key a unique function idenfier (generated by the 4 first bytes of the result of the keccak256 hash function, `bytes4(keccak256("f(param)"))`) and value a reference

```
address owner;

function initWallet(address _owner) {
  owner = _owner;
}

function withdraw(uint amount) {
  if (msg.sender == owner) {
    owner.send(amount);
  }
}

function () payable {
  // receive money
}
```

Solidity

compile

```
id = bytes4(msg.data);

if (id == 0x21faf2ee) {
  // EVM code of initWallet
}

else if (id == 0xe3710ed0) {
  // EVM code of withdraw
}

else {
 // EVM code of fallback function
}
```

EVM-like pseudo code

Figure 2.17: Function dispatch example in smart contracts.

that points to the correct section of code for execution. If the function identifier matches one of the function identifiers of the contract, the EVM executes the code of the function. Figure 2.17 shows an example of how EVM dispatches a function written in Solidity and the corresponding EVM pseudocode that the compiler generates. The EVM loads the corresponding bytecode for the withdraw function, by using its identifier's value $0xe3710ed0$.

If none of the contract's functions matches the given function identifier, the EVM executes an unnamed function, called the fallback function (Figure 2.17). The fallback function cannot have arguments (except the ones in **`msg.data`**), cannot return data and is always externally visible. If a contract receives Ether (without any data in the message call), the EVM executes the fallback function and, if it contains the `payable` keyword, then it allows the contract to receive Ether. The fallback function has a limit of 2300 gas when the user chooses the **`send`** command for sending Ether, without allowing other more complex operations except basic logging; the send command has no way to specify the gas amount. Similarly to other functions, the fallback function can execute complex operations, as long as there is enough gas passed on to it.

2.4.3.7 Types of Message Calls

Solidity supports two other very important variations of message **`call`**, the **`delegatecall`** and **`callcode`** commands. Those two variations allow the code

```
contract Caller {
 function foo(Relay relay, Target target, uint input) {
  relay.f_delegate_call(target, input);
 }
}

contract Relay {
 uint public dummy;
 address public sender_address;

 function f_call(address target, uint input) {
  // Use Target's storage, Target is modified
  target.call(sig(), input);
 }
 function f_callcode(address target, uint input) {
  // Use Relay's storage, Target is not modified
  target.callcode(sig(), input);
 }
 function f_delegate_call(address t, uint input) {
  // Use Relay's storage, Target is not modified
  target.delegatecall(sig(), input);
 }
 function sig() returns (bytes4){
  return bytes4(keccak256("set_dummy(uint256)"));
 }
}

contract Target {
 uint public dummy;
 address public sender_address;

 function set_dummy(uint input) {
  dummy = input;
  // sender_address is Relay's address for call and
     callcode
  // sender_address is Caller's address for delegatecall
  sender_address = msg.sender;
 }
}
```

Listing 2.2: Code that demonstrates the differences between callcode, delegatecall, and call commands.

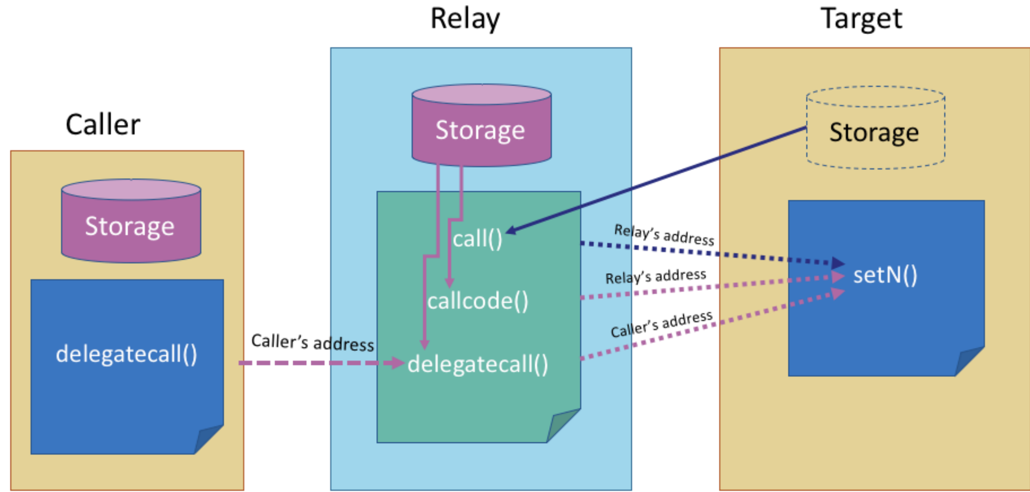


Figure 2.18: Visualisation of function calls between contracts.

of a target address to be executed in the context of the calling contract. The **delegatecall** and **callcode** instructions were introduced as low level instructions to implement libraries in Solidity. The main feature of a library is that it does not own any storage, but it can have access to the storage of the contract that is using the library. Thus, a library provides functions which can be executed by multiple different contracts, avoiding the need to have duplicate code in the Blockchain.

In this thesis, we use **delegatecall** and **callcode** to provide upgreadale contracts to the user. In particular, we utilise the properties of those commands to execute dynamically code from another contract (we name them child contracts) in the context of an existing contract (we name this contract as parent contract). To undestand the differences between those message calls, we use the example code in Listing 2.2 (modified version from stackoverflow [173]) and its abstract visual representation in Figure 2.18.

In Listing 2.2, we define three contracts: `Caller`, `Relay` and `Target`. `Target` is the main contract that contains the final code to be executed and in which all calls will eventually conclude. `Relay` is an intermediate contract that forwards calls to the `Target` contract, and it contains three simple set functions. `Caller` contract acts as the user that will call the `f_delegate_call` function of `Relay` contract.

When `Relay` does **call** on `Target`, the code runs in the context of `Target`, but

`Relay` uses the storage of `Target`. When `Relay` does **`callcode`** or **`delegatecall`** on `Target`, the code runs in the context of `Relay`; the code of `Target` is in `Relay`. Whenever the code writes to storage, it writes to the the storage of the `Relay` contract, instead of the `Target` contract. EVM matches the storage beteen contracts by using an offsetting mechanism. More specifically, the variables of the child contract are references to the variables of the parent contract; the first integer in the parent contract correlates with the first integer in the child contract.

When a contract is called, there is a special variable **`msg.sender`** that stores the address of the caller and a **`msg.value`** variable that refers to the amount of Ether sent to the contract. Those two variables are vital to Ethereum because they are used to associate users with their wallet accounts as well as to provide Ether transfer and payment between them. When **`delegatecall`** is used to forward the call to a new contract, it also propagates the **`msg.sender`** and the **`msg.value`** unchanged to the new contract; **`callcode`** does not forward those two variables and this is the main difference with **`delegatecall`**. In particular, when `Relay` does **`callcode`** on `Target`, **`msg.sender`** inside `Target` is `Relay` (Figure 2.18). When an account `Caller` invokes `Relay`, and `Relay` does **`delegatecall`** on `Target`, **`msg.sender`** inside `Target` is `Caller`. That means that `Target` has the same **`msg.sender`** and **`msg.value`** as `Relay`.

2.4.4 Security issues with Solidity smart contracts

Solidity was designed as a new high-level programming language that aimed to be a Turing complete language and allow developers to write smart contracts easily. However, many vulnerabilities and attacks on the Ethereum Blockchain happened because of design flaws of the language (*e.g.,* `transaction.origin` command) or because developers did not use it properly.

The most popular hack incident that happened in Ethereum was the DAO incident [174] where the attacker exploited the reentrancy vulnerability to steal over 3.600.000 Ether (30M USD the time the hack happened). More specifically, DAO was a smart contracted designed to represent a fully autonomous, decentralised organisation where rules and the structure of traditional organisations are specified

in the code, without the need for a central authority. Another popular hack happened to MultiSig contract [175], in which the attacker managed to steal 150,000 Ether (30M USD) because of a wrong usage of the `delegatecall` command.

Most of such hacks happened in Ethereum because of the following security issues:

- Call to unknown. Solidity supports call forwards using `call`, `delegatecall`, `send` functions and there is a risk that those calls may execute unknown malicious code. We need tools, like PROTEUS, that will help developers use those functions properly by automatically re-writing their code, using proven re-writing rules.

- Re-entrancy. A contract may allow a function to re-enter a caller function before it terminates; a function is executed multiple times, while the scope was to allow only one execution. Such vulnerability was used to the DAO attack and later led to Ethereum fork [174].

- Private, Public confusion. Some design decisions that developers of the Solidity language have taken, which are different from most commonly used programming language have, introduced errors. For example, if a function does not include the keyword `private` or `public` in its definition, is by default `public`; in Java and other popular languages the default value is `private`. This confusion led to a function in the Parity Multisig Wallet to be public instead of private and as a result to be called by a hacker.

- Immutability. When developers identify a buggy contract, they cannot update it, and thus they have usually to kill it or prevent other users from calling it, trying at the same time to move its money or metadata in other contracts.

- Lost Ether. If a transaction refers to an unknown (orphan) address and sends money (Ether) to it, then that money is lost and cannot be retrieved [175].

- Transaction Ordering Dependency. Errors may appear when the assumed state of the Blockchain is different from the real one. During the mining process,

miners give an order to transactions, and this order may affect the correct execution. Sergey *et al.* [176] analysed in depth this problem by examining the similarities between multi-transactional behaviours of smart contracts and classical shared-memory concurrency issues.

2.4.5 Research Findings on Ethereum Smart Contracts

Most of the research on Ethereum focuses on identifying security issues and bugs in smart contracts. Delmolino *et al.* [177] presented a study in which they taught students how to program smart contracts, and they exposed numerous common traps in designing secure smart contracts. They noticed problems such as contracts that do not refund their users, sensitive user data exposed without encryption, and lack of incentives for the developers to take specific actions. They further proposed approaches to mitigate these faults and suggest best practices for smart contracts programming. However, they did not develop a systematic approach for detecting such smart contracts problems.

Atzei *et al.* [178] described many attacks related to Solidity, the EVM and the Ethereum Blockchain, execution model. They also defined 12 vulnerabilities which contract creators should take into consideration when writing contracts. Cook *et al.* [179] also focused on Ethereum Smart contract security issues and recommended a live monitoring and protection system for smart contracts.

Luu *et al.* [180] recommended methods to improve the operational semantics of Ethereum and decrease the number of vulnerabilities in contracts. They introduced Oyente, a symbolic execution-based tool that finds potential security issues in smart contracts. OYENTE takes as input the bytecode of a contract and the Ethereum global state and notifies the developer about security problems. Oyente targets issues such as mishandled exceptions, transaction-ordering dependence, timestamp-dependencies and reentrancy. Among $19,336$ Ethereum contracts, Oyente succeeded to flag $8,833$ of them as vulnerable.

Kalra *et al.* [181] also focused on smart contract bugs. They presented ZEUS, a formal verification framework for smart contracts that allows users to build and verify correctness and fairness policies over them. ZEUS uses both abstract interpretation

and symbolic model checking to verify contracts for safety. In their evaluation, the authors state that about 94.6% of contracts (that hold cryptocurrency worth more than $0.5 billion) are vulnerable.

Tikhomirov *et al.* [182] provided a comprehensive classification of code issues in Solidity and developed SmartCheck, a static analysis tool to discover security vulnerabilities. They use ANTLR to parse Solidity source code and a custom grammar to generate an XML-based intermediate representation; they use XPath queries to detect vulnerabilities. SmartCheck cannot detect bugs that require more sophisticated techniques, but it has the benefit of notifying developers for fixing simple bugs fast.

Nikolic *et al.* [183] proposed a new class of trace vulnerabilities, after investigating multiple invocations of a contract over its lifetime. They focused on finding contracts that lock user's fund permanently, leak them accidentally to arbitrary users, or anyone can kill them. They developed MAIAN, a tool that precisely specifies and reasons about trace properties. In their evaluation of approximately one million contracts, MAIAN flagged 34,200 (2,365 distinct) contracts vulnerable, in 10 seconds per contract.

Other researchers have focused their attention on finding solutions for other issues that smart contracts introduce such as data privacy, gas consumption, blockchain scalability and secure communication between Blockchain and other external resources. Chen *et al.* [10] developed GASPER, an automatic analysis tool that detects gas-costly programming patterns. GASPER takes as input the bytecode of the contract and uses symbolic executions to generate a control flow graph to detect such patterns. Zhang *et al.* [184] presented Town Crier, an authenticated data feed system that acts as a bridge between smart contracts and existing external websites, which are already generally trusted for non-blockchain applications. It combines a blockchain front end with a trusted hardware back end to scrape HTTPS-enabled websites and serve source-authenticated data to relying on smart contracts. Kosba *et al.* [148] presented Hawk, a decentralised smart contract system that preserves transactional privacy from the public's view by not storing financial transactions in

the clear on the Blockchain. Christidis *et al.* [185] focused on the issues that have to be solved when connecting Blockchain networks with the Internet of Things (IoT) devices and they identified some solutions and workarounds.

Even though some of those research findings point out the limitations and the security issues that arise because of the immutability aspect of smart contracts in Ethereum, they do not tackle this problem. None of the existing research work, to the best of our knowledge, focus on providing upgradeable smart contracts or providing a solution to achieve that automatically.

Chapter 3

Darwinian Data Structure Selection

In this chapter, we propose ARTEMIS, a framework that automatically improves the performance of a program by selecting and tuning its data structures. Our goal is to expose Darwinian Data Structures from the source code of an application and express their selection and tuning as an optimisation problem that can be solved using Genetic Improvement. We show that our proposed approach generalises to different programming languages (Java, C++), and we perform a rigorous statistical evaluation of the improvements that ARTEMIS achieves.

When selecting data structures from libraries, in particular, developers tend to rely on defaults and neglect potential optimisations that alternative implementations or tuning parameters can offer. This, despite the impact that data structure selection and tuning can have on application performance and defects. Data structures are a particularly attractive optimisation target because they have a well-defined interface; many are tunable; and different implementations of a data structure usually represent a particular trade-off between time and storage, making some operations faster but more space-consuming or slower but more space-efficient. For instance, an ordered list makes retrieving the entire dataset in sorted order fast, but inserting new elements slow, whilst a hash table allows for quick insertions and retrievals of specific items, but listing the entire set in order is slow. We introduce *Darwinian data structures*, distinct data structures that are interchangeable because they share an abstract data type and can be tuned. We call them Darwinian Data Structures (DDS), since we can subject their implementations to survival of the fittest. The Darwinian data

structure optimisation problem is the problem of finding an optimal implementation and tuning for a Darwinian data structure used in an input program.

We aim to help developers perform optimisation cheaply, focusing solving the data structure optimisation problem. We present ARTEMIS, a cloud-based optimisation framework that identifies *Darwinian data structures* and, given a test suite, *automatically* searches for optimal combinations of implementations and parameters for them. ARTEMIS is language-agnostic; we have instantiated it for Java and C++, and present optimisation results for both languages (Section 3.4). ARTEMIS' search is multi-objective, seeking to simultaneously improve a program's execution time, memory usage, and CPU usage while passing all the test suites. ARTEMIS scales to large code bases because is uses a Genetic algorithm on those regions of its search space with the most solutions (Section 3.3.4). ARTEMIS is the first technique to apply multi-objective optimisation to the Darwinian data structure selection and tuning problem.

ARTEMIS promises to change the economics of data structure optimisation. Given a set of Darwinian data structures, ARTEMIS can search for optimal solutions in the background on the cloud, freeing developers to focus on new features. ARTEMIS makes economical small optimizations, such as a few percent, that would not pay for the developer time spent realizing them. And sometimes, of course, ARTEMIS, by virtue of being used, will find unexpectedly big performance gains.

ARTEMIS is a source-to-source transformer. When ARTEMIS finds a solution, the program variants it produces differ from the original program only at constructor calls and relevant type annotations. Thus, ARTEMIS' variants are amenable, by design, to programmer inspection and do not increase technical debt [186]. To ease inspection, ARTEMIS generates a diff for each changes it makes. Developers can inspect these diffs and decide which to keep and which to reject.

We report results on 8 popular diverse GitHub projects, on DaCapo benchmark which was constructed to be representative, and a corpus of 30 GitHub projects, filtered to meet ARTEMIS's constraints and sampled uniformly at random. In this study, ARTEMIS achieves substantial performance improvements for all 43 projects

in its corpus. In terms of execution time, CPU usage, and memory consumption, ARTEMIS finds at least one solution for 37 out of 43 projects that improves *all* measures. Across all produced optimal solutions, the median improvement for execution time is 4.8%, memory consumption 10.1% and CPU usage 5.1%. This result is for various corpora, but it is highly likely to generalise to arbitrary programs because of the care we took to build a representative corpus (Section 3.4.1).

These aggregate results understate ARTEMIS's potential impact. Some of our benchmarks are libraries or utilities. All of their clients will enjoy any improvements ARTEMIS finds for them. Three examples are the Apache project's powerful XSLT processor `xalan`, `Google-http-java-client`, the unbiquitious Java library for accessing web resources, and Google's in-memory file system `Jimfs`. Section 3.4 shows that ARTEMIS improved `xalan`'s memory consumption by 23.5%, while leaving its execution time unchanged; ARTEMIS improved `Google-http-java-client`'s execution time by 46% and its CPU usage by 39.6%; finally, ARTEMIS improved `Jimfs`'s execution time by 14.2% and its CPU usage by 10.7%, while leaving its memory consumption unchanged.

Our principal contributions follow:

- We formalise the Darwinian data structure selection and optimisation problem $\mathbf{DS^2}$ (Section 3.2).
- We implement ARTEMIS, a multi-language optimisation framework that automatically discovers and optimises sub-optimal data structures and their parameters.
- We show the generalizability and effectiveness of ARTEMIS by conducting a large empirical study on a corpus comprising 8 popular GitHub project, 5 projects from the DaCapo benchmark, and 30 Github projects, filtered then sampled uniformly. For all 43 subjects, ARTEMIS find variants that outperforms the original for all three objectives.
- We provide ARTEMIS as a service, along with its code and evaluation artifacts at `http://darwinianoptimiser.com`.

```
1 <T> List<T> getAsList(T value) {
2  if (value == null)
3   return null;
4  List<T> result = new ArrayList<T>();
5  result.add(value);
6  return result;
7 }
```

Listing 3.1: A function from http-java-client.

3.1 Motivating example

Listing 3.1 contains a code snippet from `google-http-java-client`[1], a popular Java library for accessing efficiently resources on the web. In the Listing 3.1, `getAsList` packages HTTP headers and is invoked frequently from other methods because they use it every time they construct an HTTP request. Thus, its functionality is important for the performance of `google-http-java-client`.

Listing 3.1 uses `ArrayList` to instantiate the `result` variable. However, other `List` implementations share the same functionality but different non-functional properties. Thus, replacing `ArrayList` with other `List` implementations may affect the performance of the program. Considering the variant created when replacing `ArrayList` (Listing 3.1, line 4) with `LinkedList`, when we compare it with the original program against the same test set for 30 runs (Section 3.3), we see that the `google-http-java-client` achieves a median 46%, with 95% Confidence Interval [45.6%, 46.3%] improvement in execution time (Section 3.4).

ARTEMIS, our optimization framework, automatically discovers underperforming data structures and replaces them with better choices using search-based techniques (Section 3.3.4). First, it *automatically* creates a store of data structures from the language's Collection API library (Section 3.3.1). Then, ARTEMIS traverses the program's AST to identify which of those data structures are used and exposes them as parameters to the ARTEMIS's OPTIMIZER (Section 3.3.4) by transforming line 4 into

```
1 List<T> result = new D<T>();
```

[1] https://github.com/google/google-http-java-client

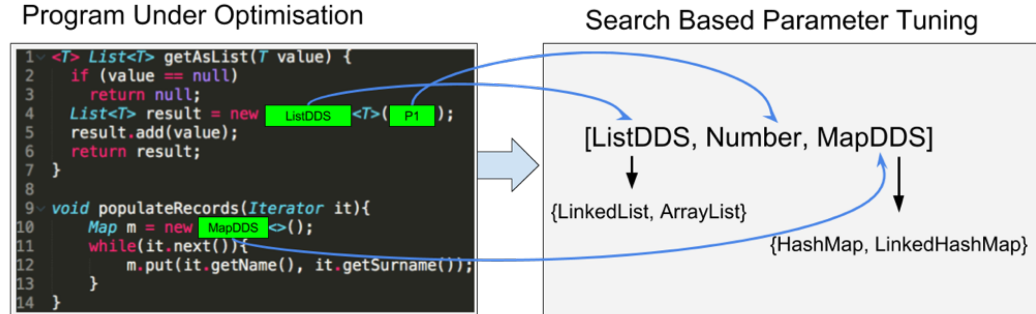


Figure 3.1: Example of how ARTEMIS maps the extracted darwinian data structures and its parameters to a search-based optimisation problem.

where *D* is the tag that refers to the exposed parameter associated with the defined data structure type (Section 3.3).

Listing 3.1 does not specify the initial capacity size of the `ArrayList`, so the default size 10 was used. If the instantiated `List` object contains less than 10 items, the default capacity can result in memory bloat. If the `List` object contains more than 10 items, the default capacity can slow the execution time; more memory realllocation operations will happen. Therefore, an appropriate value must be chosen to find a good tradeoff between memory and execution time.

ARTEMIS automatically exposes such arguments as tunable parameters, then adjusts them to improve the runtime performance of the program. For instance, ARTEMIS changes line 4 to the code below:

```
1  List<T> l = new ArrayList<>(S);
```

where *S* refers to the exposed parameter associated with the amount of pre-allocated memory. A real example of how ARTEMIS exposes data structures and its parameters for tuning can be seen in Figure 3.1. More specifically, ARTEMIS extracts two data structures, a `List` and a `Map` data structure, and one tunable parameter which referes to the pre-allocated default size of the `List` data structure.

3.2 Darwinian Data Structure Selection and Tuning

This section defines the Darwinian data structure and parameter optimisation problem we solve in this paper.

Definition 1 (Abstract Data Type)**.** *An* Abstract Data Type *(ADT) is class of objects*

whose logical behavior is defined by a set of values and a set of operations [187].

A *data structure* concretely implements an ADT. For the corpus of programs C and the ADT a, the data structure extraction function $dse(a,C)$ returns all data structures that implement a in C. This function is integral to the definition that follows.

Definition 2 (Darwinian Data Structure)**.** *When* $\exists d_0, d_1 \in dse(a,C) \wedge d_0 \neq d_1 \wedge d_0$ *and* d_1 *are observationally equivalent modulo a,* d_0 *and* d_1 *are* Darwinian data structures.

In words, Darwinian data structures are *darwinian* in the sense that they can be replaced to produce program mutants whose fitness we can evaluate. The ADT a has Darwinian data structures when it has more than one data structure that are equivalent over the operations the ADT defines. In Java, `List` is an ADT and `ArrayList`, which implements it, is a data structure. `LinkedList` also implements `List`, so both `ArrayList` and `LinkedList` are Darwinian. For the ADT a and the corpus C, Darwinian data structures are interchangeable. Thus, we can search the variants of $P \in C$ formed by substituting one Darwinian data structure for another to improve P's nonfunctional properties, like execution time, memory consumption or CPU usage.

Just as we needed a function to extract an ADT's data structures from a corpus for Definition 2, we need a function that returns the ADT that a data structure implements: when $d = dse(a,C)$, let $adte(d,C) = a$. Let Γ_D bind fully scope-qualified declarations of names to Darwinian data structures in C. We use Γ_D when creating variants of a program via substitution. We are interested not just searching the space of Darwinian data structures, but also tuning them via their constructor parameters. To this end, we assume without loss of generality that a defines a single constructor c and we let $n.c(x)$ denote calling identifier n's constructor c with parameters $x : \tau$.

To create a variant of $P \in C$ that differs from P only in its k bindings of names to Darwinian data structures or their constructor initialization parameter, we define

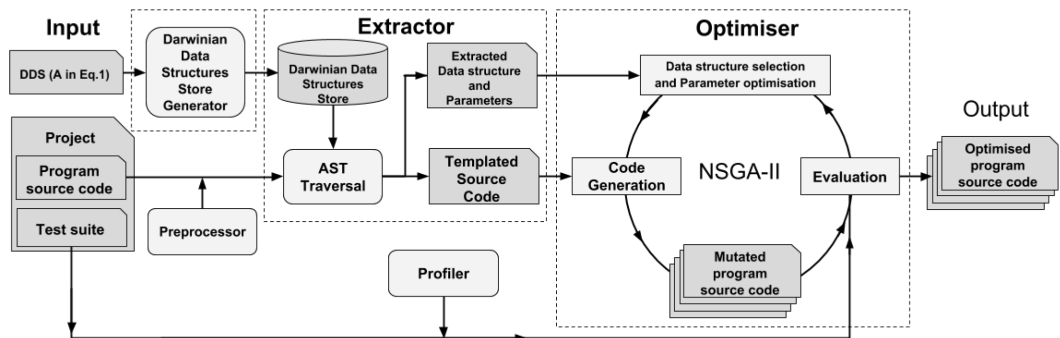


Figure 3.2: System Architecture of ARTEMIS.

$$\phi(P,(n,d_i)^k,d_j^k,x_j) =$$

$$\begin{cases} P[(n.c(x_i))^k/(n.c(x_j))^k], & \text{if } \exists d_i,d_j \text{ s.t. } adte(d_i) \neq adte(d_j) \\ P[(n,d_i)^k/(n,d_j)^k][(n.c(x_i))^k/(n.c(x_j))^k], & \text{otherwise} \end{cases}$$

Definition 3 (Darwinian Data Structure Selection and Tuning)**.** *For the real-valued fitness function f over the corpus C,* the Darwinian data structure and tuning problem *is*

$$\underset{(n_i,d_i)^k \in \Gamma_D^k, d_j^k \in adte(d_i,C)^k, x_j \in \tau}{\arg\max} f(\phi(P,(n_i,d_i),d_j,x_j))$$

This vector-based definition simultaneously considers all possible rebinding of names to Darwinian data structures in P; it is also cumbersome, compared to its point-substitution analogue. We could not, however, simply present a scalar definition and then quantify over all potential DDSS substitutions, as doing so would not surface synergies and antagonisms among the substitutions.

3.3 Artemis

The ARTEMIS's optimisation framework solves the Darwinian Data Structure Selection problem.xxx Figure 3.2 illustrates the architecture with its three main components: the DARWINIAN DATA STRUCTURES STORE GENERATOR (DDSSG), the EXTRACTOR, and the OPTIMISER. ARTEMIS takes the language's Collection API library, the user's application source code and a test suite as input to generate an

optimised version of the code with a new combination of data structures. The DDSSG builds a store that contains data structure transformation rules. The EXTRACTOR uses this store to discover potential data structure transformations and exposes them as tunable parameters to the OPTIMISER (see Section 3.3.2). The OPTIMISER uses a multi-objective genetic search algorithm (NSGA-II [188]) to tune the parameters [18, 70, 189, 190, 191] and to provide optimised solutions (see Section 3.3.4). A regression test suite is used to maintain the correctness of the transformations and to evaluate the non-functional properties of interest. ARTEMIS uses a built-in profiler that measures execution time, memory and CPU usage.

ARTEMIS relies on testing to define a performance search space and to preserve semantics. ARTEMIS therefore can only be applied to programs with a test suite. Ideally, this test suite would comprise both a regression test suite with high code coverage for maintaining the correctness of the program and a performance test suite to simulate the real-life behaviour of the program and ensure that all of the common features are covered [192]. Even though performance test suites are a more appropriate and logical choice for evaluating the non-functional properties of the program, most real world programs in GitHub do not provide such performance test suite. For this reason, we use the regression test suites to evaluate the non-functional properties of the GitHub projects of this study whenever a performance test suite is not available.

3.3.1 Darwinian Data Structure Store

ARTEMIS needs a Darwinian data structure store (DDSS) from which to choose when creating variants. Let A be a set of ADTs known to be Darwinian. A developer can augment this set; Figure 3.3 shows that ARTEMIS knows by default. For our corpus C of Java benchmarks augmented with JDK libraries over A,

$$DDSS = \bigcup_{a \in A} dse(a, C). \tag{3.1}$$

To build the default DDSS for Java, ARTEMIS extracts and traverses each project's class hierarchy, similar to the one illustrated in Figure 3.3. This hier-

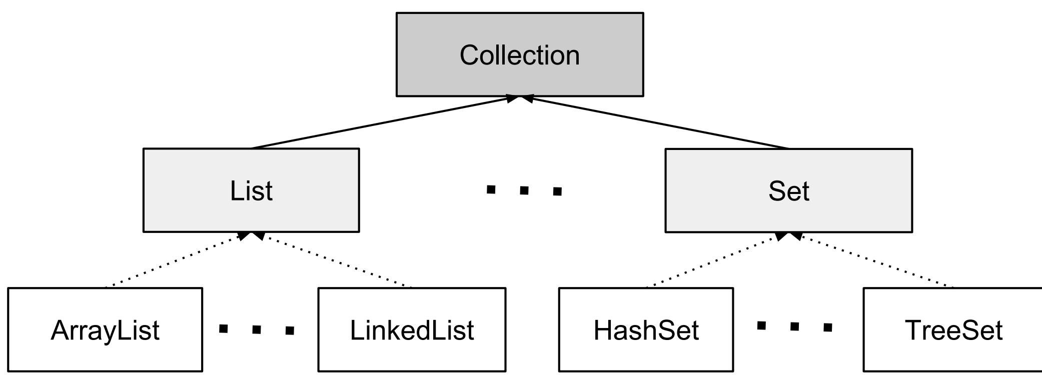


Figure 3.3: DDS in the Java Collections API.

archy shows potential Darwinian data structures of a specific interface. When this traversal finishes, ARTEMIS extracts all the implementations of a particular Darwinian data structure; *e.g.,* `List`, `ArrayList`, `LinkedList`. ARTEMIS considers these implementations mutually replaceable. For Java, a default DDSS is provided by ARTEMIS, which the developer can edit. For other languages, the DDSS can be provided manually by the user and this step can be skipped. The OPTIMISER, described next, uses the store during its search.

The developer can also extend the store with custom user-supplied implementations or with implementations from other third-party libraries such as Google Guava Collections[2], fastutil[3] and Apache Commons Collections[4].

3.3.2 Discovering Darwinian Data Structures

The EXTRACTOR takes as input the program P's source code, identifies Darwinian data structures in P modulo its store (Section 3.3.1), and outputs a scope-qualified list of names of Darwinian data structures and their constructor parameters (*Extracted Data Structures and Parameters* in Figure 3.2). For all $a \in$ DDSS, EXTRACTOR's output realises $dse(a,P)$ (Section 3.2). To mark potential substitions to the transformer, the EXTRACTOR outputs a templated version of the code which replaces the data structure with data structure type identifiers (*Templated Source Code* in Figure 3.2).

[2] https://github.com/google/guava
[3] https://github.com/vigna/fastutil
[4] https://github.com/apache/commons-collections

Table 3.1: Data structure groups.

Abstract Data Type	Implementation
List	ArrayList, LinkedList
Map	HashMap, LinkedHashMap
Set	HashSet, LinkedHashSet
Concurrent List	Vector, CopyOnWriteArrayList
Concurrent Deque	ConcurrentLinkedDeque, LinkedBlockingDeque
Thread Safe Queue	ArrayBlockingQueue, SynchronousQueue, LinkedBlockingQueue, DelayQueue, ConcurrentLinkedQueue, LinkedTransferQueue

To find DARWINIAN data structures, the EXTRACTOR builds an Abstract Syntax Tree (AST) from its input source code. It then traverses the AST to discover potential data structure transformations based on a store of data structures as shown in Table 3.1. For example, when an expression node of the AST contains a `LinkedList` expression, the EXTRACTOR marks this expression as a potential DARWINIAN data structure that can take values from the available `List` implementations: `LinkedList` or `ArrayList`. The EXTRACTOR maintains a copy of the AST, referred to as the REWRITER, where it applies transformations, without changing the initial AST. When the AST transformation finishes, the REWRITER produces the final source code which is saved as a new file.

3.3.3 Code Transformations

When implementing ARTEMIS, we encountered coding practices that vastly increase the search space. Many turn out to be known bad practices [193]. Consider Listing 3.2. In lines 2 and 8, we see two `LinkedList` variables that the Extractor marks DARWINIAN and candidates for replacement by their equivalent `ArrayList` implementation. In these lines, user is violating the precept to "program to the interface", here `List`, but is, instead, declaring the variable to have the concrete, data structure not ADT, type `LinkedList`. This bad practice [193] adds dependencies to the code, limiting code reuse. They are especially problematic for ARTEMIS, because they force ARTEMIS to apply multiple transformations to replace and optimise the data

```
1  void func1(){
2   LinkedList<T> v;
3   v = new LinkedList<>();
4   v.add(new T());
5   int value = func3(v);
6  }
7  void func2(LinkedList<T> v){
8   LinkedList<T> v1 = new LinkedList<>();
9   int value = func3(v1);
10 }
11 int func3(LinkedList<T> v){
12  T t = v.get(0);
13  return 2*t.value;
14 }
```

Listing 3.2: Code to illustrate bad practices.

structure. Further, `func3` takes a `LinkedList` as a parameter, not `List`, despite the fact that it only calls the `get` method defined by `List` on this parameter. This instance of violating the "program to the interface" precept triggers a compilation error if ARTEMIS naïvely changes `func1`'s type. ARTEMIS transforms the code to reduce the OPTIMISER's search space and handle these bad practices. ARTEMIS supports thee transformations - parserless, supertype, and profiler.

The *parserless* mode changes greadily each appearance of a Darwinian implementation. When optimising `List`, it exhaustively tries every implementation of `List` for every `List` variable. It is parserless, since it needs only a regular expression to identify rewritings. This makes it simple, easily portable to other languages, and fast, so it is ARTEMIS' default. However, it generates invalid programs and a large search space.

ARTEMIS' *sypertype* transformation converts the type of a Darwinian implementation to that of their Darwinian ADT, for example `LinkedList<T>` $\rightarrow$ `List<T>` on lines, 2,7,8 and 11. For Listing 3.2, this tranformation exposes only two DDS to the OPTIMISER and produces only syntactically valid code. To implement this transformation, ARTEMIS invokes Eclipse's re-factoring functionality via its API, then validates the result. ARTEMIS aims to be language-agnostic without any additional dependencies on language specific tools. For this case, ARTEMIS auto performs this

transformation by adding the supertype as an equivalent parameter in the store of data structures. Whenever the AST visitor traverses a variable or parameter declaration expression it may replace the DARWINIAN data structure with its supertype.

"All data structures are equal, but some data structures are more equal than others" [5]; some DDS affect a program's performance more than others, as when one stores only a few, rarely accessed items. To rank DDS, ARTEMIS *profiles* its input program to identify costly methods. The EXTRACTOR uses this info to identify the subset of a program's DDS worth considering for optimisation. ARTEMIS' instrumentation is particularly important for large programs.

3.3.4 Search Based Parameter Tuning

The OPTIMISER searches a combination of data structures that improves the performance of the initial program while keeps the original functionality. Practically, we can represent all those data structures as parameters that can be tuned using Search Based Software Engineering approaches [20]. Because of the nature of the various conflicting performance objectives, the problem we faced here requires a multi-objective optimisation approach to search the (near) optimal solutions. We used evolutionary optimisations algorithm as experiments in which data structure replacement abd constructor parameters was done randomly, it generated only 20% viable variants. Under GP, viable variants increased each generation, reaching nearly 100%.

An array of integers is used to represent the tuning parameters. Each parameter refers either to a Darwinian data structure or to the initial size of that data structure. If the parameter refers to a data structure, its value represents the index in the list of Darwinian data structures. The OPTIMISER keeps additional mapping information to distinguish the types of the parameters. For each generation, the NSGA-II applies tournament selection, followed by a uniform crossover and a uniform mutation operation. In our experiments, we designed fitness functions to capture execution time, memory consumption, and CPU usage. After fitness evaluation, ARTEMIS applies standard non-dominated selection to form the next generation. ARTEMIS

[5] Adapted from "Animal Farm" by George Orwell

repeats this process until the solutions in a generation converge. At this point, ARTEMIS returns all non-dominated solutions in the final population.

Search Space size: We used GA because the search space is huge. Let D be the definitions of darwinian data structures in program P. Let I be the number of implementations for a particular $d \in D$. The size of the search space is:

$$\prod_{d \in D} I(d) * |dom(d.c)|, \text{ where } d.c \text{ is } d\text{'s constructor.} \qquad (3.2)$$

3.3.5 Deployability

ARTEMIS provides optimisation as a cloud service. To use the service, developers only need to provide the source code of their project in a Maven build format and a performance test suite invoked by `mvn test`. ARTEMIS returns the optimised source code and a performance report. ARTEMIS exposes a RESTful API that developers can use to edit the default store of Darwinian data structures. The API also allows developers to select other Search Based algorithms; the OPTIMISER uses NSGA-II by default. To use our tool from the command line, a simple command is used:

```
1 ./artemis input—program—src
```

where this command defaults to ARTEMIS's built in DDSSG. ARTEMIS writes the source of an optimized variant of its input for each measure. ARTEMIS also supports optional parameters to customise its processing.

3.4 Evaluation

To demonstrate the performance improvements that ARTEMIS automatically achieves and its broad applicability, we applied it to three corpora: 8 popular GitHub projects, 5 projects from the Dacapo Benchmark, and 30 projects, filtered to meet ARTEMIS's requirements, then sampled uniformly at random from Github. To show also that ARTEMISis language-agnostic, we applied it to optimise Guetzli[6] (Section 3.4.3), a JPEG encoder written in C++.

[6] https://github.com/google/guetzli

3.4.1 Corpus

ARTEMIS requires projects with capable build systems and an extensive test suites. These two requirements entail that ARTEMIS be able to build and run the project against its test suite. ARTEMIS is language-agnostic but is currently only instantiated for Java and C++, so it requires Java or C++ programs.

Our first corpus comprises eight popular GitHub projects. We selected these eight to have good test suites and be diverse. We defined popular to be projects that received at least 200 stars on GitHub. We deemed a test suite to be good if its line coverage met or exceeded 70%. This corpus contains projects, usualy, optimised and peer code-reviewed by experienced developers. We applied ARTEMIS on those projects to investigate whether it can provide a better combination of data structures than those selected by experienced human developers.

This first corpus might not be representative, precisely because of the popularity of its benchmarks. To address this threat to validity, we turned to the DaCapo benchmarks [126]. The authors of DaCapo built it, from the ground up, to be representative. The goal was to provide the research community with realistic, large scale Java benchmarks that contain a good methodology for Java evaluation. Dacapo contains 14 open source, client-side Java benchmarks (version 9.12) and they come with built-in extensive evaluation. Each benchmark provides accurate measurements for execution time and memory consumption. DaCapo first appeared in 2006 to work with Java v.1.5 and has not been further updated to work with newer versions of Java. For this reason, we faced difficulties in compiling all the benchmarks and the total number of benchmarks were reduced to 5 out of 14 (see Table 3.2). In this corpus we use the following five: `fop`, `avrora`, `xalan`, `pmd` and `sunflow`.

Because of its age and the fact that we are only using subset of it, our DaCapo benchmark may not be representative. To counter this threat, we uniformly sampled projects from GitHub (Figure 3.4). We discarded those that did not meet ARTEMIS's constraints, like being equipped with a build system, until we collected 30 projects (see Figure 3.4). Those projects are diverse, both in domain and size. The selected projects include static analysers, testing frameworks, web clients, and graph process-

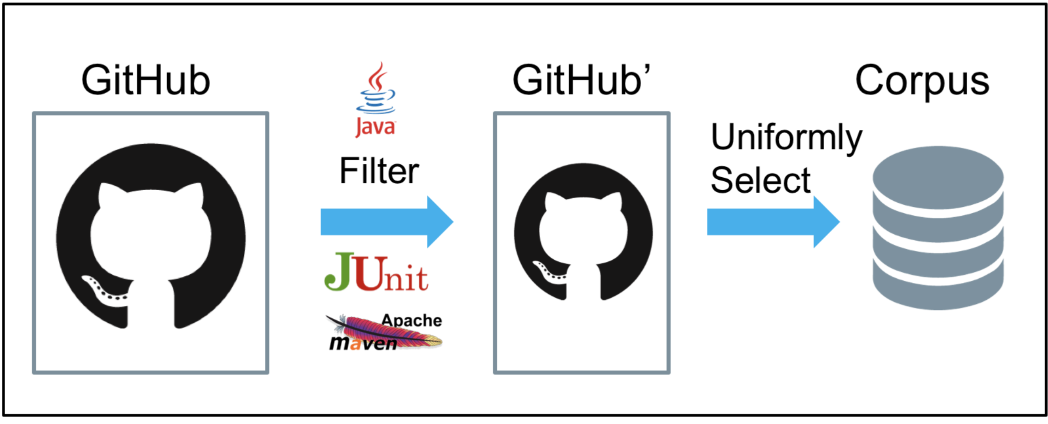


Figure 3.4: Process of generating the uniformly selected at random Github corpus. First, a Github project is selected randomly, then if it contains a maven build system, it compiles and its tests run succesfully it is added in the corpus.

Table 3.2: DaCapo projects. #Star, #Loc are the number of stars and line of codes respectively. All these subjects are retrieved from the official Dacapo Benchmark page on 11th Jan 2017.

Dacapo Benchmark Project	#Star	#LoC
fop	85	213,244
avrora	6	76,155
xalan	9	170,710
pmd	1,998	117,538
sunflow	44	168,740

ing applications. Their sizes vary from 576 to 94K lines of code with a median of 14881. Their popularity varies from 0 to 5642 stars with a median of 52 stars per project. The median number of tests is 170 and median line coverage ratio is 72% (see Table 3.3).

Collectively, we systematically built these corpora to be representative in order to demontrate the general applicably of the ARTEMIS' optimization framework. The full list of the programs used in this experimental study are available online[7] in the project's website.

Table 3.3: Subject projects studied in this research. #Star, #Loc, #Test, and Coverage(%) are the number of stars, line of code, number of tests, and the line coverage ratio, respectively. All these subjects are retrieved through GitHub on 11th Jan 2017.

Uniformly Selected Projects	#Star	#LoC	#Test	Coverage(%)
adyliu/jafka	10,698	11,944	71	71.5%
zilaiyedaren/zxing	21,164	42,521	378	68.3%
HotelsDotCom/plunger	26	3,865	175	82.8%
johnewart/shuzai	0	680	8	56.4%
rayzeng/fqueue	0	3,929	10	48.9%
lsloan/OpenLRS	0	5,229	28	26.8%
BiBiServ/jobproxy	2	2,524	28	28.1%
fuinorg/event-store-commons	4	12,652	208	58.2%
roby-rodriguez/rubix-verifier	0	576	3	81.0%
apache/commons-validator	83	23,520	527	95.9%
kinow/tap-plugin	4	2,804	34	62.7%
lynchmaniac/poilight	2	2,192	35	89.3%
gabe-alex/HospitalInfectionsMonitoringSystem	0	1,033	1	12.5%
dick-the-deployer/dick-worker	0	1,348	18	66.8%
light-4j	885	11,663	38	42.1%
truth	1,584	2,857	33	31.9%
documents4j	193	33,308	640	68.5%
jsoniter	0	48,267	792	59.6%
cmn-codec	176	24,927	19	27.5%
tablesaw	1,292	13,922	75	55.5%
querqy	0	1,058	1	9.2%
mapper	3	1,343	18	31.1%
bootique	0	3,207	61	90.7%
Glowstone	356	12,956	63	35.0%
rest-assured	8	14,962	78	18.1%
milo	0	29,116	873	76.5%
javapoet	832	39,272	170	48.6%
guice	4	14,504	114	55.5%
TelegramBots	741	94,659	588	27.1%
epubcheck	6	705	16	66.7%
Popular Projects	**#Star**	**#LoC**	**#Test**	**Coverage(%)**
google/google-http-java-client	572	20,637	636	69.1%
jOOL	898	26,128	1,175	90.9%
joda-time	3,890	86,192	4226	86.9%
google/jimfs	1,148	17,244	5,380	91.7%
google/gson	7,525	24,395	1,016	94.2%
cglib	539	36,513	974	90.2%
solo	5,642	27,820	977	89.0%
twitter/GraphJet	344	14,881	94	90.7%

Table 3.4: Hardware characteristics.

Characteristic	Value
processor	Intel 8 cores E5-2673 v3 CPU
clock speed	2.5 GHz per core
memory	14 GB 1600 MHz DDR3
disk	250 GB SSD
operating system	Ubuntu 16.04.4 LTS

3.4.2 Experimental Setup

Experiments were conducted using Microsoft Azure^{TM} D4-v2 machines with one Intel E5-2673v3 CPU featuring 8 cores and 14GB of DRAM and built with Oracle JDK 1.8.0 and Ubuntu 16.04.4 LTS (Table 3.4).

Performance measurements may lead to incorrect results if not handled carefully [194]. Thus, a statistical rigorous performance evaluation is required [3, 140, 195]. To mitigate instability and incorrect results, we differentiate VM start-up and steady-state. We ran our experiments in a fresh Azure VM that contained only the JVM and the subject. We use JUnit, which runs an entire test suite in a single JVM. We manually identified and dropped startup runs, then we spot-checked the results to confirm that the rest of the runs achieved a steady state and were exhibiting low variance. All of the means and medians we reported fall within the computed interval with 95% confidence. To assure the accuracy and reduce the bias in the measurement, program profiling period was set as 0.1 seconds, and each generated solution was run for more than 30 simulations. Also we use Mann Whitney U test [196] to examine if the improvement is statistically significant.

To measure the memory consumption and CPU usage of a subject program, we use the popular JConsole profiler[8] because it directly handles JDK statistics and provides elegant API. We extended JConsole to monitor only those system processes belonging to the test suite. We use Maven Surefire plugin[9] to measure the test suite's execution time because it reports only the execution time of each individual test,

[7] https://darwinianoptimiser.com/corpus
[8] http://openjdk.java.net/tools/svc/jconsole/
[9] http://maven.apache.org/components/surefire/maven-surefire-plugin/

excluding the measurement overhead that other Maven plugins may introduce.

For the OPTIMISER, we chose an initial population size of 30 and a maximum number of 900 function evaluations. We used the tournament selection (based on ranking and crowding distance), simulated binary crossover (with crossover probability 0.8) and polynomial mutation (with the mutation probability 0.1). We determined these settings from calibration trials to ensure the maturity of the results. Since NSGA-II is stochastic, we ran each experiment 30 times to obtain statistical significant results. We used a genetic algorithm such as NSGA-II for the optimisation process as it can find better solutions and faster than random search, and it can scale to a larger search space. In our experiments, we initially started by using random search, but we realised that ARTEMIS would generate many projects that would fail compiling and the search process would be slow. On the other side, NSGA-II converged towards better solutions after the first generations. Because of this difference that we found when running ARTEMIS, we did focus on NSGA-II, and we do not present any experiments about using random search.

3.4.3 Research Questions and Results Analysis

ARTEMIS aims to improve all objectives at the same time. Therefore the first research question we would like to answer is:

RQ1: *What proportion of programs does* ARTEMIS *improve?*

To answer RQ1, we applied ARTEMIS to our corpus. We inspected the generated optimal solutions from 30 runs of each subject by examining the dominate relation between the optimal and inital solutions regarding the optimisation objectives. We introduce the terms *strictly dominate relation* and *non-dominated relation* to describe the relation. Defined by Zitzler et al. [197], a solution *strictly dominates* another solution if it outperforms the latter in all measures. A solution is *non-dominated* with another solution if both outperform the other in at least one of the measures.

For DaCapo, ARTEMIS found at least one strictly dominant solution for 4 out of 5 projects; it found no such solution for `sunflow`. It found 1072 solutions, from which 3% are strictly dominant (median is 5.5 solutions per project) and 64% are

non-dominated (median is 18 solutions per project).

For the popular Github projects, ARTEMIS found at least one strictly dominant solution for all 8 projects. The total number of solutions found is 10218 and 16% of them are strictly dominant (median is 50 solutions per project) and 59% are non-dominated (median is 749.5 solutions per project).

For the sampled Github projects, ARTEMIS found a strictly dominant solution for 25 out of 30 projects, but found no solution for projects `rubix-verifier`, `epubcheck`, `d-worker`, `telegrambots` and `fqueue`. It found 27503 of which 10% of them are strictly dominant (median is 24 solutions per project) and 66% are non dominant (median is 125 solutions per project). With these results, we answer *RQ*1 affirmatively:

Finding1: ARTEMIS finds optimised variants that outperform the original program in at least one measure for *all* programs in our representative corpus.

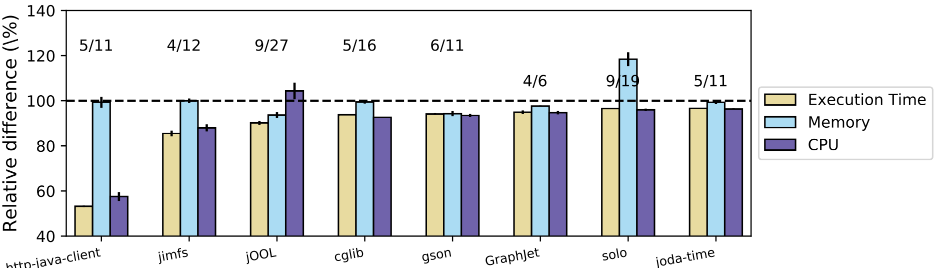
This finding understates ARTEMIS's impact. Not only did it improve at least one measure for *all programs*, ARTEMIS found solutions that improve *all measures* for 88% of the programs.

Having found that ARTEMIS finds performance improvements, we ask "How good are these improvements" with:

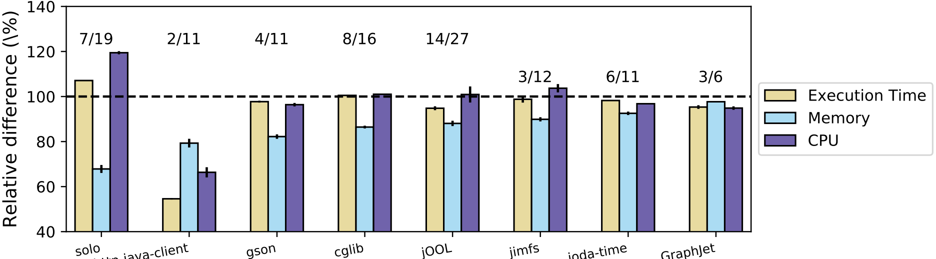
RQ2: *What is the average improvement that* ARTEMIS *provides for each program?*

Though ARTEMIS aims to improve all candidate's measures, it cannot achieve that if improvements are antagonistic. In some domains, it is more important to significantly improve one of the measures than to improve slightly all measures; *e.g.,* a high frequency trading application may want to pay the cost of additional memory overhead in order to improve the execution time. Our intuition is that the OPTIMISER will find many solutions on the Pareto-front and at least one of them will improve each measure significantly.

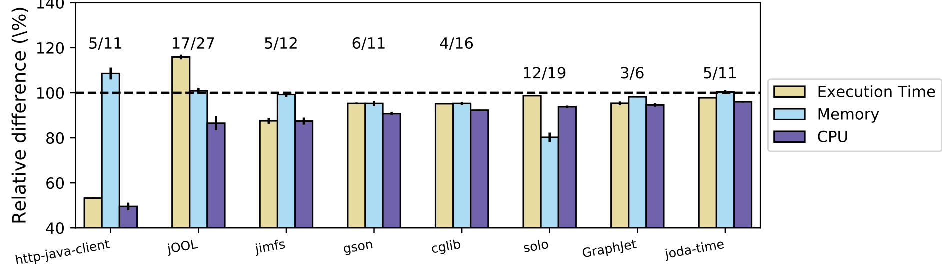
We answer *RQ*2 quantitatively. We report the maximum improvement (median value with 95% confidence interval) for execution time, memory and CPU usage for



(a) Best execution time of popular GitHub programs. The median value is 93.3%, mean is 86.4%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 4 and mean is 5.



(b) Best memory consumption of popular GitHub programs. The median value is 86% and mean is 84%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 4 and mean is 5.85.



(c) Best CPU usage of popular GitHub programs. The median value is 90.3% and mean is 84.6%. Median number of DDS is 12 and mean is 14.6. Median number of DDS changes is 5 and mean is 7.42.

Figure 3.5: Answers *RQ2*. Description.

each subject of the three corpora. We use bar charts with error bars to plot the three measures for each program. In Y axis, we represent the percentage of improvement for each measure. A value less than 100% represents an improvement and a value greater than 100% means degradation; *e.g.,* 70% memory consumption implies that the solution consumes 70% of the memory used in the input program.

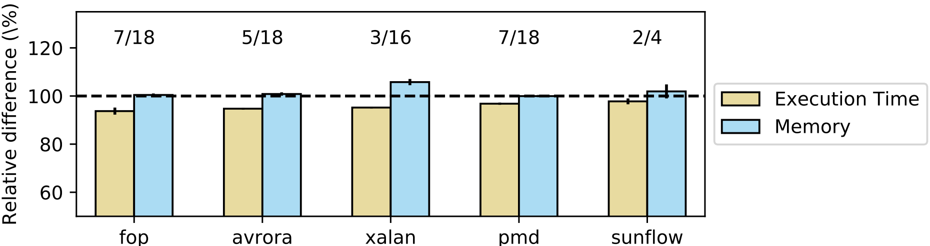
Selected popular GitHub programs. Figure 3.5a presents the three measures of the solutions when the execution time is minimised, for each program from the

popular GitHub programs. We observe that ARTEMIS improves the execution time of *every* program. `google-http-java-client`'s execution time was improved the most; its execution time was reduced by M=46%, 95% CI [45.6%, 46.3%]. We also notice that this execution time improvement did not affect negatively the other measures, but instead the CPU usage was reduced by M=41.6%, 95% CI [39.6%, 43.6%] and memory consumption remained almost the same. The other interesting program to notice from this graph is `solo`, a blogging system written in Java; its execution time improved slightly by 2% but its memory consumption increased by 20.2%. Finally, for this set of solutions, the median execution time improvement is 14.13%, whilst memory consumption slightly increased by 1.99% and CPU usage decreased by 3.79%. For those programs, ARTEMIS extracted a median of 12 data structures and the optimal solutions had a median of 4 data structures changes from the original versions of the program.

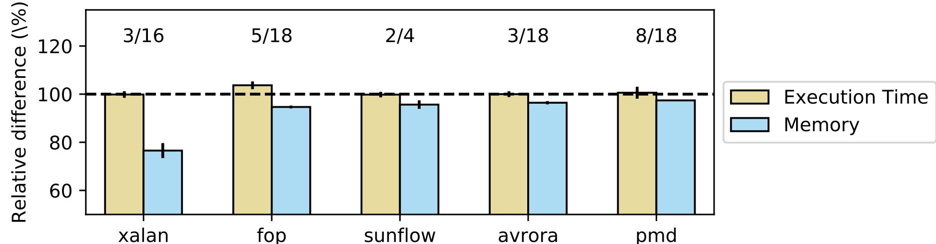
Figure 3.5b shows the solutions optimised for memory consumption. We notice that ARTEMIS improves the memory consumption for all programs, with a median value of 14%. The execution time was improved by a median value of 2.8% for these solutions, while the median value of CPU usage is slightly increased by 0.4%. We notice that `solo` has the best improvement by M=31.1%, 95% CI [29.3%, 33%], but with an increase of M=8.7%, 95% CI [8.5%, 8.9%] in execution time and M=21.3%, 95% CI [20.6%, 22%] in CPU usage. `Graphjet`, a real-time graph processing library, has the minimum improvement of M=0.9%, 95% CI [0.6%, 1.1%]. The optimal solutions had a median of 4 data structures changes per solution.

Figure 3.5c presents solutions optimised for CPU usage. The median CPU usage improvement is 9.7%. The median value of execution time improved by 5.2% and the median value of memory consumption improved by 2.3%. The program with the most significant improvement in CPU is `http-java-client` with M=49.7%, 95% CI [48%, 51.4%], but with a decrease in memory of M=9.8%, 95% CI [7.5%, 12.9%]. The optimal solutions make a median of 5 data structures changes to the original versions of the program.

DaCapo. Figure 3.6 presents all solutions optimised for execution time and



(a) Best execution time of the Dacapo benchmark. The median value is 95.20% and mean is 95.6%. Median number of DDS is 18 and mean is 14.8. Median number of DDS changes is 5 and mean is 4.8.



(b) Best memory consumption of the Dacapo benchmark. The median value is 95.7% and mean is 92.1% . Median number of DDS is 18 and mean is 14.8. Median number of DDS changes is 3 and mean is 4.2.

Figure 3.6: Answers *RQ2*. Description.

memory consumption for the DaCapo benchmark. We used only two measures for the DaCapo benchmark as those were the ones built in the benchmark suite. We chose not to extend or edit the profiling method of DaCapo, to avoid the risk of affecting the validity of its existing, well tested profiling process.

ARTEMIS found solutions that improve the execution time for every program without affecting significantly the memory consumption, except project `xalan` which had improvement (M=4.8%, 95% CI [4.6%, 5.7%] in execution time but with an increase (5.8%, 95% CI [3.5%, 7%]) in memory consumption. All solutions for optimised memory consumption did not affect execution time, except for a slight increase for program `fop`. Finally, for this set of solutions, the median percentage of execution time improvement is 4.8%, and 4.6% for memory consumption. For this set of programs, ARTEMIS extracted a median of 18 data structures per program, and the optimal solutions had a median of 5 data structures changes for the execution time optimised solutions and 4 for the memory optimised solutions.

Sampled GitHub programs. Figure 3.7, Figure 3.8, Figure 3.9 present all solu-

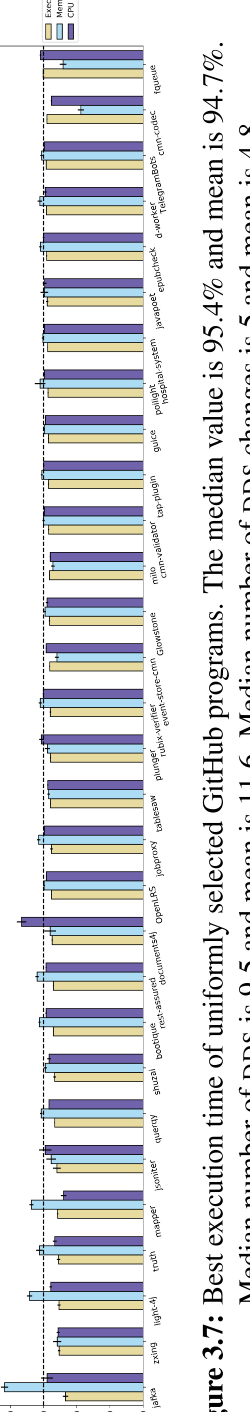


Figure 3.7: Best execution time of uniformly selected GitHub programs. The median value is 95.4% and mean is 94.7%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.8.

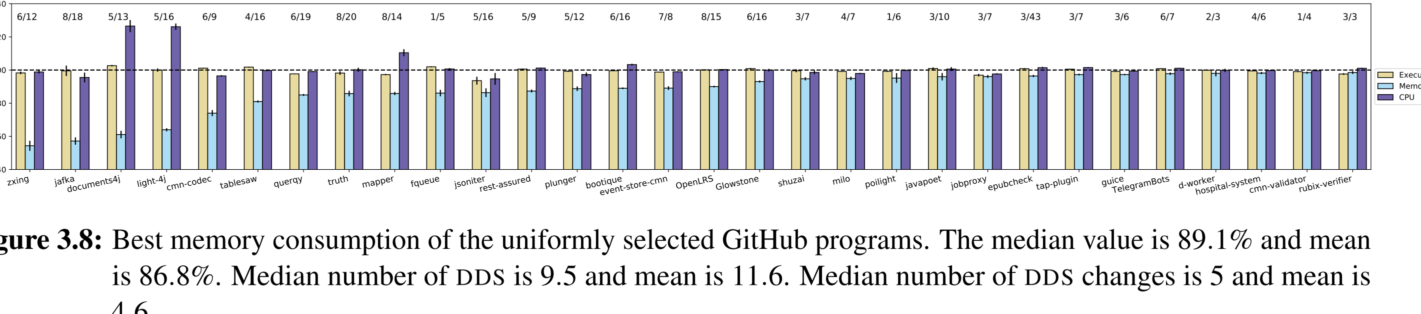


Figure 3.8: Best memory consumption of the uniformly selected GitHub programs. The median value is 89.1% and mean is 86.8%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.6.

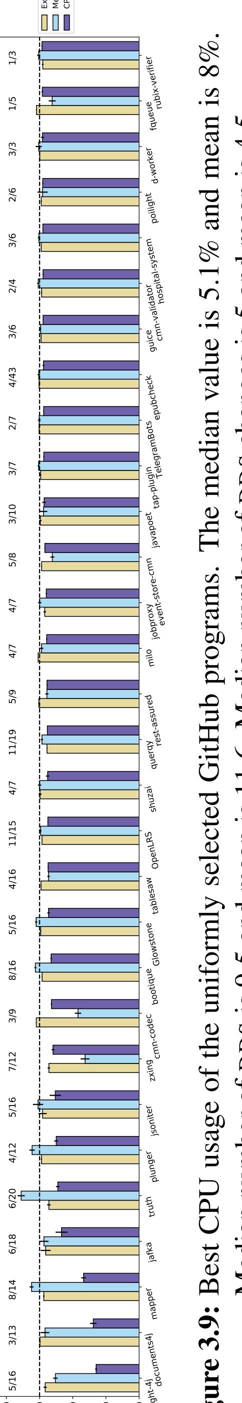


Figure 3.9: Best CPU usage of the uniformly selected GitHub programs. The median value is 5.1% and mean is 8%. Median number of DDS is 9.5 and mean is 11.6. Median number of DDS changes is 5 and mean is 4.5.

tions optimised for execution time, memory consumption and CPU usage for the sampled GitHub programs. As with the previous corpora, ARTEMIS found solutions that improved each measure significantly. ARTEMIS improves the median value of execution time across all projects by 4.6%, memory consumption by 11.4% and CPU usage by 4.6%.

ARTEMIS found solutions with antagonistic improvement for projects `jafka` and `documents4j`. ARTEMIS found a solution that improves the execution time of `jafka`, a distributed publish-subscribe messaging system, by M=12%, 95% CI [11.2%, 13.6%], but also increases its memory consumption by M=23.6%, 95% CI [21.4%, 25.7%]. It also found a solution that improves the memory consumption of `documents4j` (M=38%, 95% CI [38%, 41%]) but introduced extra CPU usage M=26.1%, 95% CI [24.2%, 28%]. A median of 9.5 data structures were extracted and the optimal solutions had a median of 5 data structures changes from the original versions of the program.

Observing again the numbers across the three corpora, we can say that they are quite consistent, showing that ARTEMIS finds optimal solutions that improve significantly the different optimisation measures. We also see that the number of Darwinian Data structures extracted (between 9.5 and 18) and the optimal solutions DDS changes (between 4 and 5) are quite similar for the three corpora.

Analysing all results from the 3 corpora we conclude the discussion of RQ2 with:

Finding2: ARTEMIS improves the median across all programs in our corpus by 4.8% execution time, 10.2% memory consumption, and 5.1% CPU usage.

RQ3: *Which Darwinian data structures does* ARTEMIS *find and tune?*

We ask this question to understand which changes ARTEMIS makes to a program. Table 3.5 contains the transformations ARTEMIS applied across all optimal solutions. We see that the most common transformation for all measures is replacing `ArrayList` with `LinkedList`, it appears 91, 86 and 87 times respectevely across all measures. This transformation indicates that most developers prefer to

Table 3.5: DDS changes for optimal solutions across all measures.

Tranformation	Time	Memory	CPU
HashMap -> LinkedHashMap	60	53	57
LinkedList -> ArrayList	16	13	18
HashSet -> LinkedHashSet	22	21	21
LinkedBlockingQueue -> LinkedTransferQueue	1	2	2
ArrayList -> LinkedList	91	86	87
LinkedHashSet -> HashSet	7	8	5
Vector -> CopyOnWriteArrayList	1	0	2
LinkedHashMap -> HashMap	17	23	19

use `ArrayList` in their code, which in general is considered to be faster, neglecting use cases in which `LinkedList` performs better; *e.g.,* when the program has many list insertion or removal operations. Except `HashMap` to `LinkedHashMap`, the other transformations happen relatively rare in the optimal solutions. Last, the median number of lines Artemis changes is 5.

Finding3: ARTEMIS extracted a median of 12 Darwinian data structures from each program and the optimal solutions had a median of 5 data structure changes from the original versions of the program.

RQ4: *What is the cost of using* ARTEMIS?

In order for ARTEMIS to be practical and useful in real-world situations, it is important to understand the cost of using it. The aforementioned experimental studies reveal that, even for the popular programs, the existing selection of the data structure and the setting of its parameters may be sub-optimal. Therefore, optimising the data structures and their parameters can still provide significant improvement on non-functional properties. To answer this research question, the cost of ARTEMIS for optimising a program is measured by the cost of computational resources it uses. In this study, we used a Microsoft Azure™ D4-v2 machine, which costs £0.41 per hour at a standard Pay-As-You-Go rate[10], to conduct all experiments.

The experiments show that an optimisation process takes 3.05 hours on average

[10] https://azure.microsoft.com/en-gb/pricing/

for all studied subjects. The program `GraphJet` and `jimfs` are the most and the least time-consuming programs respectively, with 19.16 hours and 3.12 minutes optimisation time. Accordingly, the average cost of applying ARTEMIS for the subjects studied is £1.25, with a range from £0.02 to £7.86. The experimental results show that overall cost of using ARTEMIS is negligible compared to a human software engineer, with the assumption that a competent software engineer can find those optimisation in a reasonable time.

ARTEMIS transforms the selection of data structure and sets parameters by rewriting source code, thereby allowing human developers to easily investigate its changes and gain insight about the usage of data structures and the characteristics of the program.

Finding4: The cost of using ARTEMIS is negligible, with an average of £1.25 per project, providing engineers with insights about the optimal variants of the project under optimisation.

RQ5: *Is* ARTEMIS *practical and applicable to other languages*?

To show the versatility of the ARTEMIS framework, we ask RQ2, RQ3 and RQ4 over Google `guetzli`, a very popular JPEG encoder written in C++. We used the STL containers and their operations as Darwinian data structures. More specifically, we considered the `push_back` and `emplace_back` as equivalent implementations of the same functionality and exposed those as tunable parameters to ARTEMIS's optimiser. We collected a random sample of images (available online [11]) and used it to construct a performance suite that evaluates the execution time of `guetzli`.

We answer RQ2 by showing that ARTEMIS found an optimal solution that improves execution time by 7%. We answer RQ3 by showing that ARTEMIS extracted and tuned 25 parameters and found an optimal solution with 11 parameter changes. ARTEMIS spent 1.5 hours (costs £0.62) to find optimal solutions which is between the limits reported in RQ4. Last, we spent approximately 4 days to extend ARTEMIS

[11] `http://darwinianoptimiser.com/corpus`

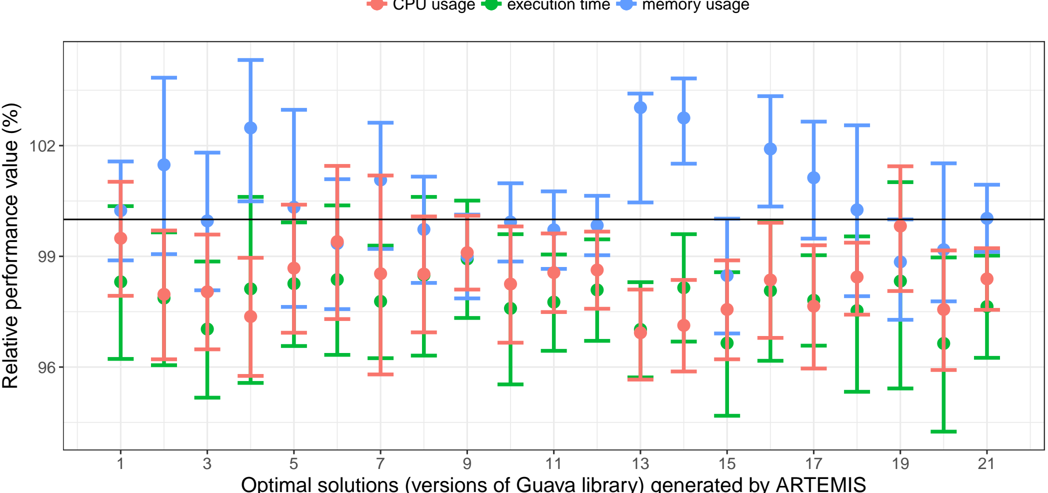


Figure 3.10: Optimal solutions with large improvement in at least one measure.

to support C++, using the *parserless* mode.

> **Finding5:** ARTEMIS is language-agnostic and can succesfully optimise code in other programming languages, such as C++.

3.4.4 Optimising Google Guava library using ARTEMIS

ARTEMIS's potential impact may be bigger if applied on libraries, because all of their clients will enjoy any improvements ARTEMIS finds for them. To assess how effectively ARTEMIS can improve libraries, we used Guava[1] as an instance of its application. Guava is a very popular open-source set of common libraries for Java. It consists of $252,688$ Lines of Code, which are tested by $1,674,425$ test cases with 61.7% branch coverage.

First, we wanted to see what is the improvement that ARTEMIS can achieve for Google Guava on each of the objectives and how the other objectives are affected. To report correct results, we compute the mean response time and report the 95% confidence interval. Then, we use effect size [198] for measuring the performance impact. To quantify the effects, we use Cohen's d [198] strength values: small ($0.2 < d \leq 0.5$), medium ($0.5 < d \leq 0.8$) and large ($0.8 < d$). In Figure 3.10, we plot the mean values for the optimal solutions that contain at least one large improvement

[1] `https://github.com/google/guava`

for one of the three measurements. The maximum improvement for each measure is 9% execution time, 13% memory usage and 4% CPU usage.

Next, we investigate how many of the solutions strictly dominate the original program. A solution is said to strictly dominate another if it outperforms the other in all measures. If ARTEMIS can provide solutions that strictly dominate the original program, those solutions can be very valuable because they represent options to improve the program without sacrificing any of the other objectives. The number of strictly dominating solution for Guava was 14 out of 51 final solutions. Those 14 solutions provide a wide range of options for users to choose depending on their favour of different objectives.

Next, we ask what is the computational cost of ARTEMIS when we run it for Google Guava. An extremely high computational cost may make the system impractical to use in real-world situations. Therefore, we measured its cost on Guava subject in terms of machine hours. In this study, a Microsoft Azure D4-v2 machine, which costs £0.41 per hour[2], was used to conduct all experiments. This cost of using is negligible compared to a human software engineer. Moreover, ARTEMIS transforms the selection of data structure and sets the parameter on source code level, which means such optimisation does not need to be carried frequently.

Last, we question how many Darwinian Data Structures where selected and tuned. To minimise the search space we applied ARTEMIS only to the most used code in Guava, as identified by the preprocessor. As a result, ARTEMIS extracted only 6 Darwinian data structures in total from the Guava library. Across all the optimal solutions that ARTEMIS produced, 1 to 6 data structures were changed in each solution, with a median of 3 data structures. For instance, ARTEMIS replaced `HashMap` with `LinkedHashMap` in 42 of the 135 changes across all optimal solutions.

In those experiments, we showed how ARTEMIS automatically selects and optimises the data structures and their arguments in popular libraries such as Google Guava. On a large real-world system, Guava, ARTEMIS found 9% improvement on execution time, 13% improvement on memory consumption and 4% improvement

[2] https://azure.microsoft.com/en-gb/pricing/

on CPU usage separately, and 27.45% of the final solutions provides improvement without sacrificing other objectives. Lastly, we estimated the cost of optimising Guava in machine hours. With a price of £0.41 per machine hour, the cost of optimising a real-world system such as Guava in this study is less than £7.85. Therefore, we conclude that ARTEMIS is a practical tool for optimising data structures in large real-world libraries.

3.5 Threats to Validity

Section 3.4.1 discusses the steps we took to address the threats to the external validity of the results we present here. In short, we built three subcorpora, each more representative than the last, for a total of 43 programs, diverse in size and domain. The biggest threat to the internal validity of our work is the difficulty of taking accurate performance measurements of applications running on VM, like the JVM. Section 3.4.2 details the steps, drawn from best practice, we took to address this threat. In essence, we conducted calibration experiments to adjust the parameters such that the algorithm converges quickly and stops after the results become stable. For measuring the non-functional properties, we carefully chose JConsole profiler that directly gathers runtime information from JDK, such that the measurement error is minimised. Moreover, we carefully tuned JConsole to further improve the precision of the measurements by maximising its sampling frequency such that it does not miss any measurements while minimising the CPU overhead. To cater for the stochastic nature of ARTEMIS and to provide the statistic power for the results, we ran each experiment 30 times and manually checked that experiments had a steady state and exhibited low variance.

The scalability of our approach is another threat to external validity. The subjects involved in this study vary in size, from hundreds of lines of code to almost a million lines of code. According to our experimental results, the effectiveness of our approach remains the same from small subjects to large subjects. Our approach also makes small source code changes (5 lines per project in average). Therefore, it is expected that our approach will scale up to even larger subjects and this threat is

reduced.

3.6 Summary

In this chapter, we introduced ARTEMIS, a novel multi-objective multi-language search-based framework that automatically selects and optimises Darwinian data structures and their arguments in a given program. ARTEMIS is language agnostic, meaning it can be easily adapted to any programming language; extending ARTEMIS to support C++ took approximately 4 days. Given as input a data structure store with Darwinian implementations, it can automatically detect and optimise them along with any additional parameters to improve the non-functional properties of the given program. In a large empirical study on 5 DaCapo benchmarks, 30 randomly sampled projects and 8 well-written popular Github projects, ARTEMIS found *strong* improvement for all of them. On extreme cases, ARTEMIS found 46% improvement on execution time, 44.9% improvement on memory consumption, and 49.7% improvement on CPU usage. ARTEMIS found such improvements making small changes in the source code; the median number of lines ARTEMIS changes is 5. Thus, ARTEMIS is practical and can be easily used on other projects. At last, we estimated the cost of optimising a program in machine hours. With a price of £0.41 per machine hour, the cost of optsimising any subject in this study is less than £8, with an average of £1.25. Therefore, we conclude that ARTEMIS is a practical tool for optimising the data structures in large real-world programs.

Chapter 4

Upgradeable Ethereum Smart Contracts

The main concept of the Ethereum Blockchain is that smart contracts are self-managed entities that function independently based on the rules written in the code, without the need of any trusted party. The Blockchain consensus protocol enforces the reliable execution of smart contracts and is considered the only trusted entity. Ethereum, being a fully decentralised Blockchain network, allows anyone to participate and interact with any published smart contract, unlike traditional distributed applications on cloud servers that limit who can interact with them. This property of the Blockchain increases the security risk when developing smart contracts because they are more vulnerable to participants that want to exploit potential bugs and security issues.

When developing a smart contract, the developers should take into account all those risks and verify carefully its correctness (*i.e.,* if its implementation follows the best available practices) and validity (*i.e.,* if the code matches the business logic of the contract). However, writing bug-free code for smart contracts is a very challenging task [174, 175]. The manual editing of contracts is error-prone, and the existing available formal verification tools cannot capture all potential bugs that may appear in it [178]. This difficulty further increases by the fact that the code of a smart contract is immutable. As a result, many buggy smart contracts have appeared in the Blockchain, having a significant negative impact [174, 175] to their users and to the

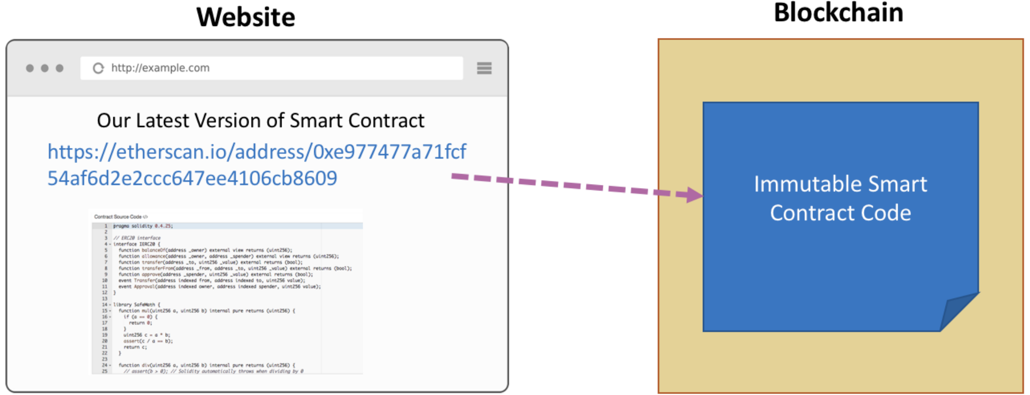


Figure 4.1: State of the art update mechanism for smart contracts.

trust of the Blockchain.

Because of the Blockchain immutability, the rules of Ethereum do not allow changing a published contract's code. Only in one extreme use case it was possible to change the code of an existing published contract. More specifically, the Etherum community decided to change the code of an existing published contract by reversing the history of the Blockchain. This happened with the famous Dao contract hack (50 million dollars at the time of the hack were taken from a buggy smart contract). The Ethereum community followed a hard fork solution, meaning that all miners had to reverse back all transactions. However, this solution is undesirable and it has led to a heated debate between Ethereum users [199] and a split to the Blockchain (Ethereum vs Ethereum classic). Thus, in practice, it is almost impossible to change any code published on Ethereum.

Hence, smart contracts rest on a paradox: They are immutable, but bugs are inevitable. The current state of practice is maintaining out-of-block address on a website to the current free-of-known bugs version (Figure 4.1). This mechanism has a number of problems. The security of the website is an attack vector that has been compromised. There's the potential for a gap between the publishing of the new version on the Blockchain and updating address. After address update, some clients might use the stale, known buggy address because nothing prevents them from using the old address.

In this thesis, we focus on solving and automating this process. More specif-

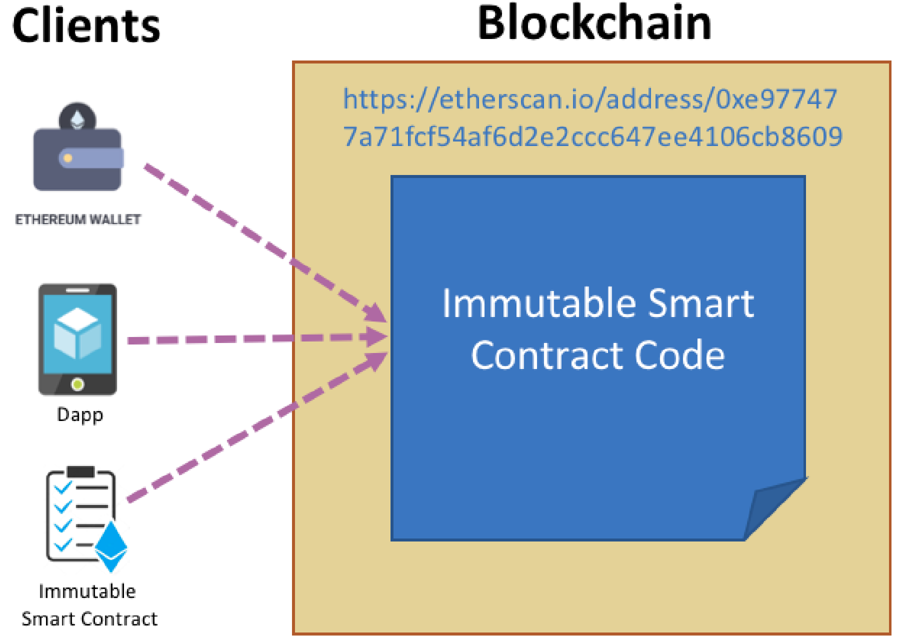


Figure 4.2: Different types of clients that interact with Ethereum Smart Contracts.

ically, we aim to solve the problem of updating smart contracts by providing a framework (PROTEUS) that will allow developers to deploy upgradeable smart contracts, avoiding inconsistencies between the out-of-block address on the website and the real contract on the Blockchain. Also, PROTEUS aims to prevent users from accidentaly calling old buggy contracts, as it will force users to check and provide the latest version of the contract that they want to use.

Types of Ethereum Clients When developers find a bug, they need to replace their smart contract with a newer version (published on a new address). This process is not trivial and error prone because EVM does not provide any built-in upgrading mechanism, and the manual contract replacement has an immediate effect on contract's clients. More specifically, all clients of the contract have to update their applications to use the new contract, which is published in a new address. Depending on how clients interact with it and how their applications are built, we have the following scenarios (see Figure 4.2):

- The user interacts with the contract through his/her wallet account. The interactions that happen in such scenario are usually simple transactions for

sending and receiving Ether tokens. In this case, the user can relatively easily switch to the new contract by updating just the contract's address in the wallet.

- The user has developed an off-chain application that is a client to the contract and interacts with it through function calls and transactions. Depending on where the smart contract is called in the code, the user needs to do the changes accordingly. Also the user needs to update the address to which those calls refer. Because the application is off-chain and written in a more traditional language, such as Javascript, there are no restrictions on changing the code and releasing a new version of the application.

- The user has published a smart contract on the Blockchain that interacts with the old contract through an address that is set either dynamically (through a transaction) or statically in the code. If the address is set dynamically, then the owner of the contract can update it through a function call and the calls are redirected to the new contract. However, if the contract's address is set statically in the code, then this dependent contract should also be considered buggy; the code is immutable and as a result we cannot change it to interact with the new contract. This dependency problem can recursively escalate to other consumer contracts.

Duplicate Smart Contracts When developing a new decentralised application that is backed by a smart contract, developers usually use pre-existing published smart contracts. More specifically, there is a community of Ethereum developers that have implemented and provided some basic templated smart contracts (available on GitHub [200] or the official Ethereum website [201]) and tutorials on best development practices. Those contracts are the ones that developers consider safe to use, as they are usually well tested and peer-reviewed. Because most developers follow this process and because currently the variety of decentrilised applications is limited, a significant number of duplicate contracts are published on Ethereum. However, the current development process is not always successful, and many buggy smart contracts have appeared in the Blockchain, having a significant negative impact

([174, 175]) for their users and the trust of the Blockchain.

Because of the big number of duplicate contracts, when one of them is hacked, it means that other clones are usually vulnerable. In the Multisig wallet bug use case, when the contract was hacked, a race started between 'good' (white-hat group) developers (some of them are from the Ethereum core team) that want to fix the buggy contracts and other potential hackers that want to exploit all the other vulnerable contracts [202, 203, 204]. If the 'good' developers are faster than the hackers, they will also hack the faulty contracts with the intention of moving their money into some safe account. Next, a new version of the faulty contract is provided, and transactions are sent to it based on the previous state of the old contract. This is a process not trivial and not guaranteed to be succesful. Next, all users are notified that they should use the newer version and ignore the old one.

Trust in Ethereum Even though the Blockchain's selling point is that we do not need to trust anyone else except for the network itself, we see that in practice, there is some level of trust. In the example of the multi-sig hack, users trust and expect that the white-hat developers will distribute the money back to the initial users. Other smart contracts, have pre-defined an owner or a group of owners that have more permissions that the contract's users. The contract owners have the ability to disable a contract (contract suicide) and transfer its money to an address that the owner will specify dynamically or that is already specified in the contract [205].

The Ethereum ecosystem is not limited only to trustless applications. For example, an exchange creates individual smart contracts for its users and manages their tokens. In this case, the users trust the exchange, which can perform actions on the smart contracts such as freeze it, transfer the funds, etc. There is also trust that the exchange will return the funds to the user, even if the exchange looses them because of some bug [206]. Other examples include private Ethereum Blockchains where there is some level of trust between the participants and there is a need of fixing potential bugs in smart contracts by providing new versions of it, after an agreement between the parties. Thus, when developing real world applications, there is a need for trusted parties even though the environment is considered fully trustless.

Upgradeable Smart Contracts Even though contracts published in the Blockchain are immutable, it does not mean that the code of a smart contract (if written in a specific format) cannot be updated. The current semantics of EVM allow the developer to provide upgradeability functionalities if they write smart contracts in a specific way, which can be achieved by implementing a versioning system using interfaces. In this chapter, we present PROTEUS, a framework that *automatically* extends the functionality of smart contracts written in Solidity and makes it upgreadable. PROTEUS takes the Solidity code of a smart contract (not published yet) as input and converts it to an upgradeable smart contract. PROTEUS allows users to choose what properties of the smart contract they want to be mutable (function implementations or interfaces) and *automatically* rewrites them. We prove that our changes to the smart contract will not change the properties and the main functionalities of the initial smart contract.

Contributions The contributions of our approach are as follows:

1. We are the first, to best of our knowledge, that addresses the problem of updating Ethereum smart contracts.

2. We improve the state of the art by providing an immutable Trampoline that cannot be hacked

3. The state of art requires ad-hoc error-prone setup. We provide an automatic transformation to contracts.

4. We provide PROTEUS, a source to source rewriting framework that *automatically* makes a smart contract upgradeable.

5. We provide various upgradeability modes based on user preferences.

6. We forbid calls to contracts that are invalidated. We protect users from sending a transaction to disabled addresses and potentially losing their funds.

7. We increase awareness to the Blockchain community about Smart Contract issues that we have found during our research.

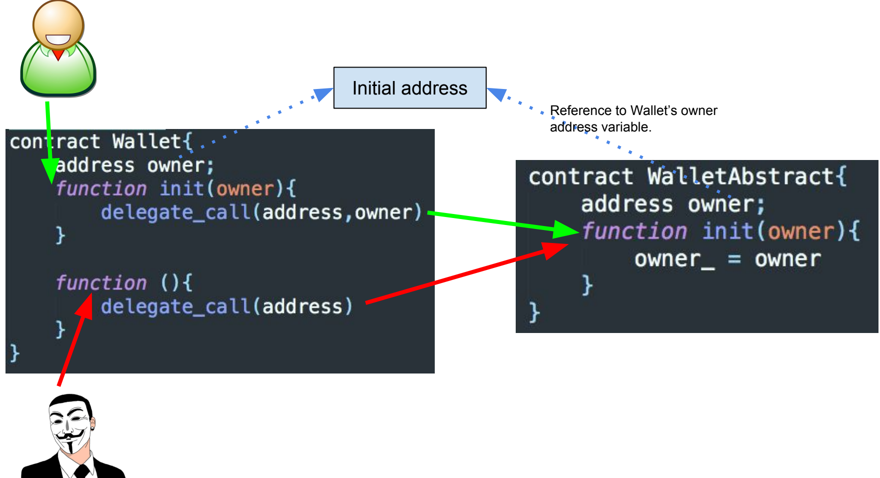


Figure 4.3: MultiSig bug. The contract owner is meant to be defined using the `init` function only by the contract owner. However, the hacker managed to change the owner of the contract by calling the `init(owners)` function of the `AbstractWallet` contract, using the fallout function of the `Wallet` contract.

4.1 Immortal Bugs

Bugs in immutable code are immortal. The lack of upgradeability mechanisms in Ethereum and the careless usage of low-level Solidity functions has led to costly immortal bugs. Next, we describe two such use cases and discuss potential solutions that will prevent such bugs in the future.

MultiSig Contract Hacked. On 19 July 2017, the Parity wallet version 1.5 and above contained a critical vulnerability that enabled the theft of 30 million worth of ETH. The vulnerability discovered in these particular Parity wallets used a multi-signature contract called Wallet.sol (see Figure 4.3). A reasonably straightforward attack allowed the hacker to take ownership of a victim's wallet. The attacker sent two transactions to each of the affected contracts: the first to obtain exclusive ownership of the MultiSig, and the second to move all of its funds. The cause of this bug was the carelessly use of the low-level function **`delegatecall`** in the MultiSig contract. The Ethereum community suggests that developers should use **`delegatecall`** function only if they are familiar and experts in understanding how it works and recommends avoiding it otherwise.

The contract creators of the MultiSig contract used such low levels instructions

to propagate the contract initialisation to another library contract [175, 204], aiming to avoid duplicate contracts published on the Blockchain and also to allow potential future updates on the contract; i.e, to move funds in case of a bug. The contract developers manually coded the desired functionality, relying on peer reviews to validate its functionality. However, none from the peer reviews had spotted the bug in the `delegatecall` that allowed the attacker to call a function that he was not intended to do. As the bug revealed, the process followed to write smart contracts with advanced functionalities (*i.e.,* upgradeability) is ineffective and may lead to costly bugs in the Blockchain.

Ethereum Address Phishing. An initial coin offering (ICO) for a startup project called CoinDash was suddenly stopped when it was discovered that the sale had been compromised shortly after it began [207]. In total, the ICO was able to raise 7.53m before the Ethereum address, used to gather users' funds, was altered to a fake one by an unidentified hacker, resulting in the Ether going to another source; phishing attack. This bug could have been avoided if the users had been forced to double check the contract in which they are sending their Ether or if the organisers had provided a mechanism that would perform such checks.

PROTEUS forces users to interact with a contract only if the user has previously checked the address of the calling contract (user has to know always what to call). This would increase awareness among users about the potential security issues that may arise when they call a smart contract and also provide a mechanism that will prevent it. This mechanism would be also useful to prevent and aware users about many address phishing attempts [208] that are lately observed around Ethereum and other addresses.

4.2 Motivating Example

Listing 4.1 contains a simplified version of a real smart contract published on Ethereum (King of Ether [178]) that allows users to bid on an Auction contract. The logic of this Auction contract is that if a new user sends a better offer than the existing proposals, then it should reimburse the users that are outbid. In Line 7, the

```
1  pragma solidity ^0.4.15;
2  contract CrowdFund {
3   address[] private refundAddr;
4   mapping(address => uint) public refundAmount;
5
6   function refund() public {
7    for(uint i; i < refundAddr.length; i++) {
8     require(refundAddr[i].transfer(refundAmount[refundAddr[i]]));
9    }
10  }
11 }
```

Listing 4.1: A contract that is vulnerable to Denial of Service attack.

`refund()` function refunds all users by iterating over the list of outbid addresses, and applying the transfer function. However, if one transfer transaction fails in the middle of a for loop, all reimbursements fail.

This approach is problematic as a malicious contract can permanently stall this contract by making it fail in a strategic way. In particular, contracts that bulk perform transactions or updates using a for loop can be vulnerable to a denial of service attack, if a call to another contract or transfer fails during the loop. If the call that refunds the frontrunner fails continuously, the auction is stalled, and the malicious user becomes the de facto winner. In this particular use case, the attacker can spam the contract, causing the array to become very large so it runs out of gas and reverts. As a result, this contract may lock users' funds, without the developer being able to fix this immortal bug.

We can fix this vulnerability of the contract by providing a newer version of the `refund()` function that does not exceed a predefined amount of gas (see Listing 4.2). This version is considered safe as it prevents an out of gas exception. Additionally, the `refund()` function stores an iterator in a private variable that will make the while loop exit, when gas drops below a certain threshold. We observe that in the particular fix, only the internal implementation of the `refund()` contract changed; its function definition and the rest of the contract remains the same. However, because the contract is already published and its code is immutable, we cannot change the functionality of the existing contract, but only provide a new contract in a new address and inform clients about this change.

A solution to the code immutability limitations is to use an approach that con-

```
1  pragma solidity ^0.4.15;
2  // Safe against the list length causing out of gas issues
3  contract CrowdFundV1{
4   address[] private refundAddr;
5   mapping(address => uint) public refundAmount;
6   uint256 nextIdx;
7
8   function refund() public {
9    uint256 i = nextIdx;
10   while(i < refundAddr.length && msg.gas > 200000) {
11    refundAddr[i].transfer(refundAmount[i]);    i++;
12   }
13   nextIdx = i;
14  }
15 }
```

Listing 4.2: A version of the contract that is not vulnerable to Denial of Service attack.

tains two contracts. This first one is the `CrowdFund()` contract, which includes the code with the desired functionality. The second contract is a Proxy contract, which we call Trampoline, that forwards calls to the `CrowdFund()` contract. Listing 4.3 shows a simplified Trampoline version of the `CrowdFund()` contract that allows calls from the `refund()` function to be forwarded to a dynamically set address that points to the current version of `CrowdFund()` contract. This mechanism can be considered similar to the Java dynamic class loading approach using reflection, in which code can be loaded and used during code execution.

```
1  pragma solidity ^0.4.4;
2  contract Trampoline {
3   address public c_version;
4   address public owner;
5
6   modifier onlyowner() {
7    require(msg.sender == owner);
8    _;
9   }
10
11  function update(address version) public onlyowner {
12   c_version = version;
13  }
14  function refund() public {
15   require(version.delegatecall(bytes4(keccak256("refund()"))));
16  }
17 }
```

Listing 4.3: A Trampoline contract that forwards calls to the CrowdFund contract.

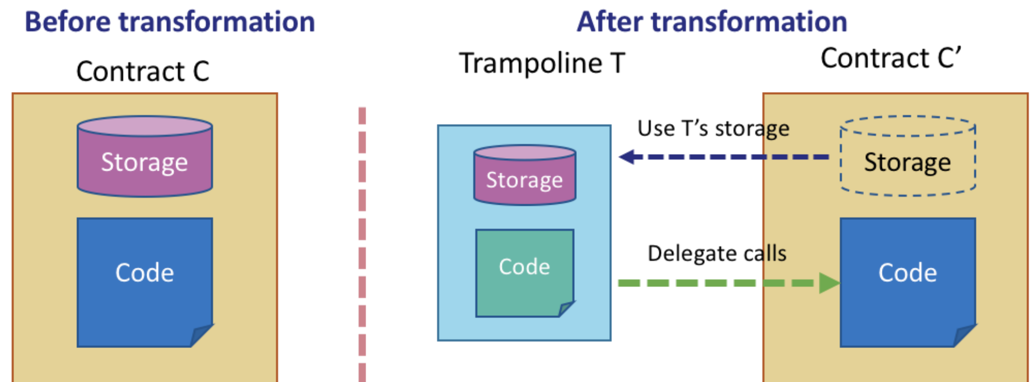


Figure 4.4: Smart contract transformation.

The owner of the contract will publish both the `Trampoline` and the `CrowdFund` contract, but will ask clients to interact only with the `Trampoline`. If a bug suddenly happens, the owner of the contract can update the Trampoline, such that it forwards the calls to the newer version of the contract. In the particular example, the original version and the new version of the contract differ only in a function implementation, and thus the format of the client calls will not change. Thus, the transition is smooth to contract clients. However, as we will explain next, the manual writing of a Trampoline version of a contract is a process laborious and error-prone and the provided version of this example is simple without any security guarantees.

4.3 Approach

To provide upgradeability properties to Ethereum smart contracts, we use the call forwarding properties of the **`delegatecall`** and **`callcode`** commands and additional features of Solidity. More specifically, we build PROTEUS, a source to source transformation tool, written in Java, that takes as input the Solidity code of a smart contract and outputs an upgradeable smart contract. PROTEUS automatically constructs two new contracts: a proxy (Trampoline) contract that delegates calls to other contracts and an upgradeable modified version of the input contract that allows those delegates (see Figure 4.4). Clients interact only with the Trampoline contract, whose code is immutable. However, we can update dynamically the address in which the Trampoline will forward the calls, allowing it to execute the new version of the code.

This mechanism indirectly allows developers to update on the background the code of the contract that the clients use, without breaking functionality.

Trampoline The Trampoline is lightweight as it contains mostly the function definitions of a contract in which it will delegate the calls. We can think of the Trampoline as an Abstract Class of a traditional object-oriented language, and every upgradeable contract as an implementation of this Abstract class. The Trampoline contains the variable declared in the program, storage and states, and it allows the upgradeable contract to access them; similarly to how a child class has access to public or protected variables of a parent abstract class that it implements. Also, the Trampoline contains additional meta-variables that PROTEUS generates and uses to provide the desired call delegate properties.

Contract Owner To provide upgradeability in smart contracts, we need to introduce an entity which will be responsible for deciding when to upgrade the contract. This entity can be a single individual contract owner or a group of owners that vote before updating a smart contract. We assume that the clients of such contracts trust that the owner will not perform malicious acts; *e.g.,* a dodgy owner could update the contract such that it may benefit from it. To increase the trust, the contract owner can use other advanced techniques (*e.g.,* voting smart contracts) provided by the Ethereum, to allow users to vote on a potential upgrade and only then to have the ability to upgrade a smart contract. For simplicity, we will assume that this entity is a single contract owner. Also, our focus is on the code transformations that will allow the contract upgradeability, and we let further advanced solutions for increasing the trust of this approach for future work.

To achieve transparency, the contract owner exposes to clients both the address of the Trampoline and the address of the contract that it forwards the calls to (contract C). However, in the default version, the clients can only interact with the Trampoline contract, and PROTEUS's built-in guards forbid direct calls to C. In particular, contract C accepts calls that are coming from the Trampoline only, avoiding accidental calls that users may make to contract C.

The proposed solution to upgrade smart contracts works because a new version

of the existing published contract can still run within the context of the Trampoline. All the states of the Trampoline and its storage are available and accessible to the new version of the contract. To avoid mistakes and bugs that may appear when manually coding this setup, PROTEUS has a set of rewriting rules to do the code transformations automatically. PROTEUS also has a built-in mechanism to deploy the contracts and update them with newer versions. Note that those transformations happen after the developer has written the contract. Thus, the developer does not need to change the contract's code manually to make it upgradeable.

4.3.1 Delegating Calls to Other Contracts

As we mentioned in Section 2.4.3.5, by delegating calls between contracts, we can execute the code of a contract within the context of the caller's contract. The proposed Trampoline contract uses the `delegatecall` and `callcode` instructions to forward calls between contracts. However, those instructions are considered advanced, and not careful usage may lead to undesired contract behaviour. The essential features of those instructions that can easily confuse the developer are 1) signature function construction 2) default function visibility, 3) uncertainty about the original message sender and 4) variable declaration order mismatch.

Function Signature Construction The `delegatecall` function gets as input a string that represents the function definition and a list of its parameters. When constructing the string manually, the developer can easily introduce bugs as there is no in-place mechanism to check if the string is correctly representing the function that the call should be forwarded; Solidity compiler does not support such functionality. Only during the contract execution such bug may be revealed, but sometimes that may not even be obvious.

We demonstrate a simple example of code that contains a bug due to wrong function signature construction. In Listing 4.4, Line 4, the code delegates the calls to the `setN(uint)` function of the Callee contract. We can observe that function definition `setN(uint _n)` in the Calle contract and the string parameter `"setN(int256)"` differ on the type of the numerical type used. The function definition inside the string uses a signed integer, but the Calle's real function is using

an unsigned integer. This difference can introduce arithmetic overflow bugs. The Solidity compiler doesn't identify this type mismatch, and the user may publish this buggy contract in the Blockchain. The call to the `delegatecallSetN()` will be forwarded to the Callee contract, but because the function signature does not match correctly any of the functions , the `Callee fallout()` function will be executed instead of the `setN(uint _n)`.

```
1  contract Caller {
2   uint public n;
3   function delegatecallSetN(address _c, uint _n) public {
4    require(_c.delegatecall(bytes4(keccak256("setN(int256)")),_n));
5   }
6  }
7  contract Callee {
8   uint public n;
9   function setN(uint _n) public {
10   n = _n;
11  }
12 }
```

Listing 4.4: A Trampoline contract that forwards calls to the CrowdFund contract.

Default Function Visibility By default, if a function or variable does not contain the private or public statement is public. This is different from traditional programming languages that have default private-visibility for their functions. This property of the Solidity language can confuse developers, as the multi-sig wallet showed [175]; the developers of the contract accidentally allowed a function that should have been private to be called by anyone. The probability of introducing bugs increases if functions are used in a more advanced context. We consider that delegate calls is an advanced feature of the Solidity language. We believe that Solidity should adopt a default-private level of visibility for contract functions. This change would have likely prevented the multi-sig exploit and others like it.

Sender's address propagation When a contract is called, there is a special variable `msg.sender` that stores the address of the user's or contract's account that sent the transaction. The usage of the `delegatecall` to forward the call to a new contract, propagates the initial sender to the new contract as well. That means that

the final contract will not know if the call was direct from the user. On the contrary, the usage of the **callcode** does not propage the original message sender but the address of the Trampoline contract. Thus, additional checks should be added to the contract to understand and get the original message sender.

Variable Declaration Order Mismatch The **delegatecall** instruction was introduced as a low-level instruction to implement libraries in Solidity. The main property of a library is that it does not have any storage, but it can directly access the storage of the contract that uses it. Similarly, in our case, the direct usage of the **delegatecall** function allows the upgradeable contract to access the storage of the Trampoline contract. In particular, the variables of the upgradeable contract are references to the variables defined in the Trampoline contract.

Variables are matched using an offsetting approach; the first integer in the Trampoline contract correlates to the first integer in the upgradeable contract. This offsetting mechanism is different from traditional programming languages where the variables match based on their names. This difference can confuse the developer who can easily introduce bugs without realising it.

```
1  contract GiveawayLib {
2   uint public threshold;
3   uint public count;
4   uint public amount;
5   address owner;
6   mapping(address=>bool) public claimed;
7
8   constructor(uint _amount, uint _threshold) public {
9    owner = msg.sender;
10   amount = _amount;
11   threshold = _threshold;
12  }
13
14  function addClaim() public {
15   if (!claimed[msg.sender] && count < threshold) {
16    claimed[msg.sender] = true;
17    msg.sender.transfer(amount * 1 ether);
18    count++;
19   }
20  }
21 }
```

```
22
23 contract Giveaway {
24  address  owner;
25  uint public count, amount, threshold;
26  uint public amount;
27  uint public threshold;
28  mapping(address=>bool) public claimed;
29  address libAddress;
30
31  constructor(address _libAddress) public payable {
32   owner = msg.sender;
33   libAddress= _libAddress;
34   count = 0;
35   threshold = 3;
36   amount = 5;
37  }
38
39  function claim() public payable{
40   require(libAddress.delegatecall(
41    bytes4(keccak256("addClaim()"))));
42  }
43  function() public {
44   require(libAddress.delegatecall(msg.data));
45  }
46 }
```

Listing 4.5: Sample code that shows how calls are forwarded between different contracts in Solidity.

We give an example of how easily a variable declaration order mismatch can happen, by using a contract and a library in Listing 4.5. The logic of the contract is that an individual can give an amount of Ether as a reward to the first users that claim it. The `count` variable in Line 3 keeps track the total number of claims and the `claimed` variable in Line 6 limits the number of claims per user. The contract uses the `claim()` function in Line 39 and fallback function in Line 44 to interact with the library.

This example is buggy because the ordering of the variables between the contract and library is different. The delegate calls to `GiveawayLib` library reference different variables from what they are supposed to. In particular, the `addClaim()` function uses the `threshold` variable, which is stored at position 0 in `GiveawayLib`.

However, in the `Giveaway` contract, the `address owner` variable is stored at position 0. This causes the delegate call to `addClaim()` to reference the `address owner`. Thus, the actual threshold used for the maximum number of claims is equal to the value of the owner address which is a 32-bit unsigned integer. As a result, the contract allows unintentionally more users to claim the giveaway; the `Giveaway` contract may lose all of its ether in one claim. Also, note that similar vulnerability can happen because of mismatch on the variable size. In the particular example, the variables are 32 bits and (i.e. storage slot 0 has a 32-bit int in the library and 2 16-bit variables in the contract)

Another vulnerability from using `delegatecall()` is when the callee contract has public or external functions that handle sensitive data. An attack could call that function through a specially modified `msg.data`. This can cause unwanted results such as changing the owner of the contract. To ensure that this does not happen, all functions in the contract should have the appropriate visibility keywords. PROTEUS protects against such bugs as it automatically generates the order of the declared variables as well as it prevents the code transformation if the visibility keywords are not used in every function of the contract.

4.3.2 Solidity Syntax

In Listing 4.6 we provide the subset of the Solidity language that PROTEUS uses. A program P is constructed by a sequence of contract declarations. A contract C consists of a constructor, a sequence of function definitions ($\forall_f \in C$), function modifier definitions ($\forall_m \in C$), and variable declarations ($\forall_v \in C$). Each contract has a unique identifier (*id*). A contract can have at most one constructor (*constructorDef*). If the constructor is not defined, then a default empty constructor is used when the contract is deployed. Function modifiers (*modifierDef*) are compile-time source code roll-ups. Function modifiers are typically used in smart contracts to make sure that certain conditions are met before proceeding to execute the rest of the body of code in the method. From the mutability keywords that Solidity provides, we focus on the *view* and *payable* keywords. The *view* keyword is used to forbid any state changes to a function and the *payable* keyword is used to allow a function to accept

Ether.

Listing 4.6 shows the subset of the Solidity language that we use in the rewriting rules of PROTEUS.

```
P ::= C*

C ::= contract id { B* } ;

B ::= stateVarDeclr | constructorDef | modifierDef | funcDef

stateVarDeclr ::= type (public | private)* id ( = expr)? ;

constructorDef ::= constructor paramList modifierList block

modifierDef ::= modifier id paramList? block

funcDef ::= function id? paramList modifierList returnParameters? ( ;
    | block )

paramList ::= ( ( parameter (, parameter)* )? )

modifierList ::= ( id ( () exprList? ) )?
| view | payable | public |  private)*

exprList ::= expr (, expr)*

parameter ::= type id?

id ::= [a-zA-Z$_] [a-zA-Z0-9$_]*

type ::= address | bool | string | int | uint | byte

expr ::= Standard, see the Solidity documentation for details [209].

block ::= Standard, see the Solidity documentation for details [209].
```

Listing 4.6: Syntax of a subset of Solidity.

4.3.3 Proteus Rewriting Rules

PROTEUS provides various versions of the contract based on the upgradeability preferences of the user; developers can explicitly specify which parts of a contract should be upgradeable. The different versions that PROTEUS can provide are based on the table preferences as shown in Table 4.1. More specifically, PROTEUS provides developers with two mutability options: a) Mutable Implementation and b) Mutable

Table 4.1: Table with mutability modes of PROTEUS. Mutable Implementation

Type	Mutable	Force Address Check
Implementation	1	0
Interfaces	0	0

Interface. Also, PROTEUS provides the option (Force Address Chec) which forces the users to provide the address of the contract in which the `Trampoline` delegetes the call, before calling it. This is necessary to avoid accidental calls to the old version of the contract and to notify users about the update that happened. In the first mode, the user can change the internal implementation of functions in a contract. In the second mode, PROTEUS can add or delete functions in the initial contract.

The proposed rewriting rules should provide the following properties:

- Functions of the contract can be updated in the newer versions.
- Functions can be added or deleted in the newer versions.
- An owner should be introduced that is allowed to upgrade a contract. We present a single owner in this paper, but multiple owners with a multi-signature can be used as well.
- All the calls to the new contract should be through the Trampoline. We want to forbid direct calls to the versionable contract as that may confuse the users and lead to money loss.
- Prove that the changes do not affect the functionality of the original contract.

4.3.4 Mutable Implementation Mode

The first mode that PROTEUS supports is the change of function implementations of a contract (Table 4.1). In this mode, we consider that the signature of each function does not change, but the internal implementation can be modified. In particular, given an initial smart contract `C`, PROTEUS *automatically* generates a `Trampoline` contract `T` and an upgradeable version `C'` of the initial contract .

Figure 4.5 shows a visual representation of how the `Trampoline` contract forwards calls to upgradeable contracts. Initially, the developer deploys both the first

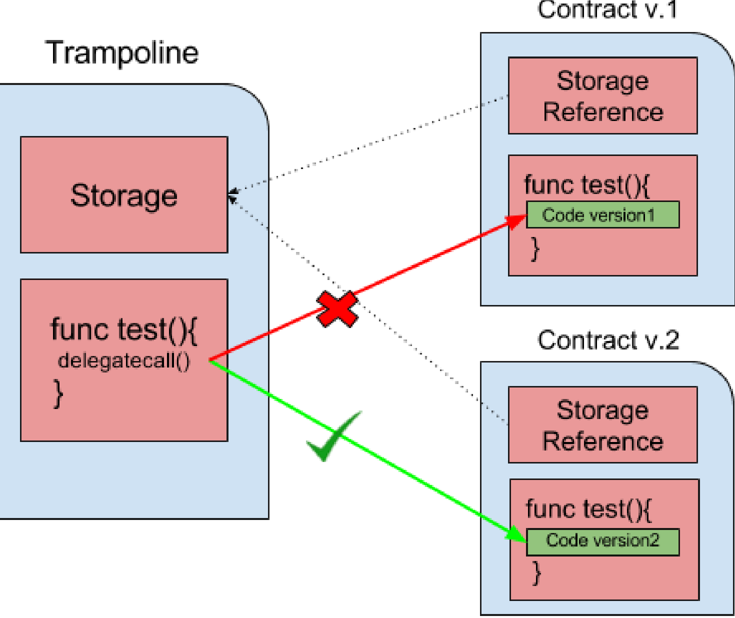


Figure 4.5: Trampoline and Contract interaction for the mutable implementation mode. After the `Trampoline` is deployed, its code is immutable. Each call to a function of the `Trampoline` is delegated to one of the upgradeable contracts. The user can deploy a new contract that will have the same function signatures, but the internal code can be modified.

version of the contract (`Contract v.1`) and the Trampoline on the Blockchain. Next, the developer calls the `update(address)` function of the published Trampoline and gives as input the address of the `Contract v.1`. The address of the Trampoline contract is then published to the users who can start interacting with it. To deploy a newer version of the contract, the developer firstly uses PROTEUS to get its upgradeable version. Then, the developer publishes the new contract with the changes (`Contract v.2`) on the Blockchain and sends a transaction to the `update(address)` function of the Trampoline with the new address of the upgradeable contract. All the calls to the `Trampoline` are now forwarded to the new version of the contract.

4.3.4.1 Trampoline Rewriting Rules

In this section, we introduce the rewriting rules that PROTEUS uses to generate automatically the `Trampoline` contract.

Delegating Calls to Functions with Returns Solidity cannot return data from

functions when a delegate call is used. The `delegatecall()` and `callcode()` functions return a boolean value, which indicates whether the invoked function terminated (true) or caused an EVM exception (false). To access the actual data returned from a function, we need to know the encoding and the size of the return data in advance. Then, low-level inline assembly code should be used. Using inline assembly code is error prone, it discards several important safety features of Solidity and developers are advised to avoid using it [210]. Our transformation rules, do not use low assembly code, to avoid its limitations.

To handle this limitation, we introduce a preprocessing step (rewriting rule) to generate a list of return meta-variables for each function that has a return statement, under the constrain that only primitive types (`uint`, `boolean`, `address`) are supported. More specifically, when a call is delegated to a funtion that returns, we store the data that the function returns to the corresponding meta-variable; we inject code inside the return functions. The name of each variable is constructed from the name of the contract (`C`), name of the function (`f`) and the data return type (`T`). We construct the variable name following this approach to reassure that it is unique and it corresponds to the data returned for this specific function.

Formally, let R map contracts to that contract's list of their return meta variables. Then, $\forall_f \in C$,

```
function f(p⃗) returns (T var) →   T CfT;
```

Trampolinify Contract (Rule 4.2) The Trampolinify Contract rule generates the basic code structure of the `Trampoline` contract from the input contract `C`. In particular, this rule introduces additional variables and helper functions necessary to support the function dispatches. This rule also injects the generated list of variables from the previous rewriting rule (Rule 4.2, Line 6). We use the term "meta-variables" for those injected variables, to distinguish them from the existing variables of the input contract. PROTEUS inserts the meta-variables and the helper functions before any existing contract variable and function definition, to prevent variable mismatch bugs. The Trampolinify Contract rule introduces 4 meta-variables, one function and one function modifier as follows:

```
1  contract C {
2   B
3  }
```

⇓ **Trampolinify Contract**

```
1  contract Tᶜ {
2   address version, owner;
3   address trampoline, msg_sender;
4
5   // meta-variables for returns
6   T_r C_r f_r T_r, ∀ r∈R(c)
7
8   modifier check_owner(){
9    require(msg.sender == owner);
10   _;
11  }
12
13  function update(address v) check_owner {
14   version = v;
15  }
16   B
17 }
```

Rule 4.2: Trampolinify Contract Rule. PROTEUS injects a set of meta variables and helper functions in the body of the Trampoline contract.

- The `owner` meta-variable stores the address of the contract owner.
- The `version` meta-variable stores the address of the current version of the upgradeable contract.
- The `trampoline` meta-variable stores the address of the Trampoline. This meta-variable is used in the upgradeable contract's code to check that the call is coming from the Trampoline contract and to forbid any other external calls (see Section 2.4.3.7).
- The meta-variable `msg_sender` is used to store the original sender (`msg.sender`) of the transaction, because the `callcode` does not forward the original sender. By introducing this meta-variable, we store both the address of the original sender as well as provide the address of the Trampoline to the upgradeable contract.

```
1    constructor(p⃗){
2      B
3    }
4    '|'
5    function C(p⃗){
6      B
7    }
```

⇓**Constructor Injection**

```
1    bool allowExecution = true;
2
3    constructor(p⃗) {
4     trampoline = address(this);
5     owner = msg.sender;
6     msg_sender = msg.sender;
7     InjectedConstructor(p⃗);
8    }
9
10   function InjectedConstructor(p⃗) private {
11    require(allowExecution == true);
12    allowExecution = false;
13    bytes4 sig =
        bytes4(keccak256("InjectedConstructor(p⃗)"));
14    require (!version.callcode(sig,p⃗));
15   }
```

Rule 4.3: Constructor Injection rule. Initialise meta variables when contract is created.

- The `update(address v)` function is used by the contract owner/s to update the contract that the trampoline delegates the calls to.
- The `check_owner(address v)` modifier is used as a guard for functions that only the contract `owner` is allowed to call.

Constructor Injection (Rule 4.3) The purpose of the `Trampoline` contract is to forward all calls to contract `C'` and execute its code. When the Trampoline is first deployed, it needs to instantiate the meta-variables injected and execute the code of the constructor. Because the constructor's code may call other private functions of the input contract, which are not available to the `Trampoline`, it is necessary to delegate the constructor's functionality to contract `C'`.

The Constructor Injection rule introduces changes to the constructor such

```
1   function f(p⃗) public '|' external
2   {
3     B
4   }
```

⇓Upgradeable Functions

```
1   function f(p⃗) public '|' external {
2    msg_sender = msg.sender;
3    bytes4 sig = bytes4(keccak256("f(p⃗)"));
4    require (! version.callcode(sig,p⃗));
5   }
```

Rule 4.4: Upgradeable Functions rule. **`public`** and **`external`** functions are forwarding the call to the upgradeable contracts.

that it provides the desired upgradeability properties. First, it injects code that instantiates the meta-variables defined by the Trampolinify Rule (Rule 4.3, Lines 2-4). The owner of the contract and the `msg_sender` are set, as well as the address of the `Trampoline`. Next, it introduces a new function (`InjectedConstructor()`) that contains the existing code of the constructor. This code is forwarded to the upgradeable contract and executed through the **`callcode`**`()` function inside contract `C`. The functionality of the `InjectedConstructor()` function is injected to contract `C'`; as we show in Rule 4.9. The `InjectedConstructor()` function is introduced as a second constructor of contract `C'` which will be instantiated automatically only when the Trampoline is deployed on the Blockchain.

The visibility of the `InjectedConstructor()` is **`private`** (Rule 4.3, Line 11) and it can be executed only once, during the Trampoline deployment (Rule 4.3, Line 12). In cases that the developer does not provide the constructor of a contract, we have added a post-processing step that generates the default constructor with the meta-variables instantiation.

Upgradeable Functions (Rule 4.4) The Upgreadable Functions rule transforms all the functions of `C` that can be upgradeable. PROTEUS applies this rule to all functions that are **`public`** or **`external`**. PROTEUS first assings the **`msg.sender`** to the metavariable `msg_sender`; this is necessary to propagate the original message sender to `C'`, when using the **`callcode`**`()` function. Next, it constructs the parameters of

```
1   function f(p⃗) public | external returns (T) {
2     B
3   }
```

⇓Upgradeable Functions with Returns (UF)

```
1   function f(p⃗) public returns (T) {
2     msg_sender = msg.sender;
3     bytes4 sig = bytes4(keccak256("fDelegate(p⃗)"));
4     require (! version.callcode(sig,p⃗));
5     return CfT;
6   }
```

Rule 4.5: Upgradeable Functions with Returns. `public` and `external` functions are forwarding the call to the upgradeable contracts. The return value is assigned to the corresponding meta-variable.

the `callcode()` by extracting the function name and its parameters. Each funciton signature is the first 4 bytes of the `keccak256()` hash function. In the Line 4, the rule checks that the function was successfully executed. If the execution fails, then the the transaction reverts and all changes that have happened in the contract are discarded. If a function contains the `return` statement, then the function returns the corresponding meta-variable (Rule 4.4, Line 5).

Upgradeable Functions with Returns (Rule 4.5) This rule is similar to Rule 4.4, with the difference that it handles functions with returns and it delegates calls to a different function of `C'`. More specifically, this rule forwards calls to a modified function of `f()` (`fDelegate()`, Rule 4.5, Line 3). The `fDelegate()` is an intermediary (helper) function for each `public` or `external` function `f()` of `C` that returns a value and assings it to the corresponding meta-variable. As we show, in Rule 4.9, this trick allows us to execute the code of a function with return succesfully.

Unexposed Functions The Trampoline forwards only functions that are `public` or `external`. Thus, all other functions and function modifiers should be excluded from the code of the Trampoline. The Unexposed Functions rule in Table 4.6 excludes from the Trampoline all private and internal functions.

4.3.4.2 Rewriting Rules for Contract `C'`

Next, we describe the rules to generate the modified upgradeable contract `C'`.

```
1  function f(p⃗) private
2  '|' internal '['returns (T)']' {
3   B
4  }
5  '|'
6   modifier f(p⃗) private
7   '|' internal {
8   B
9  }
```

⇓**Unexposed Functions**

```
1  ε
```

Rule 4.6: Unexposed Functions rule (F1). **`internal`** and **`private`** functions are not exposed through the Trampoline. Those functions can be updated with the new version of contract C.

```
1  contract C {
2   B
3  }
```

⇓**Meta-variable Declaration**

```
1  contract C' {
2   address version, owner;
3   address trampoline, msg_sender;
4   // meta-variables for returns
5   T_r C_r f_r T_r, ∀_{r∈R(c)}
6  }
```

Table 4.7: The same set of of meta-variables that were added in the `Trampoline` are also added in the code of contract `C'`.

Meta-variable Declaration (Rule 4.7) The Meta-variable Declaration rule introduces the meta-variables necessary to support the delegating of calls to contract `C'`. The meta-variables of `C'` are references to those of the `Trampoline`. Because EVM matches them using offsetting, the variables should be declared in the same order in `Trampoline` and `C'`. Thus, this rule ensures that the meta-variables declared in the `C'` contract have the same order with the meta-variables declared in the `Trampoline`.

Trampoline Only Calls (Rule 4.8) The Trampoline Only Calls rule is used to allow only the calls in contract `C'` that are coming from the `Trampoline` contract. PROTEUS adds this restriction to forbid unecessary or accidental calls directly to

```
1  function f(p⃗) public '|' external {
2   B
3  }
```

⇓Trampoline Only Calls

```
1  function f(p⃗) public '|' external  {
2   require(msg.sender == trampoline_address);
3   B
4  }
```

Table 4.8: PROTEUS allows calls to the contract C' only from the Trampoline address. This rule forbids any other calls.

```
1  function f(p⃗) public '|' external returns(T val '|' T){
2   B
3  }
```

⇓Functions with Returns

```
1  function f(p⃗) private returns(T val '|' T) {
2   B
3  }
4  function fDelegate(p⃗) public '|' external returns(T val
      '|' T) {
5   require(msg.sender == trampoline);
6   // assign to meta-variable of the function
7   CfT = f(p⃗);
8  }
```

Table 4.9: PROTEUS allows calls to the contract `C'` only from the `Trampoline` address. This rule forbids any other calls.

contract `C'` and also to prevent users from using an older version of contract `C'`. Users should automatically call the latest version of the contract, through the `Trampoline`.

Functions with Returns (Rule 4.9) To properly achieve delegates on function with returns, PROTEUS uses this rule to split an input function `f()` to two functions (`f()` and `fDelegate()`). The code of the new `f()` function is similar to the input `f()` function, but the visibility is different. The new `f()` function should be called externally only through the `fDelegate()` function and thus the visibility is changed to **`private`**. The `fDelegate()` checks that the call is from the `Trampoline` contract and executes `f()`. Then, it assigns the result to the corresponding meta-variable of function `f()`. PROTEUS uses this mechanism because it allows it to get the final

```
1  contract C {
2   mapping(address => uint256) balances;
3
4   constructor() public {
5    balances[msg.sender] = 200;
6   }
7   function setBalance(address customer, uint256 _n) public {
8    balances[customer] = _n;
9   }
10  function getBalance(address customer) public view returns
        (uint256) {
11   return balances[customer];
12  }
13 }
```

Listing 4.7: Sample contract given as input to PROTEUS.

return value from `f()`, without the need of making complicated rewriting inside its code and avoiding further logic that handles internal `if` statements and early returns of the function.

4.3.4.3 Mutable Implementation Code Transformation Example

Next, we show a full example of how PROTEUS applies the above mentioned rewriting rules to a sample contract. Listing 4.7 contains a simple contract *C*, which allow the contract owner to store user balances in the Blockchain. The contract *C* contains a constructor, in which it initialises default variables. It also contains two functions, one that allows the contract owner to add a new balance for a user (`setBalance(address,amount)`), and one that returns the balance of a user (`getBalance(address)`).

Listing 4.8 contains the code of the `Trampoline`, with which the users interact. First, we note that the necessary meta-variables are added. Next, because the `setBalance(address,uint256)` function is `public`, PROTEUS decides to make it upgradeable and thus all calls are forwarded to contract *C'*. We also note that PROTEUS generated a meta-variable that stores the result of the `getBalance(address)` function and returns it to the user (Listing 4.8, Line 38). Listing 4.9 contains the generated upgradeable *C'* contract with the necessary changes as described in the above rewriting rules.

We have to note, that PROTEUS automatically generates those contracts and further deploy them on the Blockchain, without the developer spending effort in

understanding the generated code or manually changing it. The `Trampoline`'s code is immutable and will not be generated again for the newer version of the contract. To provide a newer version of the contract, the developer should provide a modified contract C and then PROTEUS will transform it to upgradeable contract.

```
1  contract TRAMPOLINE {
2   // #### META-VARIABLES START ####
3   address public C_VERSION_, C_OWNER_;
4   address public TRAMPOLINE_ADDRESS_, MSG_SENDER_;
5   uint256 getBalance_var0;
6   modifier CHECK_CONTRACT_OWNER() {
7    require ( msg.sender == C_OWNER_);
8    _;
9   }
10  function UPDATE(address version ) public CHECK_CONTRACT_OWNER
       {
11   C_VERSION_ = version;
12  }
13  // #### META-VARIABLES END ####
14  mapping(address => uint256) balances;
15
16  constructor public {
17   TRAMPOLINE_ADDRESS_ = address(this);
18   C_OWNER_ = msg.sender;
19   constructorInjector();
20  }
21  function constructor_injector(){
22   require (C_VERSION_.callcode(bytes4(
23   keccak256("constructorInjector()"))));
24  }
25  function setBalance(address customer, uint256 _n) public {
26   MSG_SENDER_ = msg.sender;
27   require (C_VERSION_.callcode(bytes4(
28   keccak256("setBalance(address,uint256)")),customer,_n));
29  }
30  function getBalance(address customer) public  returns
       (uint256) {
31   MSG_SENDER_ = msg.sender;
32   require (C_VERSION_.callcode(bytes4(
33   keccak256("getBalanceDelegate(address)")),customer) );
34   return getBalance_var0 ;
35  }
```

```
36 }
```

Listing 4.8: Trampoline generated by PROTEUS.

```
1  contract C' {
2   // #### META-VARIABLES START ####
3   address public C_VERSION_, C_OWNER_;
4   address public TRAMPOLINE_ADDRESS_, MSG_SENDER_;
5   uint256 getBalance_var0;
6   // #### META-VARIABLES END ####
7
8   mapping(address => uint256) balances;
9   function C() public {}
10
11  function constructorInjector(){
12   MSG_SENDER_ = msg.sender;
13   balances[msg.sender] = 200;
14  }
15
16  function setBalance(address customer, uint256 _n) public {
17   require(msg.sender == TRAMPOLINE_ADDRESS_);
18   balances[customer] = _n;
19  }
20  function getBalance(address customer) private view returns
       (uint256) {
21   require(msg.sender == TRAMPOLINE_ADDRESS_);
22   return balances[customer];
23  }
24
25  function getBalanceDelegate(address customer) public  returns
       (uint256)
26  {
27   require(msg.sender == TRAMPOLINE_ADDRESS_);
28   getBalance_var0 = getBalance(customer );
29  }
30 }
```

Listing 4.9: Trampoline generated by PROTEUS.

4.3.5 Mutable Interface Mode

The next functionality that PROTEUS supports is the mutable Interface mode (Table 4.10). With this mode, the developer can deploy a contract on the Blockchain and still change its existing functions. In particular, PROTEUS supports the addition

Table 4.10: Table with mutability modes of PROTEUS. Mutable Interface.

Type	Mutable	Force Address Check
Implementation	0	0
Interfaces	1	0

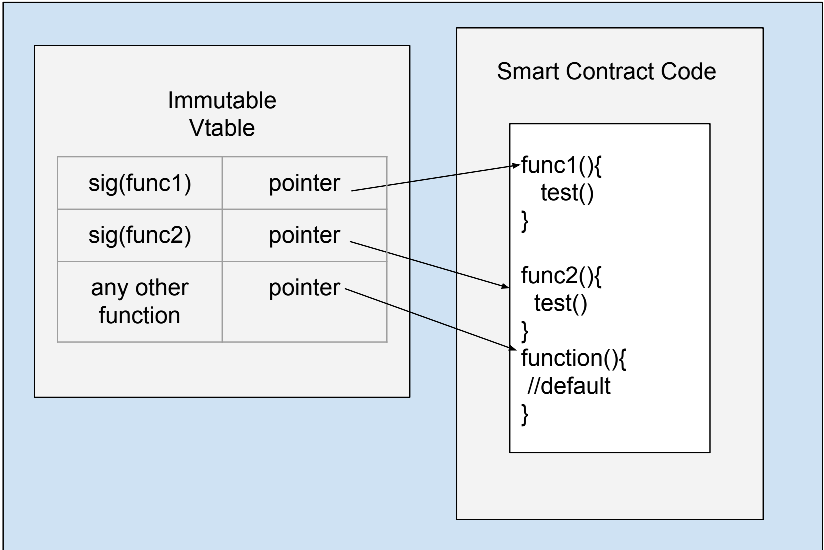


Figure 4.6: EVM uses an intenal immutable virtual function table as a lookup table of functions for resolving function calls in a dynamic/late binding manner.

and deletion of new functions in the contract, and the update of existing function signatures (by extending or removing parameters from the parameter list). To provide the desired functionalities, we introduce in the `Trampoline` a Virtual Function Table. We also use the properties of the anonymous fallback function to delegate calls. This mode is an extension of the Implementation mode as it contains most of the rewriting rules that we previously mentioned combined with some new rules, as we describe next.

Virtual Function Table Each smart contract internally has a virtual function table (Figure 4.6) that is used to support dynamic dispatches. The virtual table is used at runtime to invoke the appropriate function implementations that match the signature of the method included in the transaction data (`msg.data`). This internal virtual table is a hash table that stores as key the function signature and as a value a reference to the executable function code. The function signature is calculated by hashing the method signature using the `keccak256()` hash function and then keeping the

4 first bytes of the hash result. This internal virtual table is initialised once during the contract creation and is immutable. This functionality is implemented inside the EVM, it is not easy for the developer to have a clear understanding of how this mechanism works. This lack of understanding is one of the reasons that developers have introduced bugs when trying to do function dispatches in their contracts.

Fallback Function When a transaction is sent to the contract without transaction data or if the signature of the method call is not in the internal contract's virtual function table, the fallback function (`function()`) is executed instead. Each smart contract comes with a default fallback function, and users need to define it if they want to extend its functionality. If the fallback function is not defined explicitly in the code, an exception is thrown. If the contract is meant to receive Ether with simple transfers, the fallback function should contain the `payable` instruction in its definition. Even though the fallback function does not take parameters, it is possible to access the parameters contained in the transaction from the special global variable `msg.data`.

We combine the properties of the fallback function with delegate calls to provide the mutable Interface mode of PROTEUS. More specifically, we introduce in the Trampoline a custom dynamic virtual function table that stores all function signatures of the contract and a boolean value which indicates if those functions are allowed to be used or not. The `vtable` is a standard mutable hash table data structure that can be changed through transactions. Practically, by using a `vtable` in the Trampoline, we expose to the developer the internal mechanism that EVM uses for function dispatches (Figure 4.7). Similarly to the mutable Implementation mode, only the owner of the contract can interact with the `vtable`, and decide to add, remove or change the any of the functions of the `Trampoline`.

The process of changing a method interface and delegating calls in other contracts happens as follows. When a transaction is sent to the Trampoline, the EVM checks if the internal virtual function table contains the function signature. Because the Trampoline has no implementation of any functions, the method call will not match with any signature in the internal virtual table. As a result, the fallback

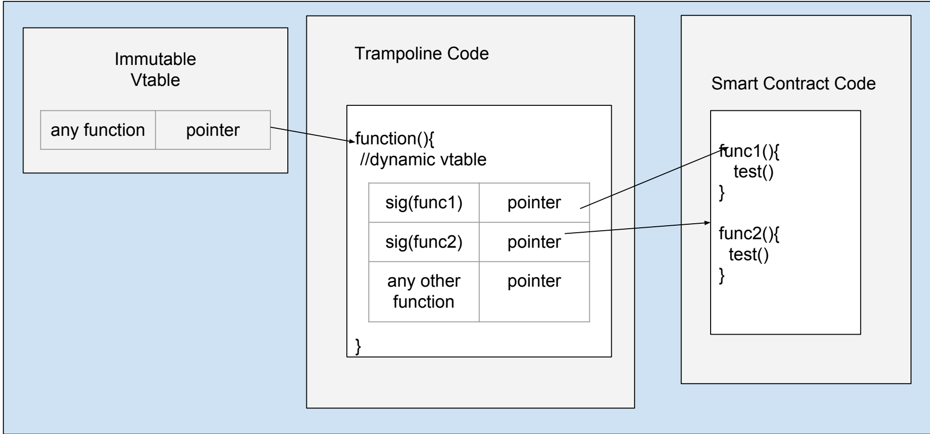


Figure 4.7: Dynamic virtual function mechanism used by the `Trampoline`. The logic of how EVM dispatches functions internally is exposed to the developer through the code contained in the `Trampoline`.

function will be triggered. Inside the fallback function, the method signature is looked in the custom `vtable`, which has been previously populated by PROTEUS with the default function signatures that are allowed to be executed. If the function signature is in the `vtable`, the call is delegated to the `C'` contract; otherwise an exception is thrown. When the contract is delegated in the `C'` contract, the internal virtual function table of the `C'` contract will be used to match the correct method for execution.

4.3.5.1 Rewriting Rules for Mutable Interface

In this section, we present the rewriting rules that PROTEUS applies to generate the `Trampoline`. In total, there are 4 new rewriting rules for the `Trampoline`. For generating contract `C'` the rules are similar to the rewriting rules used in the Implementation mode. Thus, we do not repeat them in this section.

Default Function Signature Generation PROTEUS needs to add the existing functions of the input contract `C` to the dynamic vtable of the `Trampoline`. Thus, it uses the first rule to scan all existing **`public`** and **`external`** functions of contract `C` and to generate the corresponding *insert* statements to the `vtable`.

Formally, let F be all the **`public`** and **`external`** functions of `C`. Then, $\forall_f \in F$,

$$\textbf{function}\ f(\vec{p}) \rightarrow \text{vtable}[\textbf{bytes4}(\textbf{keccak256}(\text{"f"},\vec{p}))] = \textbf{true};$$

```
1  contract C {
2   B
3  }
```

⇓Dynamic Function Dispatch

```
1  contract Tc {
2   address version, owner;
3   address trampoline, msg_sender;
4
5   // add virtual function table for function dispatch
6   mapping (uint256 => bool) vtable;
7
8   // meta-variables for returns
9   Tr Cr fr Tr, ∀r∈R(c)
10
11  modifier check_owner(){
12   require(msg.sender == owner);
13   _;
14  }
15
16  function update(address v) check_owner {
17   version = v;
18  }
19
20  function add_new_function(string fun_def) only_owner{
21   bytes4 sign = bytes4(keccak256(fun_def));
22   vtable[sign] = true;
23  }
24
25  function delete_function(string fun_def) only_owner{
26   bytes4 sign = bytes4(keccak256(fun_def));
27   vtable[sign] = false;
28  }
29 }
```

Rule 4.11: Dynamic Function Dispatch Rule. PROTEUS injects a set of meta variables and helper functions in the body of the Trampoline contract.

Next we describe the rewriting rules used by PROTEUS for the mutable interface mode. In total there are 3 rewriting rules used for this mode, as we explain next.

Dynamic Function Dispatch (Rule 4.11) With this rule, PROTEUS constructs the basic structure of the `Trampoline` contract. Similar to the Implementation mode, PROTEUS first adds the necessary meta-variables and functions to correctly delegate calls. Additionaly, this rule introduces the `vtable` lookup ta-

```
1  constructor(p⃗){
2    B
3  }
4  '|'
5  function C(p⃗){
6    B
7  }
```

⇓**Constructor Expansion**

```
1  bool allowExecution = true;
2
3  constructor(p⃗) {
4   trampoline = address(this);
5   owner = msg.sender;
6   msg_sender = msg.sender;
7
8   // generate a list of insert statements
9   // with the default functions of the contract
10  vtable[bytes4(keccak256(f,p⃗))] = true, ∀ f(p⃗)∈F(c)
11
12  InjectedConstructor(p⃗);
13 }
14
15 function InjectedConstructor(p⃗) private {
16  require(allowExecution == true);
17  allowExecution = false;
18  bytes4 sig = bytes4(keccak256("InjectedConstructor(p⃗)"));
19  require (!version.callcode(sig,p⃗));
20 }
```

Rule 4.12: Constructor Expansion rule. The `vtable` is populated with the default function definitions of contract `C`.

ble (Rule 4.11, Line 5) and two functions (`add_new_function(string)` and `delete_function(string)`) that allow the contract owner to modify the existing functions defined in contract `C`. The `add_new_function(string)` take as input a string with the function definition that the user wants to add, calculates the function signature and stores it in the pre-defined `vtable`.

Constructor Expansion (Rule 4.12) With this rule, PROTEUS extends the constructor of the Trampoline. The only difference with the Implementation mode is the injection of insert statements of the default function definitions (Default Function Signature Generation Rule) to the `vtable` (Rule 4.12, Line 10).

```
1    function(){
2      B
3    }
```

⇓**Constructor Expansion**

```
1
2    function default_fallout_function () {
3      version.callcode(msg.data);
4    }
5
6    function () {
7            sig = bytes4(sha3(msg.data));
8            if( vtable[sig] == true) {
9               version.callcode(msg.data);
10           )
11           else{
12            default_fallout_function();
13           }
14   }
```

Rule 4.13: Constructor Expansion rule. The `vtable` is populated with the default function definitions of contract `C`.

Fallout Function Expansion (Rule 4.13) With this rule, PROTEUS extends the functionality of the default fallout function. Because all functions are removed from the `Trampoline`, that means that all function calls will pass through the fallout function. Before delegating any call to contract `C'`, PROTEUS checks that the function that the user calls exists in the `vtable`. If it doesn't exist, then it calls the `default_fallout_function()` function, which is the re-written version of the initial fallout function (`function()`) of `C`. The `default_fallout_function()` function delegates calls to the default fallout function of contract `C'`

4.4 Implementation

Figure 4.8 illustrates the architecture of PROTEUS and its three main components: 1) the MUTABILITY PREFERENCES TABLE, 2) the REWRITING RULES, and 3) the AST REWRITER. PROTEUS takes as input the Solidity source code of a smart contract `C`, an instance of the mutability preferences table and a list of rewriting rules and generates two new contracts: the `Trampoline` and the upgradeable Contract `C'`.

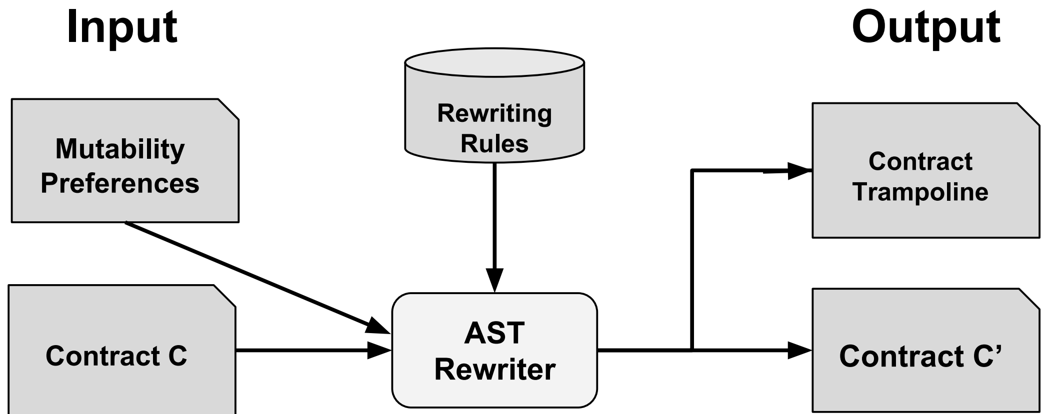


Figure 4.8: System Architecture of Proteus.

Based on the mutability option given by the users, the AST REWRITER selects the corresponding rewriting rules from the database. Next, the REWRITER traverses the AST of each file of the user's project and applies the transformation rules from the database. Finally, both `Trampoline` and Contract *C* are deployed on the Blockchain.

To find the code snippets to be replaced, the EXTRACTOR builds an Abstract Syntax Tree (AST) from its input source code. It then traverses the AST to discover potential transformations as shown in Figure 4.8. The EXTRACTOR maintains a copy of the AST, referred to as the REWRITER, where it applies transformations, without changing the initial AST. When the AST transformation finishes, the REWRITER produces the final source code which is saved as a new file.

We implemented our transformer using Antlr4 [211] and an EBNF version of the Solidity grammar[1]. We used Antlr4, a popular tool that generates parsers from the grammar of the language and allows developers to perform source-to-source transformations efficiently. We chose to implement PROTEUS independently from the Solidity compiler. We didn't follow an approach where we directly change the Solidity compiler, as potential newer versions of the compiler may break the functionality of PROTEUS. Moreover, by operating directly on the Grammar of the Solidity language, we can adapt easily our framework for potential new changes to the language (i.e. deprecate functions).

[1] `https://github.com/solidityj/solidity-antlr4`

4.4.1 Deployability

PROTEUS runs as a cloud service that can be used by developers easily. To use the service, developers only need to provide the source code of the smart contract and select one of the provided modes of upgradeability options. Note that in case that multiple contracts being in the same source file (a common development practice when writing smart contracts in Solidity), then the developer should provide the name of the contract/s that PROTEUS should make upgradeable. The code transformation runs on the background, on the cloud, without affecting developers' local repository. The results will be provided as source code along with a diff report with the changes on the original input contract.

Usage: To use PROTEUS from the command line one issues:

```
1    ./proteus —mode m —input i —output o —user preference
```

where this command defaults to PROTEUS's mode. The values of the parameters that can be specified by the user are:

```
-mode [generate, deploy, update]

-input [src_file, address]

-output [src_file]

-user [variability_table]
```

We also allow users to annotate specific functions of their contracts that they would like to be immutable. Users need to write before a function definition `//@mutable`. The rewrites will automatically match this annotation and will apply its rewriting rules only to those functions that have this annotation.

4.4.2 Deployment Process

The steps followed by the user to get an upgradeable smart contract and deploy it on the Blockchain are as follows (see Figure 4.9): First, the user writes the contract as he/she would normally do. When the contract is finished, the user gives it as input to PROTEUS and as a result, the `Trampoline` contract is generated together with a version of the modified contract `C'`. Next the child contract `C'` is deployed on the Blockchain, and its address is saved. In the final step, the Trampoline contract is

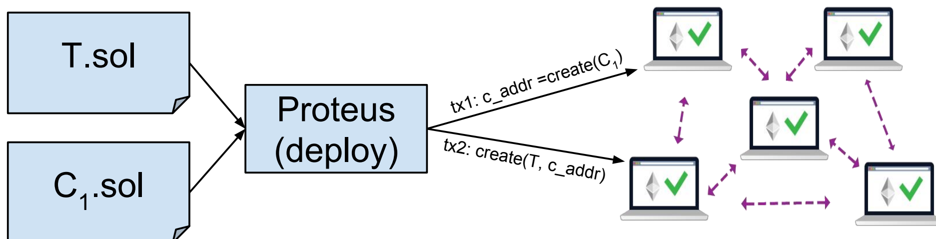


Figure 4.9: Contract Deployment. The Trampoline contract and the initial user's contract is given as an input by the user. PROTEUS generates two transactions: a) The first one publishes the Trampoline contract on the Blockchain b) the second one updates the address that the Trampoline should forward the calls.

deployed, and the address of contract `C'` is given as input. Last, the contract owner shares the address of the `Trampoline` and that of contract `C'` to users that would like to interact with it.

4.4.3 Summary

In this chapter, we presented how can provide upgradeable smart contract on Ethereum, by using the current semantics provided by Solidity and the EVM. We showed how PROTEUS improves the state of the art process of publishing and updating smart contracts on Ethereum and the security checks that it introduces. We presented PROTEUS, a framework that automatically transforms a user's contract such that it is upgradeable. We also showed how PROTEUS can deploy those contracts automatically on the Blockchain, allowing the user to focus only on the development of the logic of the smart contract. Last, we presented the rewriting rules that PROTEUS uses both for different modes and discussed the issues that may arise when someone tries to write such rules manually.

Chapter 5

Conclusions and Future Work

In this chapter, we summarise the achievements and findings of this thesis and state the general conclusions. We also discuss the potential future works that can be followed by the research community, based on our findings.

5.1 Conclusions

In this thesis, we showed how we could use GI and other code transformation techniques to improve the non-functional properties of real-world programs running on top of complex managed runtimes such as the Java and Ethereum Virtual Machine. More specifically, we showed that manual optimisation of non-functional properties of large programs is impractical for programmers. We also showed that manual introduction of advanced functionalities and features (such as upgradability in smart contracts) when is error-prone and bugs are inevitable. Thus, we need frameworks and tools that automatically optimise non-functional properties and rewrite applications such that they support advanced features, without the developer worrying about those transformations. We introduced two such frameworks, ARTEMIS and PROTEUS, and through rigorous statistical experimentation showed that it is possible to improve non-functional properties of programs, cheaply and automatically.

We concluded that developers frequently use underperformed data structures and forget to optimise them with respect to some critical non-functional properties once the functionalities are fulfilled. We introduced ARTEMIS, a novel multi-objective multi-language search-based framework that automatically selects and optimises

Darwinian data structures and their arguments in a given program. ARTEMIS is language agnostic, meaning it can be easily adapted to any programming language; extending ARTEMIS to support C++ took approximately 4 days. Given as input a data structure store with Darwinian implementations, it can automatically detect and optimise them along with any additional parameters to improve the non-functional properties of the given program. In a large empirical study on 5 DaCapo benchmarks, 30 randomly sampled projects and 8 well-written popular Github projects, ARTEMIS found improvement for all of them. Depending on the domain and the nature of the application, some of those improvements can be considered substantial with many clients benefiting from them. ARTEMIS achieved performance improvements for *every* project in 5 Java projects from DaCapo benchmark, 8 popular projects and 30 uniformly sampled projects from GitHub, and one C++ program. ARTEMIS achieved 4.8%, 10.1%, 5.1% median improvement for runtime, memory and CPU usage. ARTEMIS found such improvements making small changes in the source code; the median number of lines ARTEMIS changes is 5. Thus, ARTEMIS is practical and can be easily used on other projects. At last, we estimated the cost of optimising a program in machine hours to be £1.25 on average per day. Therefore, we conclude that ARTEMIS is a practical tool for optimising the data structures in large real-world programs.

We also concluded that even though Ethereum Blockchain does not allow the code of a contract to be upgraded after published on the Blockchain, it is possible to update the code of a smart contract, if the developer uses the provided semantics of the Ethereum Virtual Machine correctly. We showed that understanding the EVM semantics and using them is quite a challenging task which has led to many buggy contracts published on the Blockchain; even though experts wrote the code of the contracts. By introducing PROTEUS, we showed that it is possible to automate the challenging process of writing smart contracts that use advanced language properties that may lead to more bugs, such as upgradeable contracts.. PROTEUS allows developers to have better control over their smart contracts, by providing them with different mutability modes. Also, PROTEUS improves the security of smart contract

by forcing users to check the address of the contract carefully that they are calling, avoiding accidental loss of funds that may occur by calling the wrong contract.

5.2 Future Work

In this section, we present potenital future research work that can be done based on this work.

Darwinian Data Structure Selection We showed how ARTEMIS can be applied to improve the performance of a variety of programs and libraries written in Java and C++. In the future, we can apply ARTEMIS in a more significant number of libraries, and evaluate both the performance improvement of the test suite that comes with the library but also evaluate the performance of the programs that depend and use the library. For instance, we showed in this thesis, how we could improve the performance of the Google Guava library by evaluating its test suite, but in the future, we can measure how the performance of programs that use Google Guava is affected.

Even though most programs that ARTEMIS optimised in this thesis come with a test suite, some applications may have a limited test suite that does not represent actual program behaviour. For future work, we could use random testing to generate test suites automatically and then apply ARTEMIS to improve the program's performance. In the preliminary experiments that we performed, ARTEMIS improved the performance of the program, similarly to how it improved programs with given test-suites. For future work, we can further extend random testing and present a rigorous statistical evaluation. ARTEMIS used multi-objective search and more specifically the NSGA-II algorithm to find optimal solutions. In the future, we can apply other meta-heuristic search algorithms and evaluate and compare their ability to find faster and better solutions. Last, we can extend ARTEMIS to be applied in other programming languages such as CSharp or Scala and also for improving the performance of code that targets mobile devices (*e.g.,* Android).

Upgradeable smart contracts PROTEUS performs code transformations such that the contract can be updated with optimised versions and for bug fixing. To achieve this, as we described previously, PROTEUS introduces additional meta-variables and

uses call indirections mechanisms. This, however, may increase the gas consumption of the contract as more operations are performed. In the future, we should do a full analysis of the gas overhead that each transformation introduces and the trade-offs between optimised newer contract versions vs the introduced by PROTEUS gas overhead. Also, PROTEUS was the first step to enable optimisations for smart contracts on Ethereum, but this is only the first step necessary to apply GI on the Blockchain. In future work, we should build a new framework that relies on the idea introduced by the Darwinian Data Structure Selection and applies GI to automatically improve non-functional properties (such as gas or energy consumption) or functional properties (such as bug fixing through automatic test generation) of smart contracts.

Bibliography

[1] Michael A Lones. Genetic programming tutorial. http://www.macs.hw.ac.uk/~ml355/common/thesis/c6.html, 2018. [Online; accessed 31-August-2018].

[2] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1375–1382, New York, NY, USA, 2015. ACM.

[3] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[4] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *Work Pap*, 2016.

[5] Takenobu T. Ethereum evm illustrated. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf, 2018.

[6] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.

[7] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. *ACM Sigplan Notices*, 44(6):419–430, 2009.

[8] Ronald J. Nowling. Gotchas with Scala Mutable Collections and Large Data Sets. http://rnowling.github.io/software/engineering/

2015/07/01/gotcha-scala-collections.html, 2015. [Online; accessed 18-February-2017].

[9] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference On*, pages 151–160. IEEE, 2008.

[10] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 442–446. IEEE, 2017.

[11] Brett Hardin. Companies with hacking cultures fail. https://blog.bretthard.in/companies-with-hacking-cultures-fail-b8907a69e3d, 2016. [Online; accessed 25-February-2017].

[12] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.

[13] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, 2007.

[14] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[15] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan 2004.

[16] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 240–249. IEEE, 2005.

[17] Mark Harman, Robert Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1351–1358. Morgan Kaufmann Publishers Inc., 2002.

[18] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.

[19] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[20] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[21] Melanie Mitchell, John H Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing. In *Advances in neural information processing systems*, pages 51–58, 1994.

[22] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[23] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[24] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[25] Lingbo Li, Mark Harman, Emmanuel Letier, and Yuanyuan Zhang. Robust next release problem: handling uncertainty during optimization. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1247–1254. ACM, 2014.

[26] L. Li, M. Harman, F. Wu, and Y. Zhang. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering*, 43(6):580–596, June 2017.

[27] Lingbo Li. *Exact analysis for requirements selection and optimisation*. PhD thesis, UCL (University College London), 2017.

[28] Peter A Whigham, Caitlin A Owen, and Stephen G Macdonell. A baseline model for software effort estimation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):20, 2015.

[29] F. Sarro, A. Petrozziello, and M. Harman. Multi-objective software effort estimation. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 619–630, May 2016.

[30] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes. Using tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering*, 18(3):506–546, Jun 2013.

[31] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[32] Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. Sbselector: Search based component selection for budget hardware. In *International Symposium on Search Based Software Engineering*, pages 289–294. Springer, 2015.

[33] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[34] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[35] Betty HC Cheng and Joanne M Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[36] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[37] Ian F Alexander and Ljerka Beus-Dukic. *Discovering requirements: how to specify products and services*. John Wiley & Sons, 2009.

[38] Lionel C Briand and Isabella Wieczorek. Resource estimation in software engineering. *Encyclopedia of software engineering*, 2002.

[39] Tim Menzies, Zhihao Chen, Jairus Hihn, and Karen Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, 32(11):883–895, 2006.

[40] Steve McConnell. *Software estimation: demystifying the black art*. Microsoft press, 2006.

[41] F. Ferrucci, C. Gravino, R. Oliveto, F. Sarro, and E. Mendes. Investigating tabu search for web effort estimation. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 350–357, Sept 2010.

[42] Filomena Ferrucci, Mark Harman, and Federica Sarro. Search-based software project management. In *Software Project Management in a Changing World*, pages 373–399. Springer, 2014.

[43] Magne Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.

[44] Colin Kirsopp, Martin Shepperd, and John Hart. Search heuristics, case-based reasoning and software project effort prediction. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1367–1374. Morgan Kaufmann Publishers Inc., 2002.

[45] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 5–18, New York, NY, USA, 2014. ACM.

[46] Mark Harman, Yue Jia, Jens Krinke, William B Langdon, Justyna Petke, and Yuanyuan Zhang. Search based software engineering for software product line engineering: a survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 5–18. ACM, 2014.

[47] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, June 1982.

[48] Robert V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252, 1996.

[49] G. Suganya and S. Neduncheliyan. A study of object oriented testing techniques: Survey and challenges. In *2010 International Conference on Innovative Computing Technologies (ICICT)*, pages 1–5, Feb 2010.

[50] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, March 2011.

[51] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[52] M. O'Keeffe and M. O. Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–260, March 2006.

[53] Emelie Engström and Per Runeson. *A Qualitative Survey of Regression Testing Practices*, pages 3–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[54] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[55] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203 – 249, 2010.

[56] William B Langdon and Mark Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

[57] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 1–9. ACM, 1999.

[58] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *2nd International Symposium on Search Based Software Engineering*, pages 143–152, Sept 2010.

[59] Tanja EJ Vos, Arthur I Baars, Felix F Lindlar, Peter M Kruse, Andreas Windisch, and Joachim Wegener. Industrial scaled automated structural testing with the evolutionary testing tool. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 175–184. IEEE, 2010.

[60] Andrea Arcuri, Muhammad Zohaib Z Iqbal, Lionel C Briand, et al. Black-box system testing of real-time embedded systems using random and search-based testing. *ICTSS*, 10:95–110, 2010.

[61] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[62] Justyna Petke, Saemundur Haraldsson, Mark Harman, David White, John Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[63] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[64] Paul Walsh and Conor Ryan. Automatic conversion of programs from serial to parallel using genetic programming-the paragen system. In *PARCO*, pages 415–422. Citeseer, 1995.

[65] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[66] Michael Orlov and Moshe Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1043–1050. ACM, 2009.

[67] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)*, 30(6):152, 2011.

[68] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[69] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[70] William B Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 951–958. ACM, 2014.

[71] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[72] Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 1003–1006, New York, NY, USA, 2008. ACM.

[73] Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering, PP (99)*, pages 1–1, 2016.

[74] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[75] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316. ACM, 2010.

[76] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, Jan 2012.

[77] Apple. A message to our customers about iphone batteries and performance. `https://www.apple.com/uk/iphone-battery-and-performance/`, 2017. [Online; accessed 31-August-2018].

[78] Independent. Apple admits to intentionally slowing down iphones as they get older. `https://www.independent.co.uk/life-style/gadgets-and-tech/news/apple-iphones-slow-down-old-models-smartphone-speed-ios-updates-a8121906.html`, 2017. [Online; accessed 31-August-2018].

[79] Shuhaizar Daud, R Badlishah Ahmad, and Nukala S Murthy. The effects of compiler optimisations on embedded system power consumption. *International Journal of Information and Communication Technology*, 2(1-2):73–82, 2009.

[80] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, May 2003.

[81] M. Boussaa, O. Barais, B. Baudry, and G. Sunyï£Čï¿l'. Notice: A framework for non-functional testing of compilers. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 335–346, Aug 2016.

[82] HK Lam, SH Ling, Frank HF Leung, and Peter Kwong-Shun Tam. Tuning of the structure and parameters of neural network using an improved genetic algorithm. In *Industrial Electronics Society, 2001. IECON'01. The 27th Annual Conference of the IEEE*, volume 1, pages 25–30. IEEE, 2001.

[83] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine learning*, 46(1):131–159, 2002.

[84] Chih-Hung Wu, Gwo-Hshiung Tzeng, Yeong-Jia Goo, and Wen-Chang Fang. A real-valued genetic algorithm to optimize the parameters of support vector machine for predicting bankruptcy. *Expert systems with applications*, 32(2):397–408, 2007.

[85] Carlos Eiras-Franco, Leslie Kanthan, Amparo Alonso-Betanzos, and David Martınez-Rego. Scalable approximate k-nn graph construction based on locality sensitive hashing.

[86] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 321–332, New York, NY, USA, 2016. ACM.

[87] Jinn-Tsong Tsai, Jyh-Horng Chou, and Tung-Kuan Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks*, 17(1):69–80, 2006.

[88] Ranjit K Roy. *A primer on the Taguchi method*. Society of Manufacturing Engineers, 2010.

[89] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652, New York, NY, USA, 2014. ACM.

[90] Irene Manotas, James Clause, and Lori Pollock. Exploring evolutionary search strategies to improve applications' energy efficiency. In Thelma Elita

Colanzi and Phil McMinn, editors, *Search-Based Software Engineering*, pages 278–292, Cham, 2018. Springer International Publishing.

[91] Bobby R Bruce. Energy optimisation via genetic improvement: A sbse technique for a new era in software development. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 819–820. ACM, 2015.

[92] Abram Hindle. Green software engineering: the curse of methodology. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 46–55. IEEE, 2016.

[93] Nevon Brake, James R. Cordy, Elizabeth Dan Y, Marin Litoiu, and Valentina Popes U. Automating discovery of software tuning parameters. In *Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, 2008.

[94] Frank Hutter, Domagoj Babic, Holger H. Hoos, and A.J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007.

[95] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[96] Allen Troy Acree Jr. On mutation. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1980.

[97] Yue Jia, Mark Harman, William B Langdon, and Alexandru Marginean. Grow and serve: Growing django citation services using sbse. In *International Symposium on Search Based Software Engineering*, pages 269–275. Springer, 2015.

[98] William B Langdon. Genetic improvement of software for multiple objectives. In *International Symposium on Search Based Software Engineering*, pages 12–28. Springer, 2015.

[99] Nathan Burles, Edward Bowles, Bobby R Bruce, and Komsan Srivisut. Specialising guavaï£ćï¿Ăï¿Źs cache to reduce energy consumption. In *International Symposium on Search Based Software Engineering*, pages 276–281. Springer, 2015.

[100] Yue Jia, Fan Wu, Mark Harman, and Jens Krinke. Genetic improvement using higher order mutation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 803–804. ACM, 2015.

[101] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1513–1520. ACM, 2017.

[102] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, Donald Lawrence, and Earl Barr. Darwinian data structure selection. *arXiv preprint arXiv:1706.03232*, 2017.

[103] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. Optimising darwinian data structures on google guava. In *International Symposium on Search Based Software Engineering*, pages 161–167. Springer, 2017.

[104] David R White, Leonid Joffe, Edward Bowles, and Jerry Swan. Deep parameter tuning of concurrent divide and conquer algorithms in akka. In *European Conference on the Applications of Evolutionary Computation*, pages 35–48. Springer, 2017.

[105] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[106] Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*, pages 18–33. Springer, 2016.

[107] Bobby R Bruce, Jonathan M Aitken, and Justyna Petke. Deep parameter optimisation for face detection using the viola-jones algorithm in opencv. In *International Symposium on Search Based Software Engineering*, pages 238–243. Springer, 2016.

[108] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, 2010.

[109] W. B. Langdon and J. P. Nordin. *Seeding Genetic Programming Populations*, pages 304–315. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[110] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[111] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1775–1782, New York, NY, USA, 2008. ACM.

[112] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. *Multi-objective Improvement of Software Using Co-evolution and Smart Seeding*, pages 61–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[113] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug 2011.

[114] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1427–1434, New York, NY, USA, 2011. ACM.

[115] W.B. Langdon and M. Harman. Optimizing existing software with genetic programming. *Evolutionary Computation, IEEE Transactions on*, 19(1):118–135, Feb 2015.

[116] Justyna Petke, Mark Harman, WilliamB. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In Miguel Nicolau, Krzysztof Krawiec, MalcolmI. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, JuanJ. Merelo, VictorM. Rivas Santos, and Kevin Sim, editors, *Genetic Programming*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer Berlin Heidelberg, 2014.

[117] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN Notices*, volume 42, pages 245–260. ACM, 2007.

[118] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[119] Oracle. Introduction to collections. `https://docs.oracle.com/javase/tutorial/collections/intro/index.html`, 2016. [Online; accessed 31-August-2018].

[120] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[121] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[122] Dayong Gu, Clark Verbrugge, and Etienne M Gagnon. Relative factors in performance analysis of java virtual machines. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121. ACM, 2006.

[123] Matthias Hauswirth, Peter F Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-priented applications. *ACM Sigplan Notices*, 39(10):251–269, 2004.

[124] Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: A system for building customized java program analysis tools. *ACM SIGPLAN Notices*, 41(10):153–168, 2006.

[125] Peter F Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *Virtual Machine Research and Technology Symposium*, pages 57–72, 2004.

[126] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[127] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *ACM SIGPLAN Notices*, volume 38, pages 169–186. ACM, 2003.

[128] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming*, pages 429–451. Springer, 2006.

[129] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applica-

tions. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 59–70. ACM, 2008.

[130] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. *ACM Sigplan Notices*, 43(10):127–142, 2008.

[131] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. *ACM Sigplan Notices*, 45(6):174–186, 2010.

[132] Fabian Nagel, Gavin M Bierman, Aleksandar Dragojevic, and Stratis Viglas. Self-managed collections: Off-heap memory management for scalable query-dominated collections. In *EDBT*, pages 61–71, 2017.

[133] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

[134] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B Sartor, and Wolfgang De Meuter. Just-in-time data structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 61–75. ACM, 2015.

[135] Diego Costa and Artur Andrzejak. Collectionswitch: a framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26. ACM, 2018.

[136] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *ACM SIGPLAN Notices*, volume 46, pages 86–97. ACM, 2011.

[137] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming*

Systems Languages & Applications, OOPSLA '13, pages 167–182, New York, NY, USA, 2013. ACM.

[138] IBM. T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2009. [Online; accessed 18-February-2017].

[139] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. *SIGPLAN Not.*, 43(10):367–384, October 2008.

[140] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, pages 63–74. ACM, 2013.

[141] Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. Technical report, Technical Report 4-12, University of Kent, 2012.

[142] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Homestead Revision, 2015.

[143] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[144] Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

[145] Marc Pilkington. Blockchain technology: principles and applications. *Browser Download This Paper*, 2015.

[146] Melanie Swan. Connected car: quantified self becomes quantified car. *Journal of Sensor and Actuator Networks*, 4(1):2–29, 2015.

[147] George Foroglou and Anna-Lali Tsilidou. Further applications of the blockchain. In *12th Student Conference on Managerial Science and Technology*, 2015.

[148] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 839–858. IEEE, 2016.

[149] Alex Zarifis, Leonidas Efthymiou, Xusen Cheng, and Salomi Demetriou. Consumer trust in digital currency enabled transactions. In *International Conference on Business Information Systems*, pages 241–254. Springer, 2014.

[150] Yu Zhang and Jiangtao Wen. An iot electric business model based on the protocol of bitcoin. In *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*, pages 184–191. IEEE, 2015.

[151] Mike Sharples and John Domingue. The blockchain and kudos: A distributed system for educational record, reputation and reward. In *European Conference on Technology Enhanced Learning*, pages 490–496. Springer, 2016.

[152] Matthias Mettler. Blockchain technology in healthcare: The revolution starts here. In *e-Health Networking, Applications and Services (Healthcom), 2016 IEEE 18th International Conference on*, pages 1–3. IEEE, 2016.

[153] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[154] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of monero's blockchain. *IACR Cryptology ePrint Archive*, 2017:338, 2017.

[155] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[156] Val A Red. Practical comparison of distributed ledger technologies for iot. In *Disruptive Technologies in Sensors and Sensor Systems*, volume 10206, page 102060G. International Society for Optics and Photonics, 2017.

[157] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[158] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. *R3 CEV, August*, 2016.

[159] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[160] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[161] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[162] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[163] Building a private ethereum consortium. `https://www.microsoft.com/developerblog/2018/06/01/creating-private-ethereum-consortium-kubernetes/`. Accessed: 2018-08-16.

[164] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. *R3 CEV, August*, 2016.

[165] Vikram Dhillon, David Metcalf, and Max Hooper. Recent developments in blockchain. In *Blockchain Enabled Applications*, pages 151–181. Springer, 2017.

[166] Lijun Wu, Kun Meng, Shuo Xu, Shuqin Li, Meng Ding, and Yanfeng Suo. Democratic centralism: A hybrid blockchain architecture and its applications in energy internet. In *Energy Internet (ICEI), IEEE International Conference on*, pages 176–181. IEEE, 2017.

[167] Ian Grigg. Eos, an introduction. *Whitepaper) iang. org/papers/EOS_An_Introduction. pdf*, 2017.

[168] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 182–191. IEEE, 2016.

[169] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *Open and Big Data (OBD), International Conference on*, pages 25–30. IEEE, 2016.

[170] Ariel Ekblaw, Asaph Azaria, John D Halamka, and Andrew Lippman. A case study for blockchain in healthcare:âĂIJmedrecâĂİ prototype for electronic health records and medical research data. In *Proceedings of IEEE open & big data conference*, volume 13, page 13, 2016.

[171] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 706–719. ACM, 2015.

[172] etherdice.com. Etherdice is down for maintenance. we are having troubles with our smart contract and will probably need to invoke the fallback mechanism. https://www.reddit.com/r/ethereum/duplicates/47f028/etherdice_is_down_for_maintenance_we_are_having/, 2016. [Online; accessed 31-October-2017].

[173] stackoverflow. Difference between call, callcode and delegatecall. https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall, 2018. [Online; accessed 31-August-2018].

[174] Coindesk. Understanding the dao hack. https://www.coindesk.com/understanding-dao-hack-journalists/, 2016. [Online; accessed 31-October-2017].

[175] Phil Daian Lorenz Breidenbach Emin Gun Sirer, Ari Juels. An in-depth look at the parity multisig bug, 2017.

[176] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. *arXiv preprint arXiv:1702.05511*, 2017.

[177] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[178] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[179] Thomas Cook, Alex Latham, and Jae Hyung Lee. Dappguard: Active monitoring and defense for solidity smart contracts.

[180] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[181] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. NDSS, 2018.

[182] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. 2018.

[183] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[184] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, New York, NY, USA, 2016. ACM.

[185] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[186] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[187] Nell Dale and Henry M Walker. *Abstract data types: specifications, implementations, and applications*. Jones & Bartlett Learning, 1996.

[188] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[189] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Information & Software Technology*, 81:97–111, 2017.

[190] Lingbo Li. *Exact analysis for requirements selection and optimisation*. PhD thesis, UCL (University College London), 2017.

[191] Lingbo Li. Exact analysis for next release problem. In *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, pages 438–443. IEEE, 2016.

[192] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[193] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[194] Matthew Arnold, Michael Hind, and Barbara G Ryder. Online feedback-directed optimization of java. In *ACM SIGPLAN Notices*, volume 37, pages 111–129. ACM, 2002.

[195] Lingbo Li, Mark Harman, Emmanuel Letier, and Yuanyuan Zhang. Robust next release problem: handling uncertainty during optimization. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1247–1254. ACM, 2014.

[196] Michael P Fay and Michael A Proschan. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys*, 4:1, 2010.

[197] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, April 2003.

[198] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[199] Cryptocompare. The dao, the hack, the soft fork and the hard fork. `https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/`, 2017. [Online; accessed 31-October-2017].

[200] ConsenSys. Smart contract best practices. `https://github.com/ConsenSys/smart-contract-best-practices`, 2017. [Online; accessed 31-August-2018].

[201] Ethereum Core Developers. Solidity programming language. `http://solidity.readthedocs.io/en/latest/`, 2016. [Online; accessed 31-October-2017].

[202] www.ethnews.com. Meet the unknown, maverick white hat who rescued additional accounts during this week's attack. `https://www.ethnews.com/meet-the-unknown-maverick-white-hat-who-rescued-accounts-missed-by-the-whg-during-this-weeks-attack`, 2017. [Online; accessed 31-August-2018].

[203] www.vice.com. How coders hacked back to rescue 208 million dollars in ethereum. `https://motherboard.vice.com/en_us/article/qvp5b3/how-ethereum-coders-hacked-back-to-rescue-dollar208-million-in-ethereum`, 2017. [Online; accessed 31-August-2018].

[204] ParityMultisigRecoveryReconciliation. Parity multisig vulnerability - white hat group rescue reconciliation. `https://github.com/bokkypoobah/ParityMultisigRecoveryReconciliation`, 2017. [Online; accessed 31-August-2018].

[205] Aragon Community. Advanced solidity code deployment techniques. `https://blog.aragon.org/advanced-solidity-code-deployment-techniques-dc032665f434/`, 2017. [Online; accessed 31-August-2018].

[206] cointelegraph.com. Coincheck to refund all customers affected by hack, faced by community support. `https://cointelegraph.com/news/coincheck-to-refund-all-customers-affected-`

by-hack-faced-by-community-support, 2017. [Online; accessed 31-August-2018].

[207] Coindesk. 7 million dollars lost in coindash ico hack, 2017.

[208] Lawrence Abrams. Ethereum phishing attack nets criminals 15k in two hours. https://www.bleepingcomputer.com/news/security/ethereum-phishing-attack-nets-criminals-15k-in-two-hours/, 2017. [Online; accessed 31-October-2017].

[209] Solidity documentation.

[210] Solidity assembly. http://solidity.readthedocs.io/en/v0.4.24/assembly.html. Accessed: 2018-07-23.

[211] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.