Towards Solidity Smart Contract Efficiency Optimization through Code Mining

Avik Banerjee
Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things
Hamburg University of Technology
Hamburg, Germany
avik.banerjee@tuhh.de

Michael Sober
Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things
Hamburg University of Technology
Hamburg, Germany
michael.sober@tuhh.de

Stefan Schulte
Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things
Hamburg University of Technology
Hamburg, Germany
stefan.schulte@tuhh.de

Abstract

Deploying smart contracts and invoking their functions on blockchains incur gas costs, which depend on the operations executed by those functions. This makes optimizing the gas cost of smart contract functions a rewarding goal. However, existing approaches to gas cost optimization of smart contracts mainly involve rule-based optimization or automatic optimization for specific types of patterns.

In this paper, we discuss a novel approach to automatically retrieving optimized versions of Solidity functions from a repository of smart contracts. The system identifies and suggests gas-efficient alternatives that maintain functional equivalence by comparing the opcode sequences of individual functions. We evaluate this approach on a dataset of 16,529 functions from real-world contracts, demonstrating substantial gas savings, as high as 34% on average when considering the most similar functions.

CCS Concepts

• **Software and its engineering** → **Software libraries and repositories**; *Software maintenance tools*; **Software performance**; • **Computing methodologies** → *Distributed algorithms*; • **Information systems** → **Similarity measures**.

Keywords

blockchain, smart contracts, gas cost, code mining, optimization, code similarity, control flow graph

ACM Reference Format:
Avik Banerjee, Michael Sober, and Stefan Schulte. 2025. Towards Solidity Smart Contract Efficiency Optimization through Code Mining. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3672608.3707768

1 Introduction

Since the advent of blockchain technology in 2008 [37], creating a fully decentralized Web has gained traction, inspiring developments such as cryptocurrencies and non-fungible tokens [38].

First-generation blockchains like Bitcoin and Litecoin primarily offer a distributed ledger consisting of append-only blocks containing transactions, replacing the need for a central authority with a decentralized network of computing nodes [45]. Vitalik Buterin [47] introduced second-generation blockchains, adding the functionality of smart contracts, which are user-defined programs executed upon transaction. These smart contracts run within a blockchain's environment, which handles their execution and resulting state changes. The Ethereum Virtual Machine (EVM) is the most notable of these environments, responsible for enforcing rules to determine valid state transitions following the inclusion of a block in the Ethereum blockchain. The EVM manages the Ethereum network as a *distributed state machine*, consisting of a global state and its associated transactions [47].

As Ethereum and other second-generation blockchains continue to grow in popularity, the number of deployed smart contracts increases. The immutability of blockchains makes it nearly impossible to alter deployed smart contracts, underscoring the importance of writing error-free and efficient code [9]. When a smart contract is deployed or invoked, it incurs gas costs—borne by users during transactions [47]. Gas is a measure of the computational effort required to execute transactions on the Ethereum network [47]. Since gas costs correlate with the computational effort required for contract execution, optimizing contracts for lower gas cost can also reduce energy usage [40]. Therefore, optimizing smart contracts before deployment can reduce costs for users and developers while improving energy efficiency.

There are various strategies for source code optimization, ranging from applying rule-based modifications to automatically mining code patterns from repositories [42]. Optimization can address efficiency, maintainability, reusability, comprehensibility, and eliminating code smells [39].

Code mining involves analyzing an extensive collection of source files to extract patterns of interest. This allows for discovering optimization patterns without manual rule definition, including innovative and uncommon patterns or those unknown to the developer defining those rules. Code mining techniques can also summarize and visualize algorithms [32], aiding in the search for optimization patterns in smart contracts.

The primary distinction between optimizing conventional software and smart contracts is that smart contract optimization primarily focuses on reducing gas costs [2]. In contrast, traditional programs are usually optimized for time or resource efficiency. This

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707768

introduces unique constraints that demand specialized pattern-recognition techniques for constraint-based optimization [18].

Current approaches to smart contract optimization often rely on detecting predefined patterns that are known to offer optimization potential [1], as discussed in detail in Section 5. However, this approach narrows the scope of optimization and may miss uncommon or unknown patterns. Our work aims to provide a first approach towards detecting optimizing patterns for functions in a smart contract. The algorithm first compiles a contract to extract its runtime bytecode and then creates a Control Flow Graph (CFG) to extract the opcode sequences for individual functions in the contract. These sequences are then used to estimate the gas costs of these functions and to compare them to other, similarly processed functions in a repository. This comparison then outputs functions which are functionally similar but are more gas-efficient than the function under test. These outputs serve as suggestions to the developer, who have the option to modify their own code to incorporate the optimizations.

We focus on Solidity contracts, the most widely used language for smart contracts [48]. It is feasible to obtain a large collection of smart contracts, both from the verified deployed contracts on the Ethereum mainnet (through Etherscan[1], and also from public developer repositories on GitHub[2].

We evaluate our approach on the functions we obtain from processing and analyzing the contracts received from Etherscan and GitHub. We compare functions in the dataset to get more efficient versions of the existing functions and select functions that are most similar to them and functions that are similar but with the maximum gas cost difference. This leads to a mean gas reduction of around 47%, when we focus on maximizing the gas cost difference and around 33% when we focus on the highest functional similarity score.

The structure of this paper is as follows: Section 2 introduces the preliminaries relevant to our approach. Section 3 explains the code mining approach in detail. Section 4 gives details of the implementation and evaluation of our strategy. Section 5 provides information on the related work in this field. Section 6 presents concluding remarks and future directions for improvement.

2 Preliminaries

In this section, we introduce the preliminaries relevant to our work. We explain smart contracts and the concept of code mining.

2.1 Smart Contracts

In 2014, the second generation of blockchains [10] was introduced with the ability to run Turing-complete programs in the form of smart contracts.

With Ethereum, it is theoretically possible to implement and run any service on the blockchain network. In principle, a developer can use any high-level programming language to create a smart contract and compile it into bytecode, a platform-agnostic code that the EVM can execute. *Deployment bytecode* contains the constructor code and initialization logic, whereas *runtime bytecode* includes the executable functions and logic used to process transactions and interact with the contract. The bytecode is then deployed on the blockchain. This allows the program to run on each node in the network and interact with other programs or users on the same network via transactions.

In the original Ethereum paper, Buterin described smart contracts as "systems which automatically move digital assets according to arbitrary prespecified rules" [10]. A smart contract is a program that resides on the blockchain and is executed whenever a user interacts with it.

Every smart contract holds predefined functions that can be invoked once deployed on the blockchain. For instance, a smart contract is created using the constructor method. One can create and take ownership of a contract by sending a deployment transaction to the blockchain with the bytecode of the contract and calling its constructor function [22, 47]. A self-destruct or check-balance function can be generated and used after deployment similarly. Every Solidity smart contract has a distinct 20-byte address, can hold an Ether balance, and may respond to incoming transactions using its methods [4]. Users create and interact with the smart contracts, which are then stored on the blockchain. Each smart contract can store code and data accessed from the blockchain. The network and the miner nodes that try to append new blocks provide the computing power for running the smart contracts.

Ethereum introduced the programming language Solidity, which can be used to write smart contracts that run on the EVM. The EVM is a quasi-Turing complete machine [47] in that it restricts the over-utilization of resources by limiting the *gas cost*. Each transaction on the Ethereum network has an associated gas cost, which has to be paid for by the initiator of the transaction. A smart contract is a collection of operations that incur gas costs [47].

Since the changes in the Ethereum Improvement Proposal (EIP) 1559 [11] were implemented, every transaction needs to specify a priority tip which is paid out directly to the miner, along with the highest amount the user is willing to pay for the transaction. Every block has a base gas fee associated with it, which gets added to the priority tip included by the user, and is burned when the block is created. The user has to only pay for the amount of gas used. The gas limitations are essential to prevent smart contracts from going into infinite loops and exponential runtime, which can use up node resources and disrupt the entire network [47]. The Ethereum Yellow Paper [47] lists the gas cost of individual opcodes executed by the EVM. Reducing the number of operations and replacing expensive operations with cheaper ones can reduce the gas cost and, indirectly, the actual energy consumption of a smart contract [40].

2.2 Code Mining

Code mining describes the idea of finding code patterns by analyzing the large corpus of source code available in software repositories [5]. Such patterns depend on the type of end-analysis desired and can point to software development best practices, identifying duplicated vulnerable code, or optimizing constructs, among others [21].

Code mining is usually done with source code in a high-level programming language but can also be done with bytecode [46]. Good programming practices follow certain design principles specific to the language used and the application built. Still, at the

[1] https://etherscan.io
[2] https://github.com

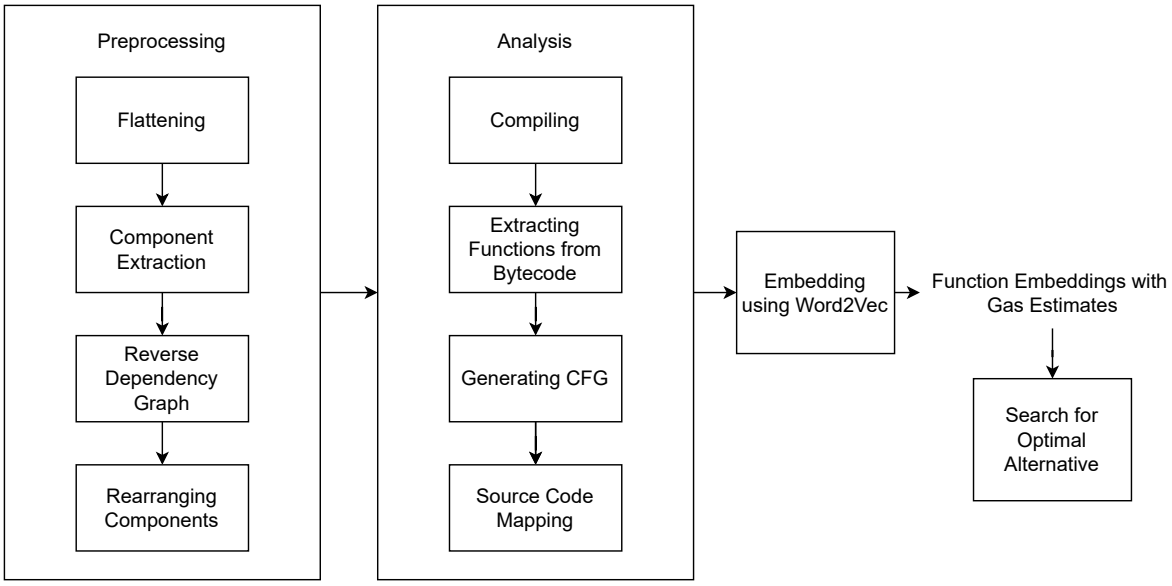

Figure 1: Overview of the code mining process

same time, general rules apply to nearly all software projects. Violations of these design principles are called code smells [39]. One of the primary purposes code mining has been used for is to detect and possibly remove code smells automatically. Different methods can be used for code mining, depending on the optimizations that should be achieved [39].

As the name suggests, code mining aims to analyze source code using data mining methods. Khatoon et al. [23] describe three mining techniques commonly used to identify bugs and code smells while developing software: *Rule mining techniques* exploit existing software projects to find some rules that can be applied to the code. *Clone detection* is needed when developers reuse code segments of other developers by just copying them. This often saves time, but it can also introduce incomprehensible errors in the code that need to be found. Lastly, *API usage* is investigated. Calling an external library often requires some checks beforehand or afterward [23]. In these areas, code mining techniques can analyze the source code and improve its quality by finding code smells.

This paper provides a mechanism for analyzing and extracting functions from a Solidity smart contract and comparing it against a repository of other functions to find another functionally similar but more gas-efficient one. Our approach utilizes the CFG of a compiled contract to extract the opcode sequences for the individual functions. These sequences are then used to estimate these functions' gas costs and derive a similarity score. This extracts optimizing patterns from existing source code without depending on developer-defined rules.

3 Code Mining to Optimize Smart Contracts

In this section, we provide details on the methodology used to perform code mining. The process (see the overview in Figure 1) includes data preprocessing, followed by an analysis of the contracts to extract the functions and estimate their gas costs. The functions are then embedded, and the embeddings are used to search for similar but more efficient ones in a dataset. The code for our approach is available in a GitHub repository[3].

3.1 Data Preprocessing

Most real-world smart contracts import preexisting libraries, interfaces, and contracts to re-utilize existing functionalities. A compiler like `solc` requires the source code of these dependencies to compile a smart contract. For this, `solc` mandates dependencies to be defined before they are imported into a contract. To ensure this and also to use contracts that import only valid dependencies, we used flattened contracts. A flattened contract contains the source code of the contract and its dependencies in the same file so that the compiler has all the information at hand while compiling the contract and does not throw import errors. Contracts can be flattened using flatteners (such as the Truffle flattener[4]), or by simply concatenating the individually retrieved source codes of the contract and its dependencies. In this work, we utilize the bytecode generated by the Solidity compiler to estimate the gas costs of functions and detect the similarity between functions. Hence, the successful compilation of a contract is the first step. Ensuring the contract file is processed, and the dependencies are arranged correctly to avoid compilation failure is essential.

To achieve this, the source code is first stripped of comments and unnecessary whitespace characters. Import statements are also removed since flattened contracts should already have relevant definitions in the same file, and many import statements contain relative paths that are invalid to the compiler. The file is then analyzed line-by-line, and all contracts, interfaces, and libraries are

[3]https://anonymous.4open.science/r/solidity_function_analyzer-5CBB
[4]https://www.npmjs.com/package/truffle-flattener

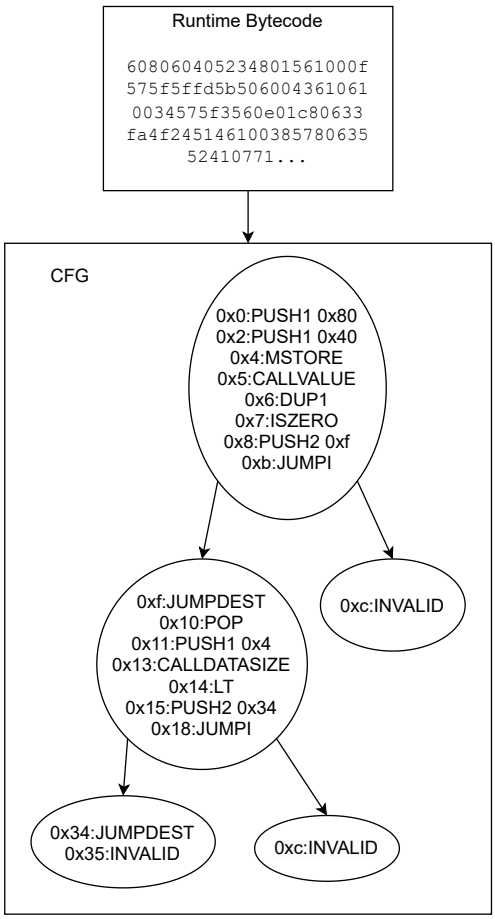

Figure 2: Construction of the CFG

extracted. The definitions of the extracted components are then parsed to determine their direct dependencies.

This information is used to create a reverse dependency graph. A dependency graph [29] is a directed graph that models the dependencies among objects. Let us consider a set S and a transitive relation $R \subseteq S \times S$, such that objects $(a, b) \in R$ only if a depends on b, i.e., b needs to be processed first to process a successfully. The dependency graph modeling this relation R can be defined as

$$G(S, E) \tag{1}$$

where S is the set of vertices (or objects) and E is the set of directed edges between those vertices. E can be defined as

$$E = \{(u, v) : u \in S, v \in S, (u, v) \in R\}. \tag{2}$$

In our case, the set of objects S is the set of contracts, interfaces, and libraries retrieved from a file. We prepare a reverse dependency graph where each directed edge points from a dependency to the contract, interface or library that imports it such that

$$E = \{(v, u) : v \in S, u \in S, (u, v) \in R\}. \tag{3}$$

The information modeled in the reverse dependency graph can then be used to order the dependencies and their imports. For this, we perform a topological sort on the graph. A topological sort [20] is a linear ordering of the vertices of a directed graph such that, for every edge (u, v), u comes before v in the ordering. We use Kahn's algorithm [20] to perform the sorting. This sorted list can be used to order a contract and its redundancies and compile it correctly.

3.2 Analyzing the Contracts

The preprocessed contracts are then analyzed to extract the functions and their opcode sequences. This process involves the following steps:

(1) **Compiling the contracts**: Newer Solidity versions can introduce breaking changes and cause fatal errors in the compilation of a contract. Hence, contracts, libraries, and interfaces specify, in their pragma declarations, the range of supported compiler versions. A pragma declaration is a statement at the beginning of a smart contract that specifies the range of compiler versions supported by the smart contract. A contract and its dependencies have varying compiler version requirements in most cases. The `pragma` declarations in the input contract file are first parsed to determine the compiler versions and ranges stated. This information is then used to determine the optimal compiler version from the available versions. The `solc-select` library[5] is used to set the system compiler version for each file.
As part of the compilation process, the Solidity compiler optimizes existing functions and creates new intermediate ones. It performs additional optimizations through its:
   (a) *opcode-based optimizer* - which performs optimizations on the bytecode after compilation, and the
   (b) *Yul optimizer* - which performs optimizations on an intermediate Yul language representation of the Solidity code.
The contract file is compiled with the opcode-based optimizer and the Yul optimizer turned off. This ensures that most of the functions in the source code are preserved during compilation and can be mapped to their opcode sequences. For each contract in the file, the compiler returns its runtime bytecode and the Application Binary Interface (ABI). The following steps are applied to each separate contract returned by the compiler.

(2) **Extracting functions from the bytecode**: The ABI and runtime bytecode are then used to obtain the function selectors present in the source code. A function selector is the first 4 bytes of the KECCAK256 hash of the function signature. This process eliminates the intermediate functions generated by the compiler during compilation and optimization. These functions then do not correspond to functions in the source code and cannot be used to search for optimizing patterns. To achieve this, the function selectors of the functions in the ABI are calculated and searched for in the bytecode. At the end of this step, we have a list of the functions from the source code that can be mapped to instruction sequences in the bytecode.

[5] https://github.com/crytic/solc-select

(3) **Generating the CFG**: In this step, the runtime bytecode is used to generate a CFG of the contract. A CFG is a directed graph that shows all possible execution paths of a program [6]. To generate a CFG, the instructions in a program are broken down into basic blocks. A basic block is a set of instructions with one entry point and one exit point [6]. A basic block should not have branches or loops. Each graph vertex is such a basic block, and the directed edges show the direction of flow of execution from one basic block to another. CFGs are essential for analyzing the structure of a program and are used in this work to detect parts of the bytecode that correspond to specific functions in the source code. Since a function can call another function and also have loops that can cause execution to jump back to itself, it is essential to analyze the execution flow to get its complete set of opcodes and, hence, the most accurate estimate of its gas cost. An example of a CFG prepared from runtime bytecode is shown in Figure 2. The basic blocks in the figure show the opcodes with their hexadecimal bytecode identifiers and operands.
In this work, the CFG is used to detect sets of instructions from the runtime bytecode corresponding to source code functions. In this case, each basic block is a sequence of opcodes with one entry and one exit point. The CFG is prepared using the `evm_cfg_builder`[6] package. A function may span over one or more basic blocks depending on its number of branches. This information is essential for extracting instruction sequences for functions from the bytecode and mapping them to source code lines.

(4) **Mapping source code to instruction sequences and estimating gas cost**: The generated CFG is parsed, and opcode sequences corresponding to individual functions are extracted. As explained earlier, optimizations are performed automatically by the compiler to remove or merge some other functions and create new intermediate ones. As a result, in some cases, not all of the functions in the source code are present in the runtime bytecode.
Each instruction contained in the CFG is a `pyevmasm`[7] object that also specifies its basic gas fee. `pyevmasm` is an assembler and disassembler library for EVM instructions. The total gas cost of a function is estimated as the sum of the basic gas fees of the constituent opcodes. Finally, the extraction of functions from the source code is performed using the analysis tool Slither [17]. The output from Slither maps lines from the source code to functions identified by their selectors. This information can then be used to map function source code directly to their opcode sequences and, consequently, to their gas costs. The source mapping is essential to directly compare the difference in source code between two functions and offer the developer the choice to modify their code.

3.3 Embedding the Functions

Once we have the sequences of opcodes for the functions, we need to convert them to a form that can be used to determine their similarity. The standard way to do this is to embed the sequences as numerical vectors [7]. In our work, we use the opcode sequence of a function to embed it rather than the source code since two functions that differ in the source code may ultimately be achieving the same goal functionally, and that should be better reflected in the set of instructions executed for each function by the EVM.

We use the Word2Vec model to achieve this. Word2Vec is a technique introduced by Mikolov et al. in 2013 [33, 34] to learn meaningful contextual word representations. This technique uses a neural network to learn embeddings of words from an existing corpus [33] and can hence be trained to learn embeddings for words that do not occur in natural language, such as opcodes. The Word2Vec embeddings can be generated either using the Continuous Bag of Words (CBOW) model, which predicts the current word based on a window of past and future words, or the Continuous Skip-gram model [33], which predicts a window of past and future words based on the current word. In this work, we utilize a Word2Vec model trained on a dataset of opcode sequences using the continuous skip-gram model, which preserves word order information and works well with rare words, as suggested in [43].

The Word2Vec model is applied to the sequence of opcodes for each function, after removing their operands. The operands do not add any functional meaning and can cause erroneous embeddings due to them not being present in the Word2Vec training vocabulary. Each sequence of opcodes is transformed into an $n \times 300$ size matrix where n is the number of opcodes in the sequence, and 300 is the selected length of each vector. To compare two functions, it is necessary to have a single representation for the entire sequence. To simplify this, the mean of the n vectors is calculated to represent a function as a whole. Thus, every data point in the dataset contains the information for each function structured as in Listing 1.

Listing 1: Structure of a datapoint in the dataset

```
{
    name: name of the function,
    ops: opcode sequence of the function from the CFG,
    source: the function source code,
    source_map_lines: the line numbers corresponding to
        the function in the source file,
    embeds: mean embedding vector representing the
        function opcode sequence,
    signature: the function selector from the bytecode,
    contract_name: the name of the contract this function
         belongs to,
    filename: the file to which the contract belongs and
        the source map lines correspond to
}
```

3.4 Searching for Optimized Versions of Functions

Based on the abovementioned steps, we can estimate the gas costs of functions and detect similar ones. This forms the basis of our code mining approach, where we compare a test function to other functions in a repository, find a more optimized similar function, and then analyze the differences that make the second function more efficient. While functional similarity of two pieces of code is an undecidable problem, it can still be approximated [24]. For this work, we detect the similarity of functions using their function selectors and then the cosine similarity of their embeddings. Checking the similarity of the selectors improves the accuracy of this method

[6] https://github.com/crytic/evm_cfg_builder
[7] https://github.com/crytic/pyevmasm

since functions with the same selector usually perform the same task. The cosine similarity function is the normalized dot product of the two embeddings, defined as

$$\frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}||\mathbf{B}|}$$

where **A** and **B** are two vectors. This further checks whether the instruction sequences of the functions are similar enough to eliminate the chance of false positives. The higher the output of this expression, the more similar the two vectors are supposed to be. Finally, we accept only those cases where one function has a lower gas cost estimate than the other.

Once we have a dataset of functions that we can use for code mining, any new incoming contract file can be processed using the steps in this section, and then the functions can be compared against those in the dataset. In the case of a completely new function with no matches in the dataset, the new function can be added to the dataset, thereby improving the dataset dynamically. If we find a more gas-efficient match in the dataset, we can suggest the new function to the developer, who can then decide on incorporating those changes.

4 Evaluation

In this section, we present the results of the evaluation of our approach, including data collection and analysis and examples of optimizing patterns found in the dataset.

4.1 Data Collection

To test our approach's initial viability, we used a dataset of 474 contract files obtained by applying Brandstätter et al.'s method [9] to contracts from Etherscan. This dataset includes both original and optimized versions generated by their algorithm.

Subsequently, we augmented this initial dataset by incorporating 3,308 flattened contracts retrieved from the `Smart Contract Repository` [16]. This repository provides access to all versions of Solidity files scraped from GitHub using the `github-solidity-scraper`[8] tool. To ensure data integrity, we only retrieved the latest scraped versions of the files, thereby excluding incomplete and erroneous contracts. The repository's API also enabled us to obtain flattened versions of contracts, which significantly reduces the presence of incorrect contracts by removing empty, incomplete, and invalid imports.

We created an extensive dataset of 3,782 contracts by combining these two datasets. This combined dataset was then used in its entirety to further test our approach, specifically to identify and analyze optimized versions of functions present within the dataset.

4.2 Analyzing and Embedding the Functions

The contract files are then analyzed using the steps described in Section 3.2. Since these steps involve the compilation of the contracts, first by the Solidity compiler and then by Slither (into their intermediate language, SlithIR), some of the contracts must be discarded due to issues during either of these steps. We solve problems that arise from incorrect syntaxes manually, but other issues arising from Slither's inability to parse certain constructs cannot be solved easily. Apart from this, since the Solidity compiler outputs bytecode only for contracts, not libraries, we cannot compare two libraries directly using this approach. Hence, the dataset of functions we obtain at the end of the analysis contains functions from contracts compiled successfully by both the Solidity compiler and Slither. From the GitHub files, 2,652 contracts had to be discarded due to syntax errors, incomplete function definitions, and errors in Slither compilation. We obtain a dataset of 16,529 functions from the remaining contracts, their opcode sequences, and gas cost estimates from the successfully compiled and processed contracts.

The function opcode sequences are then embedded using the same Word2Vec model mentioned in Section 3.3. The means of the embeddings are calculated to obtain a single vector for each function.

4.3 Results

To analyze the performance of our approach, we compared functions within the dataset itself, trying to find other, more efficient functions and those with the highest similarity. As mentioned, we restricted the search to functions with the same function selector. Additionally, we noticed that, in many cases, functions of one contract inherited by another contract have a higher gas cost when called from the child contract. To avoid these erroneous results, the approach considers only those functions with a lower gas cost estimate that differ in their source codes. Finally, we also remove results where the gas difference is above 95% since that usually means that one of the functions provides only the signature. This case occurs particularly for some contracts obtained from GitHub since, in most cases, those contracts are unverified and undeployed, and the developers leave functions unimplemented for future versions. We have summarized an overview of the results in Table 1.

When we compare the functions in the dataset, we obtain 3,988 comparisons, where more efficient versions of functions could be found. For each function, we find two other functions: One with the highest difference in gas cost and another one with the maximum similarity. The results show a high mean reduction in gas cost when considering functions with the max gas cost difference and functions having the same similarity. The high mean gas reductions of 46.84% and 33.45% root from the fact that we evaluate our approach on a larger real-world dataset of contracts from GitHub. From the results, it is evident that choosing functions with the same signature but with the maximum possible gas reduction can cause a significant decrease in the gas cost of the function under test. However, this approach also suffers from a high standard deviation of 30.22% since the outputs also include functions that may have similar function selectors but are not entirely the same in implementation—functions with reduced functionality and hence the reduced gas cost.

On the other hand, choosing functions with the maximum similarity value leads to a lower mean gas reduction and a lower standard deviation of 24.87%, thus indicating a more stable approach. Selecting functions with a high similarity score ensures that the functions are as similar in implementation as possible and that the more efficient one can be used to improve or replace the less efficient one. We can also see the difference between the two approaches in the modes. While choosing the functions with the maximum

[8] https://github.com/carl-egge/github-solidity-scraper

Table 1: Results of function comparisons

Type	Number of Comparisons	Mean Gas Reduction	Standard Deviation	Mode
Maximum gas cost difference	1,994	46.84%	30.22%	45.11%
Maximum similarity	1,994	33.45%	24.87%	19.12%
Same optimal function for both criteria	1,144	32.47%	26.81%	19.12%

gas difference causes a higher mode gas reduction of around 45%, functions with maximum similarity have a more moderate mode of around 19%. From the examples provided in Section 4.4, we can see that our approach can detect highly gas-efficient patterns capable of causing gas cost reductions of around 50%. Although not commonly found during the comparisons, these patterns cause the high mean gas reduction values, demonstrating our work's potential. Finally, we also analyze the number of comparisons in which the function with the maximum similarity offers the highest percentage of gas cost reduction. Such cases provide a similar mean gas reduction to the maximum similarity cases.

It is important to note that the optimization opportunities presented by the algorithm in this paper are entirely meant as suggestions to the developer and not as automatic replacements for existing code. This is especially important given the more straightforward approach to similarity detection used. Cosine similarity does not detect cases where the more efficient version of a function uses existing or previously declared constructs to reduce gas cost. However, the results can inspire developers to use those constructs to optimize their code.

4.4 Gas-efficient Patterns

Through the approach presented in this paper, we compare the gas costs of the functions' opcode sequences and the patterns in the source code that cause these reductions. This single approach searches for patterns that multiple fixed rules may define. When presented to developers, they will have the option to modify their code based on these optimizations.

We present some unique examples of such patterns here. In the patterns we show, the first function is the unoptimized version, and the second is the optimized version obtained from the dataset. In the comparisons, a red statement indicates a statement that needs to be removed from the less efficient function, and a green statement indicates a statement that needs to be added to arrive at the more efficient function.

(1) **Avoiding cross-contract function calls**: One of the simplest but effective optimization patterns between two contract versions can be seen in Listing 2.
Here, it is evident that removing a function call to a member of another contract or interface and using a locally stored mapping significantly reduces the gas cost. However, it has to be noted here that this optimization utilizes a previously stored mapping to get the length of the trusted accounts of the sender, which indicates that this mapping is defined elsewhere. Storing and retrieving from a mapping is typically much cheaper than a cross-contract function call since a mapping read/write accesses the local state of the contract and does not have to access the storage of and transfer data from another contract. This optimization decreases the cost by 925 gas, a reduction of 50% over the unoptimized function. As mentioned earlier, upon receiving this suggestion, the developer has to accept it and potentially change the architecture of their implementation to incorporate this optimization. Future improvements can include looking at better similarity detection techniques that ensure optimized functions have the same memory and external contract access structure as the function under test.

(2) **Using a cheaper function modifier**: This example shows another gas-efficient pattern that involves a function's definition change. Based on the same function selector but having the highest reduction in gas cost, we get the pattern in Listing 3. In this comparison, changing the modifier of the function from `creatorOnly(_id)` to `onlyOwner` reduces the number of operations to be performed since `creatorOnly(_id)` requires accessing a mapping or array to access and compare the creator for a specific `_id`, where `onlyOwner` performs a single operation to check whether the sender of the message is the owner of the contract. In addition to this, storing and retrieving the parameter `uri` as `calldata` is less gas-intensive. Since our approach returns entire functions, the body of the function is also presented as a suggestion to the developer, although that does not contribute to any significant gas reduction. Simply modifying the function definition causes a decrease of 1200 gas, around 50%. However, in this case, the developer has to decide whether `onlyOwner` satisfies their requirement. Future improvements to the algorithm can benefit from taking more context into account while searching for functions and showing more context to the developer when suggesting optimizations.

(3) **Using function chaining**: Another interesting pattern can be seen in Listing 4. Here, the function body plays a more important role in reducing the gas cost than the differences in the function signature. The optimized function uses function chaining instead of separate function calls, which is more gas-efficient. Function chaining requires less memory allocation since intermediate results can be utilized immediately without storing in the stack. On the other hand, each nested function call creates a new stack frame that increases the gas cost. Second, the optimized function converts function parameters to `Safe` data types instead of calling the `Safe` versions of the functions, which require additional checks. Finally, the efficient function uses direct division to divide by a constant, which is more gas efficient than `safeDiv()` with a large constant. These optimizations lead to a cost reduction of 211 gas or 9.84%.

Listing 2: Avoiding cross-contract function calls

```
function getTrustedCount() external view returns (uint
    ) {
return acuityTrustedAccounts.getTrustedCount();
  }

function getTrustedCount() external view returns (uint
    ) {
return accountTrustedAccountList[msg.sender].length;
}
```

Listing 3: Using a cheaper function modifier

```
function setURI(uint256 _id, string _uri) external
    creatorOnly(_id) {
emit URI(_id, _uri);
}

function setURI(uint256 id, string calldata uri)
    external onlyOwner {
_uris[id] = uri;
emit URI(uri, id);
}
```

Listing 4: Using function chaining

```
function accumulativeDividendOf(address _owner) public
     view returns(uint256) {
return safeDiv(toUint256(safeAdd(toInt256(safeMul(
    magnifiedDividendPerShare, balances[_owner])),
    magnifiedDividendCorrections[_owner])),
340282366920938463463374607431768211456);
}

function accumulativeDividendOf(address _owner) public
     view override returns(uint256) {
return magnifiedDividendPerShare.mul(balanceOf(_owner)
    ).toInt256Safe()
.add(magnifiedDividendCorrections[_owner]).
    toUint256Safe() / magnitude;
}
```

Listing 5: Casting an address to a contract

```
function sendRandomNumber(address contractAddress)
    external {
  uint256 num = 15;
  bytes32 random = bytes32(abi.encodePacked(num));
IRaffleFi(contractAddress).randomizerCallback(
    requestId, random);
  }

function sendRandomNumber(address contractAddress)
    external {
  uint256 num = 15;
  bytes32 random = bytes32(abi.encodePacked(num));
RaffleFi(contractAddress).randomizerCallback(requestId
    , random);
  }
```

(4) **Casting an address to a contract**: Listing 5 shows a pattern that applies to specific use cases.
The only difference between these functions is that the more efficient function directly calls a function on the `RaffleFi` contract instead of first casting the contract's address to the interface `IRaffleFi`. Casting to an interface is beneficial in most cases. It can lead to gas savings since the compiler performs type checking, and function implementations need not be stored in the bytecode. However, in the case of more straightforward function calls, as in this example, it can lead to unnecessary overheads that cause elevated gas cost. If the developer is already aware of the contract structure they are accessing, calling a function on that contract directly removes the extra overhead of type checking, function lookup, and memory expansion required to store the function arguments during runtime. This modification saves 30 gas, a reduction of around 11%. This optimization was done between two versions of the same contract from GitHub. This example also shows that analyzing the opcode sequences can bring out optimizations in specific contexts that may not always apply to the general case.

These gas-efficient pattern examples show that this approach can search for multiple optimizing patterns for functions without being restricted to specific rules. Notably, the examples mentioned were the easiest for us to identify.

5 Related Work

To the best of our knowledge, there is only limited work in code mining for smart contracts, most of which focuses on vulnerability detection based on code similarity. Nevertheless, a number of related approaches to gas optimization are important to the work at hand.

Brandstätter et al. [9] explored optimization strategies to implement a rule-based Solidity source code optimizer. The authors conclude that automatic rule detection and optimization need to be extended. Chen et al. have published multiple papers regarding the gas cost optimization of smart contracts [12–14, 27]. Starting in 2017, they were among the first to address the problem of gas optimization. They identified seven anti-patterns in Solidity bytecode and presented the tool GASPER to identify three gas-costly patterns [14]. In 2018, they presented an in-depth investigation and the GasReducer tool, which can automatically check bytecode and run bytecode-to-bytecode optimization for 24 under-optimized code patterns. In 2020, the tools SODA [12] and GasChecker [13] followed, again dedicated to optimizing bytecode based on predefined rules and patterns.

Nagele et al. presented the SuperOptimizer *ebso*. Using a Satisfiability Modulo Theories (SMT) solver, they developed and tested a tool that can run EVM bytecode optimizations [36]. Albert et al. presented EthIR, "a framework for analyzing Ethereum bytecode" [3]. EthIR is an extension of OYENTE published by Luu et al. [31]. OYENTE is a symbolic execution tool that works with smart contract bytecode and detects bugs for predeployment mitigation. It uses the Z3 theorem solver [15] for symbolic execution of the smart contract code [31]. Albert et al. introduced the Eclipse plugin GASOL[9] that allows the user to investigate and optimize their Solidity code with different cost models [1].

[9]GASOL: https://github.com/costa-group/gasol-optimizer

There has been limited research on automatically deriving optimization patterns from smart contract source code. Existing approaches include static and symbolic execution of the code to find possible optimization chances or refactor the sequence of operations to reduce storage manipulation-related gas cost. One such approach is Syrup [2] by Albert et al., which aims to optimize smart contracts by analyzing the CFG blocks. The tool breaks up each block into sub-blocks based on state-modifying instructions and sends each sub-block to a Max-SMT solver such as Z3 [15] to obtain an optimized version of the sub-block [2]. Feist et al. published Slither, a framework for static analysis of smart contracts. Slither provides information regarding smart contracts' performance, accuracy, and robustness. However, the static analysis focuses less on the gas cost [17]. Li [25] proposed optimization techniques for smart contracts using machine learning and deep learning techniques to estimate the gas costs of specific types of constructs, such as loops and storage manipulation.

Notably, code smell detection and optimization for conventional programs have been a vivid field of research for many years [39], and some of those concepts have trickled down to smart contracts. SmartDagger [28] is a tool that uses static analysis of smart contract bytecode to detect cross-contract vulnerability but does not consider gas optimization. Manticore [35] is a tool that uses dynamic symbolic execution to test and detect flaws in smart contract source code. Existing program verification frameworks such as Boogie or LLVM can perform formal verification of smart contracts [44]. Schneidewind et al. [41] review existing approaches to static analysis of smart contracts and introduce their tool eThor.

Many of these approaches also use machine learning to automate vulnerability detection based on code similarity. Among the most recent works, Xie et al. [26] derive learnable features from the bytecode of compiled contracts to predict vulnerable or malicious behavior. Liu [30] introduced techniques to derive specification models—formal representations of the behavior and interaction of smart contracts. This work infers *Role Models, Automata Models and Invariant Models* [30] from a smart contract, that can be used to analyze that contract and detect bugs and vulnerabilities, and debug that contract. Predicting the behavior of a smart contract from its transaction history can also prove helpful in optimizing it. Gao [19] introduced a tool called SmartEmbed to generate learned embeddings of smart contracts that can be used to detect clone-based bugs. Such tools can also be used to improve the similarity detection aspect of our approach.

From the discussion in this section, it is evident that the majority of the work in the area of smart contract optimization has focused on utilizing predefined rules or constructs to make optimizing changes to smart contracts. The automated approaches, including those that use machine learning, have focused predominantly on vulnerability detection. None of these approaches aim to automate the optimization of smart contracts or perform any operation based on gas cost estimation. Our approach provides a technique to fill that void.

6 Conclusion

In this paper, we have presented an approach to using code mining to search for optimized versions of functions in a smart contract. The contract files are automatically organized and compiled to obtain opcode sequences. The function source codes are mapped to their opcode sequences by analyzing the CFG of the contract. A pretrained Word2Vec model is used to generate embeddings of the opcode sequences, which are then used to compute the similarity between two functions. Cosine similarity is used as the similarity measure and is the most straightforward measure to use in this scenario. This restricts the search, in our case, to functions with the same function selector. This leads to the highest mean gas cost reduction of 46.84%, considering functions with the highest gas cost difference and the same function selector. The principal advantage of this approach lies in the fact that optimization patterns are not limited to specific patterns predetermined by the developer. Given a sufficiently large and diverse code dataset, this approach can also discover and suggest patterns previously unknown to the developer.

This approach can be further developed by incorporating a more accurate similarity measure and removing the restriction on the function selector. Comparing functions with different selectors will elicit even more optimizing patterns. Versions of the approaches introduced in [43] and [8] can be used. Furthermore, similarity detection should consider more context and present more context to the developer to avoid substantial changes in the developer's code. In this paper, we have given some examples of the optimizing patterns mined from the dataset. In the future, we will also provide a more in-depth analysis of mined cost optimization patterns in a separate paper.

Acknowledgment

We gratefully acknowledge the financial support of the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology, and Development, and the Christian Doppler Research Association.

References

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, 118–125. https://doi.org/10.1007/978-3-030-45237-7_7

[2] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria Anna Schett. 2022. Super-optimization of Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 70:1–70:29. https://doi.org/10.1145/3506800

[3] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *16th International Symposium on Automated Technology for Verification and Analysis (LNCS, Vol. 11138)*. Springer, 513–520. https://doi.org/10.1007/978-3-030-01090-4_30

[4] Maher Alharby, Amjad Aldweesh, and Aad van Moorsel. 2018. Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research (2018). In *2018 International Conference on Cloud Computing, Big Data and Blockchain*. IEEE, 1–6. https://doi.org/10.1109/ICCBB.2018.8756390

[5] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *10th Working Conference on Mining Software Repositories*. IEEE, 207–216. https://doi.org/10.1109/MSR.2013.6624029

[6] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*. ACM, 1–19. https://doi.org/10.1145/800028.808479

[7] Amirreza Bagheri and Péter Hegedüs. 2021. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *14th International Conference on Quality of Information and Communications Technology (CCIS, Vol. 1439)*. Springer, 267–281. https://doi.org/10.1007/978-3-030-85347-1_20

[8] Avik Banerjee, Carl Egge, and Stefan Schulte. 2024. Towards the Optimization of Gas Usage of Solidity Smart Contracts with Code Mining. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 365–367.

https://doi.org/10.1109/ICBC59979.2024.10634345

[9] Tamara Brandstätter, Stefan Schulte, Jürgen Cito, and Michael Borkowski. 2020. Characterizing Efficiency Optimizations in Solidity Smart Contracts. In *2020 IEEE International Conference on Blockchain*. IEEE, 281–290. https://doi.org/10.1109/Blockchain50366.2020.00042

[10] Vitalik Buterin. 2014. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* Retrieved 2024-09-25 from https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[11] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. 2019. *EIP-1559: Fee market change for ETH 1.0 chain.* Retrieved 2024-09-25 from https://eips.ethereum.org/EIPS/eip-1559

[12] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *27th Annual Network and Distributed System Security Symposium*. The Internet Society.

[13] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Trans. Emerg. Top. Comput.* 9, 3 (2021), 1433–1448. https://doi.org/10.1109/TETC.2020.2979019

[14] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 442–446. https://doi.org/10.1109/SANER.2017.7884650

[15] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[16] Carl Egge. 2022. *A Repository for Solidity Smart Contracts.* Bachelor's Thesis. Hamburg University of Technology.

[17] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE / ACM, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[18] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, Vincent S. Tseng, and Philip S. Yu. 2021. A Survey of Utility-Oriented Pattern Mining. *IEEE Trans. Knowl. Data Eng.* 33, 4 (2021), 1306–1327. https://doi.org/10.1109/TKDE.2019.2942594

[19] Zhipeng Gao. 2021. When deep learning meets smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 1400–1402. https://doi.org/10.1145/3324884.3418918

[20] A. B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

[21] Richard Kennard and John Leaney. 2012. An Introduction to Software Mining. In *Eleventh International Conference on New Trends in Software Methodologies, Tools and Techniques (Frontiers in Artificial Intelligence and Applications, Vol. 246)*. IOS Press, 312–323. https://doi.org/10.3233/978-1-61499-125-0-312

[22] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira Guégan, Elhadj Benkhelifa, and Anoud Bani-Hani. 2021. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Netw. Appl.* 14, 5 (2021), 2901–2925. https://doi.org/10.1007/S12083-021-01127-0

[23] Shaheen Khatoon, Azhar Mahmood, and Guohui Li. 2011. An evaluation of source code mining techniques. In *Eighth International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE, 1929–1933. https://doi.org/10.1109/FSKD.2011.6019877

[24] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, Michael Galler, and Morris Bernstein. 1996. Pattern Matching for Clone and Concept Detection. *Autom. Softw. Eng.* 3, 1/2 (1996), 77–108. https://doi.org/10.1007/BF00126960

[25] Chunmiao Li. 2021. Gas Estimation and Optimization for Smart Contracts on Ethereum. In *36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1082–1086. https://doi.org/10.1109/ASE51524.2021.9678932

[26] Tao Li, Haolong Wang, Yaozheng Fang, Zhaolong Jian, Zichun Wang, and Xueshuo Xie. 2023. Block-gram: Mining Knowledgeable Features for Smart Contract Vulnerability Detection. In *7th International Conference on Smart Computing and Communication*. Springer, 546–557. https://doi.org/10.1007/978-3-031-28124-2_52

[27] Xiaoqi Li, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, and Zhou Xu. 2020. STAN: Towards Describing Bytecodes of Smart Contract. In *20th IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 273–284. https://doi.org/10.1109/QRS51102.2020.00045

[28] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 752–764. https://doi.org/10.1145/3533767.3534222

[29] Xinxin Liu and Scott A. Smolka. 1998. Simple Linear-Time Algorithms for Minimal Fixed Points (Extended Abstract). In *25th International Colloquium on Automata, Languages and Programming (LNCS, Vol. 1443)*. Springer, 53–66. https://doi.org/10.1007/BFB0055040

[30] Ye Liu. 2022. A Unified Specification Mining Framework for Smart Contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 190:1–190:3. https://doi.org/10.1145/3551349.3559512

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269. https://doi.org/10.1145/2976749.2978309

[32] Dharmesh M. Maniyar and Ian T. Nabney. 2006. Visual data mining using principled projection algorithms and information visualization techniques. In *Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 643–648. https://doi.org/10.1145/1150402.1150481

[33] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations*.

[34] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119.

[35] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[36] Julian Nagele and Maria Anna Schett. 2020. Blockchain Superoptimizer. *CoRR* abs/2005.05912 (2020). arXiv:2005.05912 https://arxiv.org/abs/2005.05912

[37] Satoshi Nakamoto. 2009. *Bitcoin: A Peer-to-Peer Electronic Cash System.* Retrieved 2024-09-25 from https://bitcoin.org/bitcoin.pdf

[38] Markus Nissl, Emanuel Sallinger, Stefan Schulte, and Michael Borkowski. 2021. Towards Cross-Blockchain Smart Contracts. In *IEEE International Conference on Decentralized Applications and Infrastructures*. IEEE, 85–94. https://doi.org/10.1109/DAPPS52256.2021.00015

[39] Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques. *J. Softw. Evol. Process.* 27, 11 (2015), 867–895. https://doi.org/10.1002/SMR.1737

[40] Dimitri Saingre. 2021. *Understanding the energy consumption of blockchain technologies : a focus on smart contracts. (Comprendre la consommation énergétique des blockchains : un regard sur les contrats intelligents).* Ph. D. Dissertation. IMT Atlantique Bretagne Pays de la Loire, Brest, France.

[41] Clara Schneidewind, Markus Scherer, and Matteo Maffei. 2020. The Good, The Bad and The Ugly: Pitfalls and Best Practices in Automated Sound Static Analysis of Ethereum Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Springer International Publishing, 212–231.

[42] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*. OpenReview.net.

[43] Zhenzhou Tian, Yaqian Huang, Jie Tian, Zhongmin Wang, Yanping Chen, and Lingwei Chen. 2022. Ethereum Smart Contract Representation Learning for Robust Bytecode-Level Similarity Detection. In *The 34th International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc., 513–518. https://doi.org/10.18293/SEKE2022-040

[44] Palina Tolmach, Yi Li, Shangwei Lin, Yang Liu, and Zengxiang Li. 2022. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7 (2022), 148:1–148:38. https://doi.org/10.1145/3464421

[45] Florian Tschorsch and Björn Scheuermann. 2016. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Commun. Surv. Tutorials* 18, 3 (2016), 2084–2123. https://doi.org/10.1109/COMST.2016.2535718

[46] Bangrui Wan, Shuang Dong, Jianjun Zhou, and Ying Qian. 2023. SJBCD: A Java Code Clone Detection Method Based on Bytecode Using Siamese Neural Network. *Applied Sciences* 13, 17 (2023). https://doi.org/10.3390/app13179580

[47] Gavin Wood. 2024. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* Retrieved 2024-09-25 from https://ethereum.github.io/yellowpaper/paper.pdf

[48] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. SolidityCheck : Quickly Detecting Smart Contract Problems Through Regular Expressions. *CoRR* abs/1911.09425 (2019). arXiv:1911.09425 http://arxiv.org/abs/1911.09425