IEEE ICBC'24 1570982133

Towards the Optimization of Gas Usage of Solidity Smart Contracts with Code Mining

***Abstract*—Second-generation blockchains, such as Ethereum, allow users to execute smart contracts, which are distributed applications executing user-defined logic. Usually, blockchains charge gas fees for the deployment and invocation of smart contracts. These costs can be significant and even render some use cases non-economical. Therefore, the optimization of smart contracts with regard to gas costs is an important achievement, and a number of according approaches have already been presented. However, existing methods of gas optimization are often based on rule-based code optimization techniques which can perform only a subset of possible optimizations and cannot detect outlying and uncommon code patterns.**

Therefore, this paper introduces a method, using machine learning, to detect a more optimized version of a solidity smart contract. This approach trains a Siamese neural network to detect the similarity between a contract and its optimized version and hence provides the basis for informing the user about existing optimizing patterns. We evaluate our approach using a repository of 30,432 Solidity smart contracts and their optimized versions generated by the Solidity compiler's opcode-based optimizer which offers an average gas cost reduction of 36.89%.

***Index Terms*—blockchain, smart contracts, solidity, siamese neural network, code-mining**

I. Introduction

Since the introduction of blockchain technologies in 2008 [1], the idea to build a completely decentralized Web without controlling authorities has gained popularity, leading to the development of applications such as cryptocurrencies and non-fungible tokens [2].

First-generation blockchains (e.g., Bitcoin, Litecoin) essentially provide a distributed ledger consisting of append-only blocks, where the trusted central authority is replaced by a network of nodes that share compute power [3]. The second generation of blockchains was proposed in 2014 in the Ethereum white paper [4], introducing the concept of smart contracts—user-defined pieces of code that reside on the blockchain and are invoked when their accounts are transacted with. Smart contracts are executed in an environment provided by the respective blockchain that is responsible for executing the instructions in the smart contract and taking care of the resulting changes. The first and still the most important example of such an environment is the Ethereum Virtual Machine (EVM) which is responsible for defining and enforcing the rules that govern the computation of a new valid state after the inclusion of a block in Ethereum. The EVM maintains the Ethereum network as a *distributed state machine*, consisting of a state and an associated set of transactions [5].

As Ethereum and similar second-generation blockchains grow in popularity, the number of deployed smart contracts is growing in number. The defining feature of a blockchain is immutability, which makes it nearly impossible to alter smart contract code that has already been deployed. This raises the need to develop smart contract code that is error-free and efficient before deploying it on the blockchain [6]. Deployment and invocation of a smart contract on the blockchain leads to gas costs that are charged to the developer who deploys the smart contract and subsequently to the user who invokes the contract through a transaction [4]. This causes each smart contract to have a gas cost associated with it. The gas consumption can, in turn, also estimate the actual energy consumption of the smart contract, since the gas costs are intended to be proportional to the computation required for the contract operations [7]. Thus, it is important to deploy contracts that are error-free and efficient to reduce the costs to the user, as well as to optimize energy consumption.

Multiple approaches can be adopted towards optimizing source code, from modifying code based on pre-defined rules to automatically mining patterns from repositories of existing source code. Code can be optimized, not only for efficiency but also to remove code smells and to improve maintainability, re-usability, comprehensibility, and extensibility [8].

Code mining approaches involve detecting patterns from a repository of source files and analyzing the purpose those patterns serve. This provides the benefit of not defining those optimizing patterns manually and also of detecting outlying and innovative patterns from existing code. The summarization and visualization of algorithms using code mining can also be extended to search for optimizing patterns in smart contracts. Training of such code-mining approaches requires the availability of a large dataset.

The principal difference between optimizing conventional programs and smart contracts lies in the fact that smart contracts have to be optimized primarily in terms of gas cost [9], while optimization of conventional programs usually aims at resource or time efficiency. However, smart contracts have deployment and invocation gas costs associated with them and they are dependent on the sequence of operations performed when executing the contract. Optimizing smart contract code directly implies optimizing the gas cost. This introduces an additional constraint and hence requires techniques for constraint-optimized pattern recognition [10].

The existing approaches to smart contract optimization rely primarily on detecting pre-defined patterns that are known to have the potential for optimization. This largely narrows down the scope of that operation and also precludes patterns not commonly used or not known to the developer. Our work

aims to provide a foundation towards utilising code-mining for automated detection of such optimizing patterns from a source-code repository. The approach draws from the technique for smart contract similarity detection introduced in [11] using a Siamese neural network.

Our work mainly focuses on smart contracts written in Solidity since Solidity is the largest programming language with the most number of contracts on the Ethereum blockchain [12], and also because it is possible to collect a large dataset of verified contract source codes written in Solidity and also deployed on Ethereum.

The remainder of this paper is structured as follows: In Section II, we define the preliminary concepts for our approach. Afterwards, Section III elaborates on the motivation behind this work. Section IV gives an overview of the methodology followed to achieve the aims of this work. Section V describes the implementation details. Section VI speaks about the scope of improvement of this work. Section VII mentions existing work in this field and finally Section VIII concludes the discussion of this paper.

II. Preliminaries

A. Blockchain

A public blockchain is a tamper-proof distributed ledger, which is similar to a distributed storage of transactions [13] that represent interactions between users. Every transaction is registered in the ledger and the ledger is accessible to all users, thereby making it transparent. The ledger is a chain of immutable blocks which are made tamper-proof using cryptographic techniques. According to Dannen [13], a blockchain is a combination of the key concepts of peer-to-peer networking, asymmetric cryptography, and cryptographic hashing. Each block in a chain contains a set of transactions along with the hash of the previous block, which links blocks together and makes modification very difficult [1]. Every block also contains a nonce that is generated along with the block, by solving a hard cryptographic puzzle. This nonce makes it difficult for malicious parties to add blocks to the chain. Transactions are combined into blocks by *miners*.

B. Ethereum

While the Bitcoin paper introduced the concept of a blockchain along with a use case in the form of the cryptocurrency Bitcoin, the evolution of blockchains continued after that. In 2013, Vitalik Buterin published the Ethereum White Paper [4], which introduced the second generation of blockchains. The Bitcoin protocol essentially defines the procedure to handle transactions between accounts. Ethereum introduced the ability to run Turing-complete programs on the blockchain, in the form of smart contracts.

With Ethereum, it is theoretically possible to implement and run any kind of service on the blockchain network. In principle, a developer can use any high-level programming language to create an application, compile it to bytecode and deploy it 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 Ethereum, the name smart ccntracts was established for these programs.

C. Smart Contracts

In the original Ethereum paper, Buterin described smart contracts as "systems which automatically move digital assets according to arbitrary pre-specified rules" [4]. A smart contract is a program, that resides on the blockchain and is executed every time a user interacts with it.

Every smart contract holds pre-defined functions that can be invoked once it is deployed on the blockchain. For instance, a smart contract is created using the constructor method. One can create a contract and take ownership of it by sending a transaction to the constructor method [5], [14]. A self-destruct or check-balance function can be generated and used after deployment similarly. Every smart contract has a distinct 20-byte address, is capable of holding an Ether balance, and may respond to incoming transactions using its methods [15]. It can also be distinguished between deterministic and non-deterministic smart contracts, where deterministic smart contracts do not require any information from outside the blockchain and non-deterministic smart contracts need to access external information (e.g., weather data) [16]. Users create and interact with the smart contracts, which are then stored in the blocks on the blockchain. Each smart contract can store code and data that is accessed from the blockchain. The compute power for running the smart contracts is provided by the network itself and the miner nodes that try to append new blocks.

A smart contract can be written in various high-level languages but is compiled into bytecode when deployed on the blockchain. For example, Ethereum introduces the programming language Solidity, which can be used to write smart contracts, that are then run on the EVM. The EVM is a quasi-Turing complete machine [5] in that it restricts the over-utilization of resources by limiting the *gas cost*. Each transaction on the Ethereum network has an associated gas consumption, which has to be paid for by the initiator of the transaction. A smart contract is essentially a collection of operations that incur gas costs [5]. Hence the deployment and invocations of a smart contract have gas consumption associated with them.

The gas consumption of every transaction on the Ethereum chain can be delineated by two properties: `STARTGAS` and `GASPRICE`. `STARTGAS` is the limit of computational steps a contract can execute before it runs out of gas. The `GASPRICE` is the incentive to miners per computational step that they carry out. This limitation of gas is essential to prevent smart contracts from going into infinite loops and exponential runtime, which can use up node resources and hence disrupt the entire network [4]. The Ethereum whitepaper [4] lists the gas consumption of individual op-codes executed by the EVM. Reducing the number of operations and replacing expensive operations with cheaper ones can reduce the gas cost and hence, indirectly, the actual energy consumption of a smart contract [7].

D. Code-Mining

Code-mining is the process of finding code patterns by analyzing the large corpus of source code available in software repositories [17]. Such patterns depend on the type of end-analysis desired and can point to software development best practices, duplicated vulnerable code, or optimizing constructs, among others [18].

Code-mining describes the process of automatically detecting patterns from source code. This is usually done with source code in a high-level programming language, but it can also be done with bytecode. Good programming follows certain design principles that can be specific to the language that is used and the application that is built, but at the same time general rules apply to nearly all software projects. Violations of these design principles are called code smells [8]. One of the main purposes of code mining is to automatically detect and possibly remove code smells. Different methods can be used for code mining, depending on the optimizations that should be achieved.

As the name suggests, code-mining aims to analyze source code using data mining methods. Khatoon et al. describe three mining techniques that are 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 [19]. In all of these areas, code mining techniques can be used to analyze the source code and improve its quality by finding code smells. This paper aims to provide the first steps towards a clone detection and optimization system directed specifically towards smart contracts. The system aims to detect functionally similar smart contracts and choose the more optimal one from among them. For most of the contracts deployed on the Ethereum network today, the source code is not readily available. What is available, are the bytecodes and the sequence of op-codes that can be derived from them. This provides us with a much larger dataset of contracts that we can analyze and extract optimizing patterns from. This is why we selected an approach for comparing smart contracts that makes use of the bytecode and not the source codes of contracts.

III. Motivation

The principal motivation behind this work is the automation of smart contract optimization. The majority of work in this field has focused mainly on fixed rule-based optimization or using machine learning on specific constructs such as loops. These attempts, however, preclude the automated detection of optimizing patterns from a corpus of smart contracts. This paper presents an approach to detect similar contracts and then find optimal ones among them, which is, to our knowledge, the first attempt at providing the basis for a code-mining approach towards the aim of automated optimization of smart contracts.

In such an approach, the first requirement that needs to be satisfied is that of the detection of contract similarity. Over the years, there have been multiple attempts to define metrics that determine if two pieces of code are similar, as in [20] and [21], among others. However, training and evaluating such systems requires a sufficiently large dataset of mainly source codes, which are hard to come by for smart contracts deployed on a blockchain. What is more readily available, are the bytecodes and correspondingly, the sequence of opcodes that is generated by the EVM from solidity source files. Given this fact, we decided to use the approach introduced by Tian et al. [11] to detect the similarity of two smart contracts by encoding their opcode sequences and then comparing them using a similarity measure. This approach has already demonstrated robustness and effectiveness in detecting similar contracts.

IV. Methodology

A. Data Collection

Each smart contract on Ethereum can be composed of one or more individual smart contracts, which may be imported from pre-existing, standard contracts or can be ones written by the same authors. The Solidity compiler generates bytecode and opcode-sequence for each individual contract in the source code. These opcode sequences and bytecodes were collected and separated using a recursive lookup of the compiler output, thereby producing sets of contracts that cannot be divided further.

Each opcode sequence thus obtained contains a series of EVM opcode mnemonics, each followed by zero or more arguments, as seen in Listing IV-A. These arguments do not contribute to the understanding of the overall functionality of the smart contract and can hence be ignored. The final opcode sequence that is further used, is composed of a list of opcode mnemonics, without the arguments, as in Listing IV-A.

Opcode sequence for a contract

```
PUSH1 0x80
PUSH1 0x40
MSTORE
CALLVALUE
DUP1
ISZERO
PUSH3 0x11
```

Opcode sequence without arguments

```
[PUSH1,
PUSH1,
MSTORE,
CALLVALUE,
DUP1,
ISZERO,
PUSH3]
```

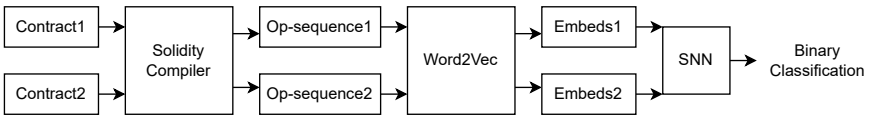

Fig. 1. An overview of the system

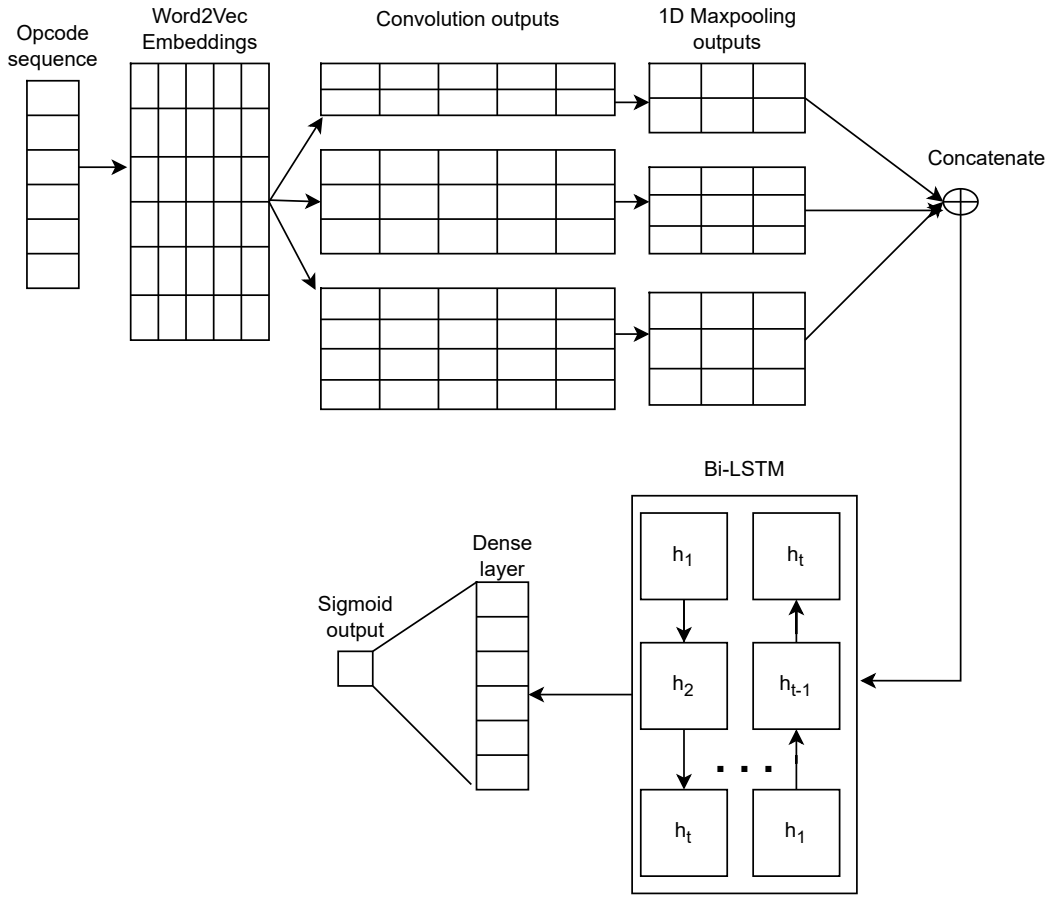

Fig. 2. An overview of the neural network architecture

B. Initial Embeddings

Once we have the sequences of opcodes, we need to convert them to a form that can be understood by a machine learning model. The standard way to do this is to convert each opcode sequence to a sequence of numerical vectors, on which the machine learning model can perform matrix transformations.

To achieve this, we make use of the Word2Vec model, as suggested by Tian et al. [11]. Word2Vec is a technique introduced by Mikolov et al. in 2013 [22] [23], with the aim of learning meaningful and contextual word representations. This technique utilizes a neural network to learn embeddings of words from an existing corpus [22] and can hence be trained to generate 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 [22], which predicts a window of past and future words based on the current word.

After training, the opcodes in the sequences are replaced by the corresponding embeddings to obtain a set of matrices, each of size $n \times k$ where n is the number of opcodes in the sequence and k is the length of the embeddings produced by the Word2Vec model.

C. Machine Learning Model

The machine learning model used to determine the similarity of the smart contracts is prepared according to the suggestions of [11]. The model consists of a Siamese neural network, with each network made of a layer of convolutional filters of different sizes followed by a Bi-directional LSTM layer. The LSTM layer is necessary to generate an embedding for a sequence of opcodes, while the convolutional layer encodes contextual information surrounding each opcode. The

kernels of multiple sizes ensure interactions between opcode sequences of multiple sizes are recorded [11].

The Word2Vec model converts each opcode sequence into a matrix $\mathbf{M}^{n \times k}$ where n is the number of opcodes in the sequence and k is the length of each embedding. The convolutional layer applies m kernels of shape $l \times k$ to generate an output matrix of shape $(n - l + 1) \times m$.

The outputs of the convolutional filters are then fed to a maxpooling layer, again with kernels of multiple sizes that further downsample the matrices. The max-pooling layers ensure that only the most significant elements in the sequences are captured and passed on to further layers. Having filters of multiple sizes for both the convolution operation and the max pooling operation ensures that context windows of multiple sizes are taken into account and interactions among opcode sequences of various lengths are encoded.

The output feature matrix from the convolutional layers is then input to the bidirectional LSTM layer which is responsible for generating a latent vector representation for the entire contract. The bidirectional LSTM layer scans the sequence from both directions and provides additional context from the sequence. It ensures that context from the entire sequence is taken into account at each timestep. The output of the last timestep acts as the representation of the entire sequence. An overview of the architecture of the neural network can be found in Fig. IV-B

This model architecture is deployed in the form of a Siamese neural network (SNN), which uses the same weights to generate embeddings for a pair of smart contracts provided as input. The embeddings thus produced are then compared using a simple neural network in the form of a binary classification task to determine if they are similar or not.

Having the classifying network as part of the training process enables the Siamese network to learn the weights that produce meaningful embeddings of the contracts provided as input, through backpropagation [11].

The Siamese networks in this case receive the embedded opcode sequences of the pairs of unoptimized and optimized contracts and produce latent embeddings that are then classified by the dense network as being similar or dissimilar. The training of this network on a dataset of optimized and unoptimized sequences of opcodes ensures that the SNN can identify sequences of opcodes for the same contract even with significant differences in the sequence of opcodes.

D. Checking Contracts for Optimized Versions

The trained machine learning model can now be used to check for optimized versions of a contract from a dataset of existing contracts. Given the large number of contracts collected and also the fact that one contract is divided up into multiple dependent contracts, it was difficult to get the creation gas cost of the contracts by deploying them on a test network. This is primarily due to the fact that contracts have varying constructor argument requirements and the arguments that can be obtained from the Etherscan API are applicable for the entire contract and do not account for their dependencies.

Hence, the gas cost for each contract is estimated by adding up the cost of executing each opcode in its sequence. This is done based on a collection of opcodes and their corresponding gas costs, prepared and shared publicly by Imapp with their Gas Cost Estimator project[1].

The dataset prepared after estimating the deployment gas costs contains the name and the address of each contract along with the optimized and unoptimized versions of their opcode sequences, and the estimated gas consumptions. New incoming contracts can be embedded using Word2Vec and the SNN can be used to detect similarities with the existing contracts in the database. If a similar contract has a lower estimated gas consumption than the incoming contract, a more efficient contract can be suggested to the developer.

Since the dataset, prepared in this form, includes the constituent contracts individually and also their dependencies, the suggestions for more optimized versions will be granular and will offer the developer optimized suggestions for individual parts of their smart contract.

V. IMPLEMENTATION

An overview of the entire system can be seen in Figure IV-B.

The dataset of smart contracts used to train the machine learning model was obtained from the top verified smart contracts offered under an open source license by Etherscan[2]. Although a much larger number of contracts are available from Ethereum, through the publicly available dataset on Google BigQuery, only the dataset provided by Etherscan contains verified source code. Since we wanted to create a dataset containing optimized and unoptimized pairs of smart contracts, the source codes were necessary.

The process starts with retrieving contracts using the Etherscan API. Etherscan provides a list of the top 5000 verified contracts deployed on the Ethereum mainnet. The contracts are provided as a CSV file containing, for each smart contract, the transaction hash of the contract creation transaction, the address, and the name of the contract. The Etherscan API[3] allows the user to retrieve the source code of the contract, along with some metadata from the Etherscan database, using the contract address and the user's API key. The responses to the API calls were parsed for the source code and the compiler version used to compile those contracts prior to deployment to the Ethereum blockchain.

The retrieved source codes are then compiled using the Solidity compiler version mentioned in the metadata of each smart contract. The compilation process eliminates a small subset of contracts that have invalid compiler versions in the response and generates 15,216 pairs of unoptimized-optimized bytecodes and opcode sequences. Since the training process requires the machine learning model to learn from both positive and negative samples, we prepare a negative dataset by randomly pairing compiled

[1]https://github.com/imapp-pl/gas-cost-estimator

[2]https://etherscan.io

[3]https://api.etherscan.io/api

TABLE I
COMPARISON OF OPTIMIZED AND UNOPTIMIZED CONTRACTS

	Unoptimized Contract		Optimized Contract	
Contract Name	**Number of opcodes**	**Estimated gas cost**	**Number of opcodes**	**Estimated gas cost**
DEJIZARUZATURA	9702	30743	6098	18905
Ownable	667	2118	395	1171
ERC20	3040	9559	1828	5722
DogpadArmy	7172	22547	4482	13792
BABYKAIK	12720	40290	8191	25409

opcode sequences belonging to different contracts. To make sure that the model can also distinguish between optimized and unoptimized versions of different contracts, the negatives dataset also contains unoptimized opcode sequences paired with optimized ones from different contracts. The final dataset has an equal number of positives and negatives and contains 30,432 data points of the format `<Contract1_address, Contract1_name, Contract2_address, Contract2_name, Opcode_sequence1, Opcode_sequence2>`. The positive and negative datasets are combined and shuffled randomly.

The entire dataset is then used to train the Word2Vec model. We use the Continuous Skip-gram architecture with a window size of 2 and output vector size of 300 to generate embeddings for the opcodes, thus ensuring that the embeddings encode sufficient information, while also keeping the resulting file size within manageable limits. The model is trained for 30 epochs and converts each opcode sequence into a matrix $\mathbf{M}^{n\times 300}$, where n is the number of opcodes in the sequence. This forms the input to the machine learning model.

Since the opcode sequences are of varying lengths, they are padded to the maximum length in the dataset with zero values. This prepares each opcode sequence as a matrix $\mathbf{M*}^{\mathtt{MAX\_LEN}\times 300}$.

The prepared dataset is then input to the machine learning model which consists of a convolutional layer containing 5 kernels each of sizes 2×300, 3×300 and 4×300. These kernels with varying sizes of receptive fields process the interaction between 2, 3 and 4 opcodes at a time and generate a single value encoding each of those interactions. The outputs of the convolutional kernels are then downsampled using a max-pooling layer with kernels of size 5.

The outputs of the max-pooling filters are then concatenated and provided as input to the bidirectional LSTM (Bi-LSTM) layer with 2 features in the hidden state. The bidirectional LSTM layer, in its output, contains a concatenation of the final forward and reverse hidden states and presents a latent representation of the entire contract opcode sequence.

The outputs of the Bi-LSTMs are then concatenated and passed through a dense network with a single neuron in the output layer with a sigmoid activation function. The output of this network performs a binary classification to determine whether the two input contracts are similar or not. A threshold value of 0.5 is applied on the output of this network.

The entire network is trained on the dataset of contracts previously prepared. The dataset is randomly divided into training, validation and test sets in a 80%, 10% and 10% proportion. However, due to constraints on memory and storage requirements, the entire dataset could not be used for training all at once. The model was trained using a VM running on Amazon Web Services, supporting 32 CPU cores and 256GB of RAM. For the purpose of experimentation, the dataset was trained with 8000, 12000 and 16000 samples for 20 epochs with final test F1-scores of 95% and 96% and 97% respectively. The Adam optimizer [24] was used to optimize based on the binary cross-entropy loss function. The inference performance of the network improves with the number of samples used for training, as can be seen with the improving F1 scores on the test set.

The final part of this system involves utilizing the trained model for identifying similar and optimized contracts. This involves augmenting the existing dataset with the gas costs for the individual contracts. As explained in Section IV-D, this is prepared using the list of gas costs of individual opcodes provided by the Gas Cost Estimator project. This procedure serves to suggest optimized versions of the complete contracts. For experimentation, the dataset was prepared with the existing set of contracts retrieved from Etherscan, with the compiler-optimized opcode sequences being the optimized versions. Hence this dataset can only provide source codes for the versions of the contracts already available on Etherscan. Table I shows a comparison of the lengths of the opcode sequences and the estimated gas consumption of some of the contracts in the dataset.

Using this dataset of 15,216 contracts, the optimized versions of the contracts, which are prepared by the rule based optimizer of the Solidity compiler, offer an average gas cost saving of 36.89

VI. LIMITATIONS AND FUTURE IMPROVEMENTS

The primary area of improvement of the presented procedure lies in enhancing the diversity of the dataset which is used to train the machine learning model and also the final dataset that can be used to search for optimizations to new contracts. The only optimizations performed in the existing dataset is by the Solidity compiler, which follows pre-defined rules to optimize existing patterns. This work intends to showcase an approach for code-mining-based smart contract optimization and hence a dataset that could be easily and automatically prepared, was selected. However, a more diverse dataset with more real-world optimization patterns ensures better training

of the machine learning model and also better optimizations for the developer, which can be unknown to the Solidity opcode optimizer. The diversity of the dataset can also be improved by adding more smart contract coding languages such as Vyper and enabling optimizations and suggestions for more languages in use.

The machine learning model, as inspired by the work of Tian et al. [11] is shown to work really well, but other architectures of neural networks can be experimented with, particularly for extending the application of the network to detect similarity directly from source codes. This is an important area of future work and improvement since the process of getting the contracts ready for similarity testing is involved and could benefit from simplification if intended for regular use by developers. The opcode base optimizer can, however, after being fine-tuned on an appropriate dataset, be included as part of the compilation and optimization process. However, for that application, it must be ensured that the inference is sufficiently fast and does not cause excessive load on resources.

The granularity at which the system operates is also another important area of improvement. The existing system can only compare entire smart contracts, which is less useful than optimizing individual blocks and methods of a smart contract. To achieve this, the machine learning-based approach can be augmented with classical metric-based similarity detection techniques, such as presented by Kontogiannis et al. [20] to offer more fine-grained optimization suggestions.

Finally, using a properly curated and prepared dataset, a generative machine learning model can be used that can generate optimized code suggestions in real-time based on the developer's existing code and the trained optimization patterns that apply to that code. However, such an application will require a large dataset of optimized code that can cover a large and diverse set of optimizing patterns, on a much more granular level, probably at the level of individual code statements.

VII. Related Work

As has been mentioned before, there is—to the best of our knowledge—only limited work in the field of code mining for smart contracts. Nevertheless, there are a number of related approaches to gas optimization which are important to the work at hand.

Brandstätter et al. [6] 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 [25]–[28]. Starting in 2017, they were one of 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 of these gas-costly patterns [25]. 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 [27] and GasChecker [26] followed, again dedicated to optimize bytecode based on pre-defined rules and patterns.

Nagele et al. presented the SuperOptimizer *ebso*. Using techniques of a Satisfiability Modulo Theories (SMT) solver, they developed and tested a tool that can run EVM bytecode optimizations [29]. Albert et al. presented EthIR, "a framework for analyzing Ethereum bytecode" [30]. EthIR is an extension of OYENTE that was published by Luu et al. [31]. OYENTE is a symbolic execution tool that works with smart contract bytecode and detects bugs for pre-deployment mitigation. It uses the Z3 theorem solver [32] for symbolic execution of the smart contract code [31]. Albert et al. introduced the Eclipse[4] plugin GASOL that allows the user to investigate and optimize their Solidity code with different cost models [33].

The published work discussed till now focuses on the analysis of the compiled bytecode and on detecting similarities with predefined patterns that can indicate vulnerabilities and malicious constructs. There has been limited research on deriving those patterns from source code automatically, which could lead to the automation of existing optimization possibilities while providing opportunities for previously undetected optimization patterns. Such existing approaches tend to utilize static and symbolic execution of the code to find possible chances at optimization or to refactor the sequence of operations to reduce storage manipulation-related gas consumption. One such approach is Syrup [9] by Albert et al., which is aimed at optimizing smart contracts based on an analysis of the blocks in the Control Flow Graph. 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 [32] to obtain an optimized version of the sub-block [9]. Feist et al. published Slither, a framework for static analysis of smart contracts. Slither provides information regarding the performance, accuracy, and robustness of smart contracts. However, the static analysis is less focused on the gas cost [34]. Chunmiao Li proposed optimization techniques for smart contracts using machine learning and deep learning techniques to estimate the gas consumption of specific types of constructs, such as loops and storage manipulation [35].

Notably, code smell detection and code optimization for conventional programs have been a very vivid field of research for many years [8] and some of those concepts have trickled down to smart contracts. SmartDagger [36] is a tool that uses static analysis of smart contract bytecode to detect cross-contract vulnerability, but does not take into account gas optimization. Manticore [37] 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 be used to perform formal verification of smart contracts [38].

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 pre-defined rules or constructs to make optimizing changes to smart contracts. These rules are

[4]Eclipse IDE: https://www.eclipse.org/ide/

applicable on specific flaws in the code and do not take into account alternative ways to perform the optimizations and also ways that can optimize better. None of the work aims at developing a system that can detect optimizing patterns from a corpus of smart contracts without being restricted to specific constructs, such as loops. This work provides an approach to do so, by detecting the similarity between a contract and its optimized version, based on their opcode sequences.

The principal part of this work is inspired by the approach to detecting the similarity of smart contracts presented by Tian et al. in [11]. In this paper, the authors have presented a Siamese neural network with convolutional and Bi-LSTM layers for detecting if two smart contracts are similar, using embeddings of their opcode sequences. In our work, we have re-utilised a version of this approach with a similar architecture for the network, but the network training has been finetuned towards detecting the similarity of a contract and its optimized version. Our work aims to show that this finetuned network can be used on a dataset of smart contracts to detect optimized versions of a new contract. Our work provides a foundation for further applications and finetuning of this approach to include more fine-grained optimizing patterns in the future.

VIII. Conclusion

This paper attempts to provide a first approach towards automated and generalized optimization of smart contracts using code mining. Unlike existing solutions, this approach does not depend on searching for pre-defined patterns that can benefit from optimizing patterns. It focuses on automatically retrieving optimal code from a corpus. The similarity detection approach, fine-tuned for detecting optimized versions of contracts, achieves a high test F1-score of 97% and the dataset of Solidity optimized contracts which the model is trained and tested on can provide a gas consumption reduction of 37% on an average.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2009, white Paper.

[2] M. Nissl, E. Sallinger, S. Schulte, and M. Borkowski, "Towards Cross-Blockchain Smart Contracts," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, 2021, pp. 85–94.

[3] F. Tschorsch and B. Scheuermann, "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[4] V. Buterin, "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform," 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[5] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, no. 2014, 2014.

[6] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing Efficiency Optimizations in Solidity Smart Contracts," in *2020 IEEE International Conference on Blockchain (Blockchain)*, 2020, pp. 281–290.

[7] D. Saingre, "Understanding the energy consumption of blockchain technologies : a focus on smart contracts," Ph.D. dissertation, 2021.

[8] G. Rasool and Z. Arshad, "A Review of Code Smell Mining Techniques," *J. Softw. Evol. Process*, vol. 27, no. 11, p. 867–895, 2015.

[9] E. Albert, P. Gordillo, A. Hernández-Cerezo, A. Rubio, and M. A. Schett, "Super-Optimization of Smart Contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, jul 2022.

[10] W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao, V. S. Tseng, and P. S. Yu, "A Survey of Utility-Oriented Pattern Mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1306–1327, 2021.

[11] Z. Tian, Y. Huang, J. Tian, Z. Wang, Y. Chen, and L. Chen, "Ethereum smart contract representation learning for robust bytecode-level similarity detection," in *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022*, R. Peng, C. E. Pantoja, and P. Kamthan, Eds. KSI Research Inc., 2022, pp. 513–518. [Online]. Available: https://doi.org/10.18293/SEKE2022-040

[12] P. Zhang, F. Xiao, and X. Luo, "SolidityCheck : Quickly Detecting Smart Contract Problems Through Regular Expressions," 2019. [Online]. Available: https://arxiv.org/abs/1911.09425

[13] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1st ed. USA: Apress, 2017.

[14] S. Khan, F. Loukil, C. Ghedira, E. Benkhelifa, and A. Bani-Hani, "Blockchain Smart Contracts: Applications, Challenges, and Future Trends," *Peer-to-Peer Networking and Applications*, vol. 14, pp. 2901–2925, 2021.

[15] M. Alharby, A. Aldweesh, and A. van Moorsel, "Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research (2018)," in *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*, 2018, pp. 1–6.

[16] V. Morabito, *Smart Contracts and Licensing*, 2017, pp. 101–124.

[17] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 207–216.

[18] R. Kennard and J. Leaney, "An introduction to software mining," in *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Eleventh SoMeT '12, Genoa, Italy, September 26 28, 2012*, ser. Frontiers in Artificial Intelligence and Applications, H. Fujita and R. Revetria, Eds., vol. 246. IOS Press, 2012, pp. 312–323. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-125-0-312

[19] S. Khatoon, A. Mahmood, and G. Li, "An evaluation of source code mining techniques," in *2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, vol. 3, 2011, pp. 1929–1933.

[20] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, *Pattern Matching for Clone and Concept Detection*. USA: Kluwer Academic Publishers, 1996, p. 77–108.

[21] M. di Angelo and G. Salzer, "Assessing the similarity of smart contracts by clustering their interfaces," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 1910–1919.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *arXiv e-prints*, p. arXiv:1310.4546, Oct. 2013.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[25] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 442–446.

[26] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1433–1448, 2021.

[27] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "SODA: A Generic Online Detection Framework for Smart Contracts," 2020.

[28] X. Li, T. Chen, X. Luo, T. Zhang, L. Yu, and Z. Xu, "STAN: Towards Describing Bytecodes of Smart Contract," 2020. [Online]. Available: https://arxiv.org/abs/2007.09696

[29] J. Nagele and M. A. Schett, "Blockchain Superoptimizer," 2020. [Online]. Available: https://arxiv.org/abs/2005.05912

[30] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode," in *Automated Technology for Verification and Analysis*. Springer International Publishing, 2018, pp. 513–520.

[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, p. 254–269.
[32] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, pp. 337–340.
[33] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gas Analysis and Optimization for Ethereum Smart Contracts," 2019. [Online]. Available: https://arxiv.org/abs/1912.11929
[34] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.
[35] C. Li, "Gas Estimation and Optimization for Smart Contracts on Ethereum," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1082–1086.
[36] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability," in *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, p. 752–764.
[37] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts," 2019. [Online]. Available: https://arxiv.org/abs/1907.03890
[38] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," 2020. [Online]. Available: https://arxiv.org/abs/2008.02712