A Repository for Solidity Smart Contracts

Carl Egge
Institute for Data Engineering
Hamburg University of Techology
Hamburg, Germany
email address or ORCID

Avik Banerjee
Institute for Data Engineering
Hamburg University of Technology
Hamburg, Germany
email address or ORCID

Stefan Schulte
Institute for Data Engineering
Hamburg University of Technology
Hamburg, Germany
email address or ORCID

Michael Schröder
Institute of Information Systems Engineering
Technische Universität Wien
Vienna, Austria
email address or ORCID

Jürgen Cito
Institute of Information Systems Engineering
Technische Universität Wien
Vienna, Austria
email address or ORCID

***Abstract*—Blockchain technologies are a growing field of research and public interest. Second generation blockchains, like Ethereum, allow users to execute smart contracts, which are distributed applications executing user-defined logic. This not only expands the utility of blockchains, but also provides new opportunities for tools for solving the same issues that arise with conventional programs. Such tools include optimization of code, detecting code smells and vulnerabilites and developing automated code generation models among others. Such developments require the presence of a dataset of similar code which can be annotated and analyzed according to the desired end application. Existing sources of smart contract code include block explorers that provide limited search and retrieval capabilities on the single version of the contract deployed on the blockchain. Hence to aid the progress of research, this paper introduces the *Smart Contract Repository*—a repository of publicly available Solidity smart contracts, complete with multiple versions of the same contract to help the analysis of the effects of incremental changes in source code.**

***Index Terms*—blockchain, smart contracts, solidity, source code repository**

I. Introduction

Since the introduction of blockchain technologies in 2008 [1], the idea to build a completely decentralized Web without firewalls and 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 consequences. The first and the most important example of such an environment is the Ethereum Virtual Machine (EVM) that is responsible for defining and enforcing the rules that govern the computation of a new valid state after the inclusion of a block. 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, with the development of newer cryptocurrencies with an emphasis on financial inclusion. Smart contracts provide a way to generate new cryptocurrency tokens, known as *application level tokens* on existing networks without the need to define a new protocol [6].

Do we really need this last sentence here? Actually, I think that everything written from "in number" is not absolutely necessary,

The defining feature of a blockchain is immutability, which makes it very difficult 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 [7]. Deployment and invocation of a smart contract on the blockchain leads to gas fees 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 cost is proportional to the computation required for the contract operations [8]. Thus, it is important to deploy contracts which are error-free and efficient in order to reduce the costs to the user, as also to optimize the energy consumption.

Source code repositories are an important aid to train systems that identify code smells and vulnerabilities. Code smells are violations of fundamental design principles in the source code. Code smells can thus, apart from introducing errors in the code, also lead to suboptimal performance and efficiency. Code can further be optimized for maintainability, reusability, comprehensibility and extensibility [9]. Apart from this, versions of the same source code can also provide an insight into the change of performance of the software over consecutive versions [10].

Code mining approaches involve detecting patterns from a repository of source files and analyzing the purpose those patterns serve. Existing code mining approaches mainly involve extracting business logic, automating the translation of code in one language to another, provisioning searching and indexing source files, detecting similarities (used primarily in detecting code smells), to summarize and visualize algorithms and also to extract and analyze information from source code metadata [11].

There is a little bit of a logical gap between this paragraph and the next one. Here, you write about general code mining, in the next about optimization. Try to bridge that gap a little bit.

However, 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 the gas cost [12]. The smart contract code, when deployed on the Ethereum network, is compiled into *bytecode*. The bytecode is a string made of concatenated op-codes that are executed in sequence when the smart contract is invoked [5]. The execution of each opcode is associated with a certain energy consumption, which is calculated and charged from the user accoording to the EVM *gas model* [5]. This means that optimizing and analyzing smart contracts need to be done not only at the level of the source code, but also at the corresponding bytecode. Optimizing smart contract code directly implies optimizing the gas cost through the bytecode. This introduces an additional constraint and hence requires techniques for constraint-optimized pattern recognition [13].

As already mentioned, consecutive versions of the same source code can help in detecting specific changes to the code performance due to specific changes in the source code [10]. Whereas a comparison between disparate files requires establishment of a similarity measure at the beginning [14], earlier complete versions of the same source code can provide a benchmark on which to measure the performance of later versions [10]. This can make it easier to detect optimizing patterns in the source code. These patterns, once established, can then be applied to optimizing similar inefficiencies in other source files. Hence, a repository containing multiple versions of each source file would prove to be helpful as a starting point in applying code mining approaches to optimizing smart contracts, which is a relatively unexplored research direction.

Apart from code smell detection and optimization for smart contracts, which is our primary research direction, we believe that this repository can be helpful to researchers in a wide range of other application areas. For instance, code recommendation systems use publicly available code on the Internet to suggest usable pieces of source code to software developers. With the recent progress and development of large language models (LLMs), code generation models such as Codex [15] are being used by developers to speed up software design and development, and are proving to be a viable business models for companies. Apart from recommender models focused on source code, general purpose LLMs such as ChatGPT, which is a version of GPT-3 fine-tuned for factoring in human feedback [16] also use large datasets of source code in order to make the token embeddings as generalized and diverse as possible [17]. Such models can also be used for code and algorithm summarization and visualization [18]. In security research, commits of source code repositories can be used to analyze patterns in smart contract code that eliminate or introduce certain vulnerabilities [19]. A dataset of source code can also be used to evaluate and benchmark the performance of static and dynamic analysis tools, which can also be used to detect software vulnerabilities [20]. Analysis of code repositories can also provide useful to generate developer guidelines on writing clean and maintainable code [21], in the evaluation of software test methods [22], in the structural and stylistic comparison of code written in different environments [23] and in many other studies.

Since the current ver-

The multitude of potential research areas which are based on source code shows the need for the Smart Contract Repository presented in the work at hand.

Very fine abstract, showing the need for the work at hand. However, again a logical gap to the next paragraph (i.e., the one about Ethereum

Ethereum was the first blockchain to introduce smart contracts and the high level language Solidity to program smart contracts in. Solidity has grown to be the most mature and popular programming language for smart contracts [24]. A majority of the contracts available on GitHub is written for the Ethereum blockchain, in Solidity [25], and hence, Solidity was the language of choice in preparing the dataset. The dataset is compiled by crawling through and scraping publicly accessible repositories on GitHub that have Solidity as the main language. The repositories are identified and all available commits of each file are downloaded using the GitHub API. Due to the varying licenses of GitHub repositories, the dataset is being distributed in the form of a Python script which can be run to scrape GitHub and prepare the dataset in an SQLite database. During experimentation, the script collected 163,961 files from 1021 repositories.

The remainder of this paper is structured as follows: In Section II, we define prerequisites for our approach. Afterwards, Section III discusses the relevant and existing work, Section IV analyses the requirements for preparing the repository, Section V describes how the script was implemented and how it generates and stores the repository. Finally, Section VI discusses some limitations of the existing approach and Section VIII concludes the paper and lays the foundation for future work.

II. Preliminaries

A. Blockchain:

The blockchain technology was introduced by Satoshi Nakamoto in the Bitcoin White Paper [1]. A blockchain is a tamper-proof distributed ledger, which is similar to a distributed storage of transactions [26] 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 or immutable blocks which are made tamper-proof using cryptographic techniques. According to Dannen [26], the 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

Please extend this paragraph by writing down what we are actually delivernonce 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 only 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. The technical specifications of Ethereum were spelled out in the Ethereum Yellow Paper [5].

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 Contracts 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. According to Dannen, smart contracts run business logic and enforce payment agreements between parties [26]. Albert et al. have defined a smart contract from a more technical perspective as a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain [27].

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], [28]. 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 [29]. 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) [30]. 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 complied into bytecode when deployed on the blockchain. For example, Ethereum introduces the programming language called Solidity, which can used to write smart contracts which are then run on the Ethereum Virtual Machine (EVM). The EVM is a quasi-Turing complete machine [5] in that it restricts the over-utilisation 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 cost [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 which 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 Ethereum Virtual Machine (EVM). Reducing the number of such operations and replacing expensive operations with cheaper ones can reduce the gas cost and hence, indirectly, the actual energy consumption of the smart contract [8].

D. Code Mining:

Mining software repositories is the process of finding patterns by analyzing the large corpus of source code available in software repositories [31]. 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.

Code mining describes the process of automatic evaluation and optimization of software code. This is usually done with source code in a high level programming language, but it can also be done with low-level byte code. 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 [9]. One of the main purposes of code mining is to automatically detect and possibly remove code smells. Different algorithms can be used for code mining, depending on the optimizations that should be achieved.

Fowler et al. were the first to describe the term code smell in their book Refactoring [32].They defined the term as diagnosing constructs that may indicate that the software needs to be refactored. In their book, they identified 22 code smells and suggest ideas of how to handle these design violations.

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 safes 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 afterwards [33]. In all of these areas, code mining techniques can be used to analyze the source code and improve its quality by finding code smells.

Code mining can therefore include different techniques and tools and is applicable to achieve different goals. Pham et al. find that it can be used "to help understanding, analysing, transforming, improving, maintaining or evolving a particular system, or to improve best practices for the development of new systems" [34]. Hence, it is a promising approach to use code mining in the field of gas cost optimization of smart contracts.

III. Related Work

To the best of our knowledge, the analysis of smart contract code has only gained relatively little attention by the research community so far. Nevertheless, there are a number of according approaches to the optimization of gas cost of smart contracts, which we discuss in the following.

Notably, code smell detection and code optimization for conventional programs have been a very vivid field of research for many years [9] and some of those concepts have trickled down to smart contracts. SmartDagger [35] is a tool which used static analysis of smart contract bytecode to detect cross-contract vulnerability. Manticore [36] is a tool that uses dynamic symbolic execution to test and detect flaws in source code. Existing program verification frameworks such as Boogie or LLVM can be used to perform formal verification of smart contracts [37].

Brandstätter et al. [7] 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 [38]–[41]. Starting in 2017, they were one of the first to address the problem of gas optimization. They identified seven antipatterns in Solidity bytecode and presented the tool GASPER to identify three of these gas-costly patterns [38]. 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 [40] and GasChecker [39] followed, again dedicated to optimize byte code based on pre-defined rules and patterns.

Nagele et al. presented the SuperOptimizer *ebso* in 2020. Using techniques of a Satisfiability Modulo Theories (SMT) solver, they developed and tested a tool that could run EVM byte code optimizations [42]. In 2017, Albert et al. presented EthIR, "a framework for analyzing Ethereum bytecode" [43]. EthIR is an extension of OYENTE that was published by Luu et al. [44]. 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 [45] for symbolic execution of the smart contract code [44]. In 2020, Albert et al. introduced the Eclipse[1] plugin GASOL that allows the user to investigate and optimize their Solidity code with different cost models [27].

Most of the published work in the field of optimization of smart contracts focus on the analysis of the compiled bytecode and on detecting similarities between the source code and predefined patterns that can indicate vulnerabilities and malicious constructs. There has been limited research on deriving those patterns from a automatically, which could lead to the automation of existing optimization possibilities while providing opportunities for previously undetected optimization patterns. The existing attempts at automating the optimization of smart contracts mainly utilize static and dynamic analysis of the smart contract source code or bytecode. In 2022, Albert et al. published another tool called Syrup [12] that 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 [45] to obtain an optimized version of the sub-block [12]. 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 [46]. Di Angelo et al. released an investigation and listing of 27 different tools designed to analyze smart contracts [47]. They concluded that the research community was in need of an open source benchmark suite of smart contracts. 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 [48].

As is evident from the previous discussion, a reasonably large repository of smart contracts such as the one presented in this paper is essential, both for deriving new approaches to automatically optimizing smart contracts and for evaluating and benchmarking such tools. Using such a repository combined with static analysis of smart contracts can provide additional patterns that can be used to optimize new contracts, without depending on fixed rules or the optimization capability of SMT solvers.

In the domain of repositories of smart contracts, Pierro et al. developed the Smart-Corpus software that creates a repository of verified smart contracts extracted from Etherscan and allows the download of and computation of metrics on the contracts in the repository [49]. Other repositories, such as the ones by Ren et. al. [50], Ferreira et. al. [51] and SCRAWLD [52] all contain deployed versions of smart contracts or synthetic smart contracts and do not have associated version histories. The repository presented in this paper is distinct and novel

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

in that it includes metadata as well as the complete version history of each contract and is not limited to a particular use case. Thus this repository can prove useful to all use cases mentioned in this section.

Apart from improving other smart contracts, a repository can also benefits researching conducting studies on contract design patterns. For example, Wöhrer et al. presented recommendations for the development of smart contracts closely connected to the money transferring fashion of smart contracts [53]. In 2020, Marchesi et al. presented 24 design patterns for smart contracts categorized based on their features [54]. While most of such studies are based on manual inspection of existing smart contracts, a corpus of smart contracts may be helpful in automating the detection of such patterns and potentially analyzing their effects on gas cost. Additionally, evaluating and designing new programming languages aimed at providing predictable gas consumption with efficient bytecode, requires a dataset of smart contracts.

As mentioned in Section I, many applications developed for conventional code can be extended to smart contracts with training and evaluation on the proposed dataset. Recommendation systems such as HelpMeOut [55], ParseWeb [56] and others can be trained to recommend snippets of Solidity code . Large Language Models focused on code, such as Codex [15] can be trained on the repository to work with smart contracts to add to its capabilities. In security research, apart from analysing individual deployed smart contracts, their version histories stored in the repository can be used to detect patterns that introduce or resolve vulnerabilities extending the work done in [19]. Since the programming languages used in writing smart contracts are high level languages similar to conventional programming languages, most of the existing tools and research can be extended to smart contracts. A complete repository of source code is the first step in achieving that aim. Our repository aims to fill that gap and provide researchers with complete information regarding smart contract source code and their versions. This repository is different from the repository of smart contracts that can be obtained from existing APIs, such as those provided by Etherscan[2], since such APIs can only provide a single version of a smart contract deployed on the blockchain.

IV. Requirements Analysis

As written above, the repository introduced in this paper aims to help researchers and software developers move away from the existing rule-based approaches and derive applicable patterns from analyzing existing source files and versions thereof. Multiple versions of the same source file can be analyzed to find out incremental optimizations to gas consumption and the specific statements or constructs that are behind those optimizations. That is why this repository needs to source its data from publicly available repositories that also store multiple incremental versions of the smart contracts. This can be achieved by crawling repositories that use versioning systems like Git to store commits of the source files.

As has been described in Section I, we opted to support Solidity in the repository, since Solidity is the most established programming languages for smart contracts today [24]. We keep the option open to add further languages like Vyper or Yul if they find a wider distribution in the future.

In order to follow the best practices for creating a repository, a number of approaches for collecting datasets were studied. One of the best examples of preparing a large scale dataset from GitHub repositories can be found in [17]. In the area of smart contracts, repositories can be found in the works of [**?**], [51] and others. Similar to these datasets, our repository should contain the source code of all available contracts. Apart from the source files themselves, additional features need to be incorporated through the storage of metadata about the smart contracts, in order to provide researchers with complete information and to reduce the amount of pre-processing required before analyzing the code in the repository. Also, multiple versions of the source code, if available, can be used to compare incremental changes to the gas consumption and also to detect the introduction to vulnerabilities through changes in the source code. The metadata may include the name of the contract, a description of what it is intended for, details about the source repository, the compiler versions used to generate the bytecodes and also the commit messages, which can be helpful in manual annotation of changes. As mentioned in [17], the repository creation procedure should take into account repository licenses, particularly if the repository is to be made publicly available. The repository should be able to filter out files without permissive licenses.

The repository needs to be stored in a database along with an appropriate database management system (DBMS) that can enable easy and efficient querying capabilities. For this purpose, SQL databases such as SQLite, MySQL or NoSQL databases such as MongoDB can be used. The smart contracts should be searchable using their metadata, names, also the language used and the corresponding compiler versions. A general purpose relational DBMS or an off-the-shelf document retrieval system operating on a NoSQL database provides ready deep and optimized querying capabilities while ensuring scalability and extendability to newer fields that can be added to the database.

V. Implementation

The task of generating this repository was completed through the following sub-tasks::

- *Selecting a data source*: Pierro et al. list some sources that can be used to collect smart contract source code from the Web [57]. These resources include commonly known code repository hosting platforms (GitHub[3], GitLab[4] and BitBucket[5]). Additionally, several Ethereum block explorers are available. These services allow for different kinds

[2] https://etherscan.io/

[3] https://github.com

[4] https://about.gitlab.com

[5] https://bitbucket.org

of interactions with and the analysis of the Ethereum blockchain. EtherScan[6] and EtherChain[7] are two examples of block explorers. In their paper, Pierro et al. use the API of EtherScan to collect verified smart contracts and store them in a database [57]. However, block explorers can only provide the version of the smart contract that is deployed on the blockchain, but not its previous versions. Hence, for the work at hand, GitHub was selected as the source of the data in the repository. GitHub also provides a large volume of source files available and integration with Git, which ensures availability of versions of the source files. Other similar repositories such as GitLab or BitBucket have severely limited or pay-walled provisions for searching and crawling. GitHub, on the other hand, ensures accessibility of the code and its versions through its APIs. GitHub's APIs also allow the user to search for and download files based on the licensing scheme adopted by the developers. This enables the script introduced in this paper to allow users to distinguish between files with viral and non-viral licenses.

- *Automating the data collection process*: To perform data collection from GitHub automatically, a python script was implemented. The script is based on the *github-searcher*[8] written by Schröder et al. to run a stratified search for code snippets on all public repositories [58]. This is done to circumvent the limitation of 5000 requests per hour per API-token imposed by the GitHub API in its free version.
- *Collecting the required files*: The implemented script prepares the search by requesting the user for the following information:
  - `database`: the name of an existing database to store the results in. If not provided, the script creates a database file called `results.db`.
  - `statictics`: the name of an existing file to store the search statistics in.
  - `min-size`: the minimum file size to search for and collect.
  - `max-size`: the maximum file size to search for and collect. GitHub itself imposes a limit of 384 KB on the file size.
  - `stratum-size`: the lengths of file-size ranges to be used for the stratified search.
  - `no-throttle`: to disable request throttling.
  - `search-forks`: this option, if set to `true`, adds `fork:true` to the GitHub search query and includes forks of repositories in the search. This expands the field of search but risks including duplicate files in the results.
  - `license-filter`: if set to `true`, filters the search results to include only open-source licensed repositories.
  - `github-token`: lets the user enter a GitHub API token in order to access the search APIs. If an API token is present in the environment variables, it is used automatically.

  In addition to the restriction of a maximum of 5000 requests per hour, the GitHub search API also imposes a limitation of 1000 results per query. To get around this restriction, the script restricts searches to a certain file size. The script makes use of the technique of *stratified sampling* [59], running the same query, but with different file size ranges in order to retrieve a larger number of files. Further, to download all files returned by the GitHub API, even within the 1000 file limit, the script makes use of *Pagination* [60], with 100 elements per page.

  The sampling statistics file is used to store information about a previously incomplete search, in order to start from the last checkpoint in case of a connection failure, a fatal exception or an interruption by the user. To avoid the secondary rate limit imposed by the GitHub API (to prevent denial of service attacks through a large number of requests from the same source within a short period of time), the script introduces a delay between consecutive calls of the search API. For each file-size range, the query is repeated twice, once in the original order and again in a reverse order, thus sampling the population from both ends, in order to avoid the 1000-results-per-query limit to the best of our ability.

  After the required parameters are collected, the script uses the GitHub API to search for repositories that have Solidity as the main language.

  In general, the script performs the following operations to collect data:
  - It accesses a list of files for each public repository using the GitHub API[9]. Searching for Solidity files within this list creates a collection of contract files for each repository. Using the regex `\w\.sol` on the file path, the script determines if a file is written in Solidity. For each file, the path and the repository are also stored.
  - The script accesses a list of commits for each file through the corresponding GitHub API[10]. Iterating through all the files and calling the API for each one creates a version history of each Solidity file, ordered chronologically.
  - The script downloads the raw source code for each file using the GitHub user content API[11].

  During experimentation, this resulted in a collection of 163,961 Solidity files, including 299,437 commits. At this point, the script has collected all data it needs to store in the persistent storage to create the repository. All these solidity files have their complete version history

[6] https://etherscan.io/

[7] https://etherchain.org/

[8] https://github.com/ipa-lab/github-searcher

[9] https://api.github.com/repos/<owner>/<repository>/git/trees/master?recursive=1

[10] https://api.github.com/repos/<owner¿/<repository>/commits?path=<file_path>

[11] https://raw.githubusercontent.com/<owner>/<repository>/<commit_sha>/<file_path>

and metadata that are the distinctive features of this repository.

The source stored for each file in the repository helps to track modifications to the same file and also files belonging to the same repository or project. However, one drawback of this approach is that forks cannot be filtered and hence can lead to duplicate files in the repository. The `search-forks` option tries to solve this problem by eliminating fork search altogether. However, a more nuanced comparison of duplicates is required to have the expanded field of search while avoiding duplicates. A hash-based solution (as implemented by the duplicate-file-finder[12]) can be adopted to compare files and remove duplicates.

The collected data are stored in an SQLite database. SQLite was chosen because of its lightweight, cross-platform nature and the absence of the need for any administration. SQLite databases enable the user to analyze the dataset using simple yet powerful command line tools. SQLite interfaces come included with popular programming languages like Python and R, thereby reducing the users' overheads for setting up and maintaining a database. The script introduced in this paper stores all collected information, including the file metadata in three tables:

1) `REPO`: This table contains information about each collected repository. The schema is as follows:
   - `REPO_ID INTEGER PRIMARY KEY`: An integer uniquely identifying each collected repository. Acts as the primary key for the table.
   - `NAME TEXT NOT NULL`: The name of the repository.
   - `FULL_NAME NOT NULL`: The full name of the repository, including the owner's username.
   - `DESCRIPTION TEXT`: A description of the repository from the README.md file.
   - `URL TEXT NOT NULL`: The complete URL of the repository.
   - `FORK INTEGER NOT NULL`: The number of forks of the repository.
   - `OWNER_ID INTEGER NOT NULL`: The user ID of the owner.
   - `OWNER_LOGIN`: The username of the owner.
2) `FILE`: A table to store the files collected from the `master` branch of each repository. The schema is as follows:
   - `FILE_ID INTEGER PRIMARY KEY`: A unique ID for each file.
   - `NAME TEXT NOT NULL`: The name of the file.
   - `PATH TEXT NOT NULL`: The full path of the file in the GitHub repository.
   - `SHA TEXT NOT NULL`: The SHA-1 hash of the latest commit of the file.
   - `REPO_ID INTEGER NOT NULL`: The unique repository ID; references the `REPO_ID` from the `REPO` table.
3) `COMMIT`: A table to store the commits for each collected file. This table stores the version history of the files to aid in analyzing incremental changes. The schema is as follows:
   - `COMMIT_ID INTEGER PRIMARY KEY`: A unique ID assigned to each commit.
   - `SHA TEXT NOT NULL`: The SHA-1 hash of the committed file.
   - `MESSAGE TEXT NOT NULL`: The commit message.
   - `SIZE INTEGER NOT NULL`: The size of the commit in bytes.
   - `CREATED DATETIME DEFAULT CURRENT_TIMESTAMP`: The timestamp of the commit.
   - `CONTENT TEXT NOT NULL`: The contents of the committed file. The source code is stored in this field.
   - `PARENTS TEXT NOT NULL`: The SHA-1 hash of the commit the current commit is based on.
   - `FILE_ID INTEGER NOT NULL`: The file ID each commit corresponds to. References the `FILE_ID` from the `FILE` table.

   The size, in bytes, of the file content is obtained from the length of the content of the response object. The timestamp is collected directly from the API response.

The repository can be accessed using standard SQLite CLI tools and also through the Python and R interfaces for in depth and automated analysis.

VI. Limitations

The script presented in this paper attempts to create a repository of smart contracts, complete with their metadata, source information and most importantly and distinctively, version histories, in order to aid researchers offer a fully functional corpus and provide as much readily available information as possible to perform their aimed analyses.

The application itself, however, has some limitations with regard to additional functionalities. Though the repository is fully functional, the information stored in the repository can still be enhanced to add even more ready information to the researchers. Each version of a source file can be associated with its corresponding bytecode and ABI, for faster analysis of incremental changes to gas costs. Also the Solidity compiler versions can be retrieved from the file header and stored separately to enable querying for files with specific compiler Solidity versions.

Limitations also arise from the crawling process, the most important one being the fact that the scraping is currently limited to Solidity. Although the data model stores strings and hence can handle any syntax and language, the database

[12] https://github.com/michaelkrisper/duplicate-file-finder

has been built with Solidity contracts in mind. However, as discussed before, the script can be extended to include other languages including letting the user decide their language of choice.

It can be argued that a database containing files originating only from public repositories of GitHub may not be representative of the currently deployed smart contracts. As mentioned before, multiple block explorers are available that list deployed smart contracts. However, most of these block explorers do not make the source code of the contracts is publicly accessible, neither do they offer access to multiple versions of the same contract, a limitation which arises from the immutability of the blockchain. Under such circumstances, choosing GitHub is the logical choice since GitHub is the most popular among software development organizations, which makes it easier to assume that the code available on GitHub at least provides an estimation of deployed code. Similar approaches of using GitHub as a source of data for code mining algorithms can be found in literature [**?**], [58], [61]. However, usage of GitHub can lead to the inclusion of unfinished and untested source code in the repository, and as mentioned before, duplicate code, in the absence of a mechanism to properly handle forks.

Apart from this, restrictions imposed by the GitHub APIs make it difficult to construct a tree-like version history of a particular source file, given the fact that repositories are structured into branches, each branch having a different set of modifications of a file.

VII. Analysis of the Repository

In the previous section, it was mentioned that crawling GitHub might lead to distortions in the dataset. This section presents a short quantitative analysis of a sample of the database generated during experimentation. As written in Section V, we were able to generate a corpus of 163,961 Solidity files, which were the results of 299,20437 commits to Github.

Figure 1 shows the distribution of the number of versions of the collected files. The absolute frequency of occurrence of the number of versions are represented in the graph. The scale of the Y-axis is logarithmic and the data point at 200 versions included all files that have 200 or more versions, in order to improve the readability of the graph. 132519 out of 163931 or around 80% of the files have only 2 versions. It is evident that only a few smart contracts include more than 100 versions of the source code. In spite of that 31412 files have multiple versions. This provides a reasonably large dataset for code mining.

When collecting data from GitHub, in the first phase, only the repositories containing Solidity source files were collected. This was followed by scraping all Solidity source files from each of those repositories. This order ensured that all relevant files of each repository were accessed, since the GitHub search API does not return all relevant files by simple content search. As a consequence, all 162931 files in the database come from the same 1021 repositories. To investigate this further, the distribution of files per repository was calculated (see Figure 2). This diagram shows the absolute frequency of the number of files that being to a particular repository. The Y-axis is again logarithmic and the data point at 500 includes all repositories having 500 or more relevant files. The repository *ethereum/solidity-fuzzing-corpus* contributed the largest number of files - 33145 - to the repository. 29947 files come from the repository *tintinweb/smart-contract-sanctuary-ethereum*. Additionally, the mean of the distribution is at 160 while the standard deviation is 1514. This shows that a few large repositories have a large impact on the dataset, and consequently, on the quality of source code in it. 13.7% of the repositories contribute only 3 files. But the high standard deviation captures the widely varying numbers of files contributed by repositories. The outlier of exactly 290 Solidity files that are present in 9 different repositories seems to originate from a specific Solidity tutorial project called *cryptozombies*[13] that was forked by multiple users on GitHub.

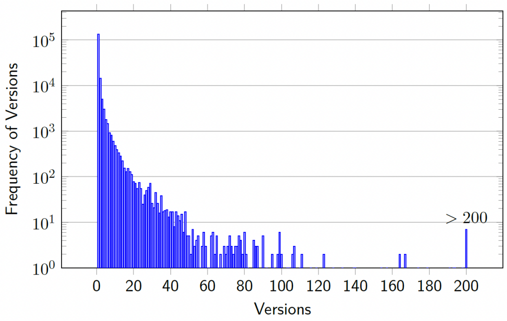

Fig. 1. Distribution of versions of smart contracts in the repository

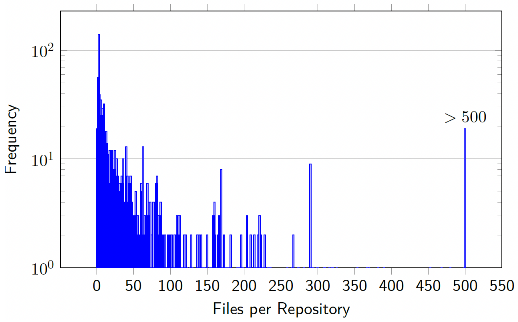

Fig. 2. Distribution of the number of files in the GitHub repositories

VIII. Conclusion

Blockchain technologies are a growing industry and research field. With the introduction of smart contracts on second generation blockchains, the scope of application of blockchains has increased, as has corresponding gas costs and consequently, energy consumption. The immutable nature of a blockchain is what makes it secure, but that also means that developers have to write efficient and bug-free code before

[13] https://cryptozombies.io

deploying smart contracts. Hence, tools for developers to optimize and check their code, are under active research. The repository introduced in this paper aims to aid such researchers by helping them to understand and analyze the effects of specific changes of the source code on the outcomes of the smart contract, especially in automating and finding new ways to optimize gas costs. In spite of the mentioned limitations, the repository offers to provide researchers with complete information about a smart contract and its versions.

As has already been mentioned before the script, and consequently the repository, can be extended to include source files using other smart contract programming languages. Also, a compiler can be included as part of the script in order to generate and store the bytecode of the collected contracts. The source of data in the repository can be expanded, crawling other public repositories, or even private repositories a user may have access to. An attempt can be made to reconstruct the version history of a source file across multiple branches of the repository, to enable more fine-grained analysis. A gas-cost estimation framework, such as GASOL [27] can be integrated into the script to generate gas cost estimations of the collected contract versions. Finally, the repository can be associated with a Web app that can make it easier to visualize the repository of contracts and also provide APIs for more nuanced filtering and retrieval of the dataset, even without knowledge of SQL. These tools can also be included as a plugin to an IDE that can enable easier interaction with the database.

Apart from the improvements to the repository itself, the next steps involve actually using the repository for applying code mining algorithms, as intended. Upcoming research using this repository should aim at using the enhanced information in this repository towards achieving the research aims mentioned in the previous sections, primarily in the fields of gas consumption, machine learning and qualitative and quantitative analysis of the smart contracts.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 03 2009.

[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*, vol. 151, no. 2014, pp. 1–32, 2014.

[6] S. Balbo, G. Boella, P. Busacchi, A. Cordero, L. De Carne, D. Di Caro, A. Guffanti, M. Mioli, A. Sanino, and C. Schifanella, "Commonshood: A blockchain-based wallet app for local communities," in *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, 2020, pp. 139–144.

[7] 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.

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

[9] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *J. Softw. Evol. Process*, vol. 27, no. 11, p. 867–895, nov 2015. [Online]. Available: https://doi.org/10.1002/smr.1737

[10] D. G. Reichelt and S. Kühne, "How to detect performance changes in software history: Performance analysis of software system versions," ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 183–188. [Online]. Available: https://doi.org/10.1145/3185768.3186404

[11] H. Kagdi, M. Collard, and J. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance*, vol. 19, pp. 77–131, 03 2007.

[12] 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. [Online]. Available: https://doi.org/10.1145/3506800

[13] 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.

[14] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," vol. 31, 03 1999, pp. 266–270.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[16] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022. [Online]. Available: https://arxiv.org/abs/2203.02155

[17] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," 2022. [Online]. Available: https://arxiv.org/abs/2211.15533

[18] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," 2022. [Online]. Available: https://arxiv.org/abs/2207.04237

[19] G. Antal, M. Keleti, and P. Hegedűs, "Exploring the security awareness of the python and javascript open source communities," 2020. [Online]. Available: https://arxiv.org/abs/2006.13652

[20] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 544–555. [Online]. Available: https://doi.org/10.1145/3533767.3534380

[21] S. A. Chowdhury, G. Uddin, and R. Holmes, "An empirical study on maintainable method size in java," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 252–264. [Online]. Available: https://doi.org/10.1145/3524842.3527975

[22] V. Veloso and A. Hora, "Characterizing high-quality test methods: A first empirical study," ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 265–269. [Online]. Available: https://doi.org/10.1145/3524842.3529092

[23] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A large-scale comparison of python code in jupyter notebooks and scripts," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 353–364. [Online]. Available: https://doi.org/10.1145/3524842.3528447

[24] 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

[25] A. Das, G. Uddin, and G. Ruhe, "An empirical study of blockchain repositories in github," ser. EASE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 211–220. [Online]. Available: https://doi.org/10.1145/3530019.3530041

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

[27] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," 2019. [Online]. Available: https://arxiv.org/abs/1912.11929

[28] 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, 09 2021.

[29] M. Alharby, A. Aldweesh, and A. v. 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.

[30] V. Morabito, *Smart Contracts and Licensing*, 01 2017, pp. 101–124.

[31] 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.

[32] *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[33] 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.

[34] H. S. Pham, S. Nijssen, K. Mens, D. Di Nucci, T. Molderez, C. De Roover, J. Fabry, and V. Zaytsev, "Mining patterns in source code using tree mining algorithms," in *Discovery Science*, P. Kralj Novak, T. Šmuc, and S. Džeroski, Eds. Cham: Springer International Publishing, 2019, pp. 471–480.

[35] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "Smartdagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 752–764. [Online]. Available: https://doi.org/10.1145/3533767.3534222

[36] 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

[37] 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

[38] 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.

[39] 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.

[40] 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," 01 2020.

[41] 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

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

[43] 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. [Online]. Available: https://doi.org/10.1007%2F978-3-030-01090-4_30

[44] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[45] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[46] 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)*. IEEE, may 2019. [Online]. Available: https://doi.org/10.1109%2Fwetseb.2019.00008

[47] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019, pp. 69–78.

[48] 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.

[49] G. A. Pierro, R. Tonelli, and M. Marchesi, "Smart-corpus: an organized repository of ethereum smart contracts source code and metrics," 2020. [Online]. Available: https://arxiv.org/abs/2011.01723

[50] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 566–579. [Online]. Available: https://doi.org/10.1145/3460319.3464837

[51] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," *CoRR*, vol. abs/2007.04771, 2020. [Online]. Available: https://arxiv.org/abs/2007.04771

[52] C. S. Yashavant, S. Kumar, and A. Karkare, "Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities," 2022. [Online]. Available: https://arxiv.org/abs/2202.11409

[53] M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the ethereum ecosystem," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1513–1520.

[54] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020, pp. 9–15.

[55] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1019–1028. [Online]. Available: https://doi.org/10.1145/1753326.1753478

[56] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 204–213. [Online]. Available: https://doi.org/10.1145/1321631.1321663

[57] G. A. Pierro, R. Tonelli, and M. Marchesi, "Smart-corpus: an organized repository of ethereum smart contracts source code and metrics," 2020. [Online]. Available: https://arxiv.org/abs/2011.01723

[58] M. Schröder and J. Cito, "An empirical investigation of command-line customization," *Empirical Softw. Engg.*, vol. 27, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10036-y

[59] R. Levin and Y. Kanza, "Stratified-sampling over social networks using mapreduce," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 863–874. [Online]. Available: https://doi.org/10.1145/2588555.2588577

[60] J. Cao, W. Wang, and Y. Shu, "Comparison of pagination algorithms based-on large data sets," 01 2010, pp. 384–389.

[61] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," 2021. [Online]. Available: https://arxiv.org/abs/2103.04682