Differential Testing Solidity Compiler through Deep Contract Manipulation and Mutation

Zhenzhou Tian

tianzhenzhou@xupt.edu.cn

Xi'an University of Posts and Telecommunications, China

Fanfan Wang

Xi'an University of Posts and Telecommunications, China

Yanping Chen

Xi'an University of Posts and Telecommunications, China

Lingwei Chen

Wright State University

Research Article

Keywords: Smart Contract, Differential Testing, Solidity Compiler Fuzzing, Contract Generation, Deep Learning, Bug Detection

Posted Date: February 7th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-3924118/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Software Quality Journal on April 23rd, 2024. See the published version at https://doi.org/10.1007/s11219-024-09673-5.

Differential Testing Solidity Compiler through Deep Contract Manipulation and Mutation

Zhenzhou Tian[1,2,3*], Fanfan Wang[1], Yanping Chen[1,2,3], Lingwei Chen[4]

[1*]Department of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, 710121, Shaanxi, China.
[2]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an, 710121, Shaanxi, China.
[3]Xi'an Key Laboratory of Big Data and Intelligent Computing, Xi'an University of Posts and Telecommunications, Xi'an, 710121, Shaanxi, China.
[4]Department of Computer Science and Engineering, Wright State University, Dayton, 45435, OH, USA.

*Corresponding author(s). E-mail(s): tianzhenzhou@xupt.edu.cn;
Contributing authors: wangfanfan@stu.xupt.edu.cn;
chenyp@xupt.edu.cn; lingwei.chen@wright.edu;

Abstract

Solidity, the language utilized for developing smart contracts, has been gaining increased importance in blockchain system. Ensuring bug-free of its accompanying language compiler, which converts the contract source codes into executables finally deployed on the blockchain, is thus of paramount importance. This study presents DeSCDT, a **De**ep learning-based **S**olidity **C**ompiler **D**ifferential **T**esting approach, to explore possible defects in solidity compiler. At the core lies a well-behaving deep contract generator following the Transformer architecture and learnt with diverse contract code. From an initial seed pool of contracts carefully picked through semantic encoding and clustering, the generator is capable of stably producing highly syntactic-valid and functional-rich smart contracts, with three meticulously formulated generation strategies and a set of mutation operations. Subsequently, in the meantime of compiling these generated contracts to trigger compiler crashes, a differential testing environment is setup to explore misoptimization bugs, by observing the inconsistencies between the outcomes and

the aspects including gas consumption and opcode size of the optimized and non-optimized bytecodes. For the experiments, the syntactic validity and diversity of the contracts generated with DeSCDT, as well as its ability in discovering compiler defects, are investigated. The findings indicate that DeSCDT can effectively generate syntactically correct contracts with a pass rate of 90.8% alongside high diversity. Among the contracts tested for a 24-hour running of DeSCDT, 37.4% of them expose inconsistencies across the optimized and non-optimized version of the same contract. Six bugs that could trigger direct crashing of the compiler have also detected.

Keywords: Smart Contract, Differential Testing, Solidity Compiler Fuzzing, Contract Generation, Deep Learning, Bug Detection

1 Introduction

Solidity is an emerging programming language specifically designed to ease the development of smart contract, pivotal in underpinning the burgeoning blockchain transactions. To date, approximately 50 million contracts have been deployed on Ethereum, forming a foundational layer for a myriad of decentralized application spanning across diverse domains, including cryptocurrency wallets, financial services, social and games. The source code of a smart contract must be processed with its language processor, i.e., the solidity compiler, so as to be deployable and invokable on the blockchain. Thus, the bug-free or correctness of the compiler itself is of paramount importance. Failing to do so may make the compiled contract operate inconsistently as its was designed for, cause catastrophic runtime errors, or even introduce security vulnerabilities that can be exploited to enforce contract attacks.

Fuzz testing [1] is acknowledged as one of the most highly effective methods for uncovering potential bugs or defects within both the regular software programs and the compiler infrastructures. Basically, the fuzzing process involves systematically producing and feeding inputs into the program under test (PUT) and observing its responses to identify whether the PUT violates certain correctness policies, such as straightforward crashes, unexpected or incorrect outputs [2]. While extensive researches have been conducted on fuzz testing for the more conventional language processors [3], such as compilers for C and Java, and interpreters for Python and JavaScript, investigations into fuzz testing for the new and evolving Solidity compiler remain relatively underexplored.

In fuzzing the solidity compiler, ceaselessly producing syntactically valid test cases, i.e., smart contracts, without doubt is a critical step [4]. It ensures both the effective revealing of deep hidden bugs and the maintaining of a valid testing throughput, as any error in the generated test programs can halt the compiler's execution and hinder the tester from probing into the more complex aspects of translation logic. Besides, the unique gas mechanism [5–7], which necessitates the consumption of gas fees for either deployment or execution of contracts on the blockchain (to avoid squandering of computational resources and under-priced DoS attacks due to contract abuse), imposes special requirements to the compiled outcomes. In response to this peculiarity, the

solidity compiler offers an optimization module with the purpose to both diminish the code size (which reduces the deployment gas cost) and optimize the comprising opcodes (which reduces the gas fees for invoking the contract) of the compiled outcomes. This brings a different set of bug oracles in fuzz testing the solidity compiler compared with other conventional compilers.

In addressing the above discussed first point, i.e., **the effective generation of syntactic valid contracts**, we establish a reliable deep learning-based contract generator based on the Transformer architecture, trained with an enormous amount of varied contract codes sourced from Etherscan. Then, starting from a curated set of initial seed contracts, which are identified through semantic encoding and clustering, the learnt generator can consistently produce syntactically valid and functionally diverse contracts. This is achieved through carefully devised generation strategies, with which to guide the generator manipulating the seed contracts without breaking the integrity of their statements. Complementing this, we also integrated fine-grained mutation methodology, which serves to further augment the variety of the contracts produced.

Pertaining to the second point, i.e., **novel bug oracles attributable to the unique gas mechanism**, differential testing is leveraged to unearth potential misoptimization defects within the Solidity compiler. This process involves the establishment of a localized Ethereum Virtual Machine (EVM) execution environment, in which the optimized and non-optimized bytecode versions of contracts compiled from identical source code are executed. Meanwhile, meticulous recording and comparative analysis of the metrics, including gas consumption, opcode sizes, and computational outcomes, are performed. Should there be discrepancies in their computational results, or deviations in gas consumption and opcode sizes that contravene the expected patterns of the compiler optimizer, then a misoptimization bug has been discovered.

The contributions are summarized as follows:

- We present DeSCDT, a novel deep learning driven differential fuzz testing approach specifically targeting the emerging Solidity compiler. We distinguish DeSCDT being the first to pay attention to the misoptimzation defects due to the unique gas mechanism besides the direct and obvious crash-inducing bugs within the Solidity compiler.
- To advance the quality of compiler test case generation, a Transformer-based contract generator is learnt. It is adept at producing syntactically valid and diverse contracts, alongside the light-weight semantic encoding and clustering for seed contract selection, carefully devised generation strategies, fine-grained mutations and heuristic seed pool update. To the best of our knowledge, we are the first to explore integrating a deep generative model for fuzz testing the Solidity compiler.
- The experimental evaluations show remarkable capability of DeSCDT in generating syntactically valid contracts, as evidenced by a 90.8% compilation pass rate on the most widely used Solidity compiler version. It uncovers a substantial proportion of discrepancies between optimized and non-optimized versions compiled from identical contracts, attributable to misoptimization defects within the Solidity compiler. Six unique bugs that trigger direct crashing of the compiler during the contract compilation have also detected. The training dataset and source code of DeSCDT have been made publicly available at https://github.com/Xutp-F/DeSCDT.

The remainder of this paper is organized as follows. Section 2 outlines some foundational background knowledge and delineates the research motivation; Section 3 presents a high-level overview of DeSCDT as well as the detailed elucidation of the primary designs; Section 4 is dedicated to an extensive analysis and discussion of the experimental evaluations; Section 5 discusses related works, and Section 6 concludes the paper.

2 Preliminaries

2.1 Solidity Smart Contract

Smart contracts are Turing-complete programs living on a blockchain, triggered to execute business logic as programmed in response to user transactions. Of the prominent blockchain platforms that support contracts, Ethereum is the first and also the largest one, wherein the contract bytecodes are executed within a stack-based Ethereum Virtual Machine (EVM). To navigate away from the intricacies of programming on the low-level EVM opcodes, high-level languages are utilized to facilitate the development. In this landscape, Solidity distinguishes itself as the predominant contract developing language on Ethereum.

Due to the immutable nature of blockchain-stored data, once deployed, the contracts become no longer modifiable, presenting challenges when discrepancies between intended designs or security flaws are later identified. Even when the developers deploy an updated contract, its old problematic version still persists on the blockchain, unless a self-destruct function is provided [8]. Consequently, making sure no miscompilation issues are introduced during the contract compilation, i.e., the correctness of the compiler, becomes critical significant, as a single defect in the compiler affect all the contracts that it processed. However, Solidity being an emerging programming language, it has been undergoing continuous evolution and refinement since its inception, encompassing both the language and its accompanying compiler `solc`, to enhance the efficiency of the generated EVM bytecode as well as to rectify compiler bugs [9].

2.2 Gas Mechanism

Gas is the fuel that powers the Ethereum. Every computational action within Ethereum, including deploying and executing smart contracts, incurs certain gas charges. This unique gas mechanism allows Ethereum to compensate those who provide computational resources (i.e., the miners), and also fortifies itself against abuse by deterring malicious actors from overwhelming the network with deceitful actions. The specific gas fee is computed by gas price $\times$ gas cost, where gas price fluctuates according to the Ethereum market, and the gas cost refers to the amount of gas consumed. The gas cost of the EVM opcodes varies. Some operations are cheap, some, like ADD, AND, EQ, and POP, are inexpensive as they are solely stack operations. In contrast, others, such as SSTORE (for storage updates) and CREATE (for creating a new smart contract), are costlier. Since gas is effectively tied to monetary value, specifically in Ether (Ethereum's native currency), a contract that requires less gas for deployment or execution while delivering the same functionality is more desirable [5, 6].

In response to this peculiarity, the Solidity compiler offers an optimization module, which simplify complicated expressions within the code, to reduce the bytecode size that lowers the contract deployment gas cost, and refine the comprising opcodes that decreases the gas fees for contract invocation. The following example taken from the official document [10] of Solidity illustrates one such simplification operation. It prunes away the *if* branch as the condition in line 3 always evaluates to *false*, producing the semantically equivalent yet more computational efficient and concise code "`data[7] = 9; return 1;`".

```
1.  uint x = 7;
2.  data[7] = 9;
3.  if (data[x] != x + 2) // the condition is never true
4.      return 2;
5.  else
6.      return 1;
```

2.3 Differential Testing

Differential testing [11] is a method used to uncover subtle semantic bugs, which are less obvious than explicit errors such as crashes or assertion failures. It employs at least two programs of the same functionality as cross-referencing oracles, to deal with the situations where standard test oracles are unavailable. By comparing their outputs across many inputs, differences in the behaviors of these programs when given the same input are identified as potential bugs.

To identify potential misoptimization defects within the Solidity compiler, we adopt a variant of the standard differential testing. It regards the non-optimized and optimized versions of the contracts, compiled from the same source code, as the cross-referencing oracles. If these two compiled versions exhibit divergence in their deterministic computational results under the same inputs, or their opcode size and gas consumption violate Solidity compiler's design intent, it then suggests the presence of a misoptimization bug.

The following provides a simplified contract case generated by DeSCDT. It revealed a misoptimization defect within the Solidity compiler's optimization process, that could lead to inconsistent outputs in differential testing. Particularly, the function "`isEqual()`" shows divergent outputs of *true* or *false*, when executed from its bytecode versions compiled with the optimization switch on and off, respectively. Examination of the disassembled opcodes from both compilations revealed that the functions f1 and f2 were combined into one when the optimization switch is on. This suggests that the compiler's optimization module generally focuses on function statements rather than identifiers, thus leading to an improper merging of functions with distinct names but behavior-consistent body, in an effort to minimize the opcode size.

```
1. contract C {
2.    function f1() public {}
3.    function f2() public {}
4.    function isEqual() external returns(bool) {
5.       return f1 == f2;
6.    }
7. }
```

3 Design of DeSCDT

Figure 1 exhibits the overall structure of DeSCDT, encompassing three principal modules. The offline training module, is charged with the task of cultivating a reliable generative model on the basis of the Transformer architecture, with a copious corpus of diverse contract codes collected from the wild after pre-processing. In contract generation module, the aforementioned trained generator is leveraged to manipulate the seed contracts, which are carefully picked from the wild with light-weight semantic-encoding and clustering, under the guidance of the predefined generation strategies, to steadily yielding syntactically valid contract codes. On this basis, fine-grained mutations are further enforced in this module to augment the diversity of the resulting contracts. Subsequently, in the compiler differential testing module, the contracts are compiled with the `solc` compiler, where failed compilations that trigger compiling crashes or timeout are directly reported. For the contract cases that successfully pass through the compilation, they are further fed into a localized EVM for bytecode execution. Information including gas consumption, opcode sizes, and computational results between the optimized and non-optimized versions of each contract are observed and compared, for uncovering the implicit misoptimization defects. By the end of each fuzzing loop, the interestingness of the newly generated contract is estimated based on the abnormal behaviors it triggered. The least interesting contract in the seed pool is then substituted with the newly generated more interesting one, so as to gradually and heuristically guide later fuzzing loops towards generating contracts that are more likely to trigger bugs.

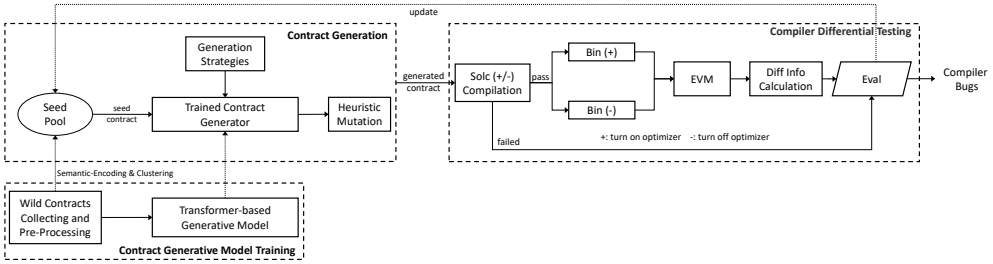

Fig. 1 The basic workflow of DeSCDT.

3.1 Contract Generative Model

In this section, we present the details of the Transformer-based contract generative model, including the construction of a large-scale contract corpus and its processing, as well as the training of the model.

3.1.1 Contract Collecting and Pre-Processing

To ensure the quality of the contract corpus, which is important for training a robust Transformer model, we collect real-world contracts from Etherscan, the preeminent

Ethereum explorer, which offers access to a vast array of diverse smart contracts. Since the contract can only be effectively retrieved from Etherscan on the availability of their respective deployment addresses, we firstly query the public Ethereum Cryptocurrency database hosted on Google BigQuery to obtain the addresses of all the unique contracts showing distinct EVM bytecodes. Then, We iteratively query Etherscan with the previously obtained addresses for obtaining the corresponding source code of each contract.

Subsequent to this acquisition phase, we undertake a slight preprocessing of the raw source code for each contract. This step involves the removal of comments and the version pragma directive from the code, as well as the consolidation of consecutive white spaces between code elements into a single space. Simplifying the contracts in such a way presents enhanced opportunities for the subsequent model to assimilate and comprehend the syntactic aspects inherent to the Solidity programming language. Finally, The contracts that show distinct md5hash values calculated on their format-unified source code, then comprise the dataset for training the model.

3.1.2 Training of the Transformed-based Generation Model

Transformer [12] is the de facto most advanced and widely adopted neural network architecture, with which diverse models, including the emerging notable large language models like ChatGPT and Llama, have been built. Owing to its more rational network structure and the powerful multi-head self-attention mechanism, it generally surpasses other neural network structures such as RNNs in capturing long-range dependencies more effectively. In light of this, we also utilize the Transformer, particularly its encoder-decoder configuration, to develop a robust model for generating contracts.

Particularly, we approach the smart contract source code as a sequential array of characters, denoted as s. By employing a sliding window with a fix span of d characters, the contract is segmented accordingly. The processing produces a set of input-output pairs $\langle x_i, y_i \rangle$, where $x_i = s[i : i + d]$, $y_i = s[i + d + 1]$ and $0 \leq i < \text{len}(s) - d$, to be used as the training corpus of the Transformer model.

The specific Transformer architecture we adopted is composed of six layers each in the encoder and decoder, with a capacity to handle up to 512 tokens in sequence. 90% of the pairs in the corpus are taken as the training samples, while the remaining 10% serve as validation samples. We performed the model training on a Linux server equipped with two RTX3090 GPUs for 300 epochs employing the Adamw optimizer. We initiated the learning rate at 1e-3, reducing it by a factor of 10 if there was no improvement in the validation loss for three consecutive epochs. The model at the epoch that shows the lowest validation loss is selected as the final Solidity contract generation model.

3.2 Smart Contract Generation

This section elaborates on the process of generating contracts by jointly leveraging the trained generative model and fine-grained mutations. The designed generation strategies and the selection of initial seed contracts are also discussed.

3.2.1 Contract Seed Pool Construction

The selection of seed is considered beneficial in boosting the fuzz testing efficacy [1, 13]. Therewith, we perform semantic encoding and clustering on the contract dataset constructed in Section 3.1.1, to purposely construct an initial contract seed pool. Compared with the way of randomly picking contract instances as the seeds, this enables the contract generator have sufficiently diverse seed contracts to manipulate on.

Specifically, we firstly utilize SmartEmbed [14] to encode each contract in the dataset into a 128-dimensional numerical vector. The basic idea of SmartEmbed is to leverage fastText to learn a dictionary of static token embeddings, and sum up the embeddings of all tokens that make up a contract to achieve its semantic representation. Following this, we perform clustering on these semantic encodings, grouping the contracts into clusters such that each cluster only consists of contracts with high similarity. Algorithm 1 outlines the detailed process of selecting the seed contracts. It takes in a contract dataset $\mathcal{D}$, a similarity threshold ϵ, and a certain version Solidity compiler *solc*, and outputs a contract seed pool $\mathcal{P}$.

In first part of the algorithm, all the contracts are grouped into a collective of clusters denoted as $\mathcal{C}$, which initially contains just one cluster $\mathcal{C} = \{c_0\}$. For each contract in the dataset then, we either append it to an existing cluster in $\mathcal{C}$ or form a new cluster with it. During each specific iteration, the semantic encoding of the contract D_i is computed with the SmartEmbed model first. Its similarity with respect to each cluster $c_j \in \mathcal{C}$ is then assessed with a function designated as `avgSim`, which compute an average score between the encoding of D_i and all the contracts within cluster c_j, utilizing the Euclidean-based similarity metric adopted in SmartEmbed. Then, the contract is attributed to the cluster $\tilde{c}$ that exhibits the highest similarity exceeding the threshold ϵ. In the absence of such a cluster in $\mathcal{C}$, a new cluster that exclusively contains D_i is established and appended to $\mathcal{C}$. Upon the completion of this iterative clustering process, we pick one contract that can be successfully compiled by the compiler at random from each cluster, to make up the seed contract pool.

3.2.2 Generation Strategies

Similarly as in DeepFuzz [15], we manipulate the contracts in the seed pool to derive new contacts. This is achieved by generating new statements with the trained generative model on prefixing sequences taking from the seed contract, and inserting them or replacing existing statements with them. However, we restrict the generation locations to be always within functions, by identifying from the concrete syntax tree (CST) of the contract all root nodes of its constituent function units and randomly selecting locations that are within their corresponding dominated sub-trees. The underlying rationale for this restriction is the recognition that the state variables outside of functions are likely to be used across multiple functions. Therefore, manipulating these state variables using the generative model has a high probability of rendering the entire contract syntactically invalid.

Under this premise, the following generation strategies are explored for the current study, including:

Algorithm 1 Semantic Clustering based Seed Selection

Input:

D: the wild contract dataset
ϵ: the similarity threshold with a default value of 0.9
solc: a certain version Solidity compiler

Output:

$\mathcal{P}$: the pool of seed contracts
$vec = \text{encode}(D_0)$ ▷ encode the contract with SmartEmbed
$c_0 = \{\langle D_0.addr, vec\rangle\}$, $\mathcal{C} = \{c_0\}$, k
for $i = 1$ to $\text{len}(D) - 1$ **do**
    $vec = \text{encode}(D_i)$
    $\tilde{c} = \text{argmax}_{c_j \in \mathcal{C}} \text{avgSim}(vec, c_j)$
    $e = \langle D_i.addr, vec\rangle$
    if $\text{avgSim}(vec, \tilde{c}) > \epsilon$ **then**
        $\tilde{c} = \tilde{c} \cup e$
    else
        $k+ = 1$, $c_k = \{e\}$, $\mathcal{C} = \mathcal{C} \cup \{c_k\}$
    end if
end for
$\mathcal{P} = \{\}$
for each c in C **do**
    repeat
        $e = \text{randChoice}(c)$
        $sc = \text{getContract}(D, e.addr)$
    until $passCompile(solc, sc)$
    $\mathcal{P} = \mathcal{P} \cup \{sc\}$
    return $\mathcal{P}$
end for

- STG_1: The newly generated code snippet is directly inserted righted after the prefixing sequence. Meanwhile, the rest characters of the original single statement that the tailing characters of the prefixing sequence resides in is deleted, and the other statements remain unchanged.
- STG_2: Multiple different generation locations are randomly picked from the same function in the contract. At each selected location, new code snippet is independently generated by the trained model, conditioned on the corresponding prefixing sequence, and then inserted back following STG_1.
- STG_3: The same number of statements in the original contract that subsequent the prefixing sequence is substituted with the newly generated code snippet.

3.2.3 Fine-grained Mutation

Build upon the generated contract, DeSCDT further integrates a suite of fine-grained mutators, to intensify the code distortion for better triggering the corner bugs. Details of each code mutators, including a concise summarization of the primary code elements

it operates on, a simple illustrative example, and a brief explanatory note, are listed in Table 1. While the potential exists for incorporating more sophisticated mutators, the current selection was made with the intent of minimizing the risk of syntactic invalidity in the altered code. During the mutation phase, one out of the six types of mutators is randomly chosen in each iteration to alter the designated code elements. This process is iterated randomly between one to six times.

Table 1 A brief description of the code mutators adopted

Mutator	Mutation Object	Illustrative Example	Supplemental Description
$\mathcal{M}_1$	modify the boundaries within the loops	$for(uint\ i = 0; i < 10; i++) \rightarrow for(uint\ i = 0; i < 10 + a; i++)$	a denotes an integer value between 1 and 100
$\mathcal{M}_2$	mutate the conditional operators	$if(a > 1) \rightarrow if(a <= 1)$	the available operators used for permutation include: $>=, <=, >, <, ==, !=$
$\mathcal{M}_3$	mutate the arithmetic operators	$uint\ a = 1 + b \rightarrow uint\ a = 1 * b$	the available operators used for permutation include:$+, -, *, /, \%, **$
$\mathcal{M}_4$	mutate the logical operators	$c = !a\ \&\&\ b \rightarrow c = !a\ ||\ b$	the available operators used for permutation include:$\&\&, ||, !$
$\mathcal{M}_5$	modify the type of local variables	$uint\ a \rightarrow int16\ a$	use the fewer digits version of the same type for subsitution
$\mathcal{M}_6$	modify the control structures	$while(a < 10)\{a += 1;\} \rightarrow while(a < 10)\{a += 1; break;\}$	insert *break* or *continue* into the body of control structures

3.3 Unified Testing Harness

With the newly generated contract, DeSCDT attempts to trigger two main types of Solidity compiler defects. The much obvious and of immediate interest compiler crashes or build timeout errors, where a contract case makes the compiler either directly crash or simply can not finish the compilation within a limited time. The other kind is the much implicit misoptimization defects, where inconsistent behaviors violating the compiler designing rules are observed between the optimized and non-optimized versions of the same contract that pass through the compilation phase. DeSCDT sets up a unified test harness that accommodate contract compilation and differential testing for revealing these bugs.

Particularly, in the contract compilation phase, DeSCDT attempts using the compiler `solc` to build two bytecode versions of the input contract, with the optimization parameter switching on and off, respectively. If either of these two compilations ends with reporting "*InternalCompilerError*" that crashes `solc`, or exceeds the limited compilation time, then a compiler defect have been detected. For simplicity, this kind of bugs is termed as BT1.

For the contracts that successfully compile without obvious errors, differential testing is then enforced to run, observe and compare their conflicting aspects, to uncover potential misoptimization defects. This process involves executing each compiled contract within a localized EVM in terms of each its constituent function units iteratively. The types of the formal parameters for each function are firstly collected via the command "*solc* xxx.sol –hashes -o", which returns the list of function prototypes. Next, actual parameters for each function are instantiated with a predefined array of common and extreme values for each data type. Thereafter, the EVM interface is called to execute the bytecodes of each function with its respective actual parameters. During this process, the contract aspects regarding its gas consumption, opcode sequence sizes, and computational outcomes are profiled, with which differential analysis is to be conducted between each compiled contract pair.

Following summarizes the specific misoptimization defects defined on these collected contract aspects. For these defects, either semantic divergences are observed

between the compiled contract pair, or the intended optimization goal of simultaneously reducing both the code size and gas consumption is violated when the optimization switch is on.

- BT2: refers to the case where divergent outputs are observed between the optimized and non-optimized bytecodes fed with the same inputs.
- BT3: refers to the case where the gas consumption of the optimized version is larger than that of the non-optimized version, while the opcode size is smaller.
- BT4: refers to the case where the opcode sequence size is larger than that of the non-optimized version, while the gas consumption is smaller.
- BT5: refers to the case where both the gas consumption and opcode sequence size of the optimized version exceed those of the non-optimized version.

3.4 Seed Pool Update

DeSCDT maintains a dynamic seed pool, by appending to the pool newly generated interesting contracts and retiring the less interesting ones as the fuzzing iteration goes, to heuristically increase the ratio of effective fuzzing loops that are more likely to trigger bugs. The interestingness of a contract is comprehensively estimated in accordance with both its ignored time in the seed pool and the abnormal behaviors observed during the compilation and differential testing phases.

Formally, for a contract $\tilde{c}$ newly derived from an existing contract c in the pool, its quantitative interesting score is computed as:

$$\begin{aligned} \text{IScore}\,(\tilde{c}) = {} & w_1 \times \text{sign}(BT1, \tilde{c}) \\ & + w_2 \times \text{sign}(BT2, \tilde{c}) \\ & + w_3 \times \text{gasDiff}(\tilde{c}_+, \tilde{c}_-) \\ & + w_4 \times \text{opDiff}(\tilde{c}_+, \tilde{c}_-) \end{aligned} \tag{1}$$

$$\text{sign}(BT1, \tilde{c}) = \begin{cases} 1, & \text{if } \tilde{c} \text{ triggers BT1 bugs} \\ 0, & \text{else} \end{cases} \tag{2}$$

$$\text{sign}(BT2, \tilde{c}) = \begin{cases} 1, & \text{if } \tilde{c} \text{ triggers BT2 bugs} \\ 0, & \text{else} \end{cases} \tag{3}$$

$$\text{gasDiff}(\tilde{c}_+, \tilde{c}_-) = \begin{cases} \dfrac{\text{gas}(\tilde{c}_+)}{\text{gas}(\tilde{c}_-)}, & \text{if } \text{gas}(\tilde{c}_+)/\text{gas}(\tilde{c}_-) > 1 \\ 0, & \text{else} \end{cases} \tag{4}$$

$$\text{opDiff}(\tilde{c}_+, \tilde{c}_-) = \begin{cases} \dfrac{\text{len}(\tilde{c}_+)}{\text{len}(\tilde{c}_-)}, & \text{if } \text{len}(\tilde{c}_+)/\text{len}(\tilde{c}_-) > 1 \\ 0, & \text{else} \end{cases} \tag{5}$$

where, $\text{sign}(\cdot,\cdot)$ is a signal function indicating whether the contract triggers certain kind bugs in the current fuzzing iteration; $\text{gasDiff}(\cdot,\cdot)$ and $\text{opDiff}(\cdot,\cdot)$ estimate the degree of deviation from the intended optimization goals with respect to the contract's optimized and non-optimized bytecodes' gas consumption and opcode sequence length,

respectively; the w_is are weights signifying the contributions of each partial part to the total interesting score, where DeSCDT adopts a value of 3 for w_1 and 1 for the rests.

Algorithm 2 depicts the process of updating the contract pool by the end of each fuzzing loop. Firstly, the interesting scores of existing contracts within the seed pool are updated iteratively, with increasing the scores of the contracts not being selected in the last round fuzzing loop, while decreasing the score of the contract just being manipulated. In such a way, seed contracts that have not been selected for a long time could gain higher chance to be selected in the next fuzzing loop, ensuring that potentially interesting contracts are not missed before they are replaced out of the pool. During the iteration as outlined from line 6 to line 16 in the algorithm, the least interesting contract, which shows the minimum interesting score, is also found out. Subsequently, in line 17, the interesting score of the newly generated contract $\tilde{c}$ is calculated according to Equation 1. If the score of $\tilde{c}$ surpasses the minimum interesting score found earlier, then the corresponding least interesting contract in the pool is replaced with $\tilde{c}$.

Algorithm 2 Heuristic Seed Pool Update

Input:

1: $\mathcal{P}$: the pool of seed contracts
2: $\tilde{c}$: the newly generated contract that just goes through the unified testing procedure of current fuzzing iteration
3: W: the weight array specifying the values of w_1 to w_4

Output:

4: $\mathcal{P}$: the updated seed contract pool
5: min_score = INF, min_idx = 0
6: **for** $i = 0$ to len($\mathcal{P}$) $- 1$ **do**
7: **if** $\mathcal{P}_i$ is not c **then**
8: $\mathcal{P}_i$.score += 1
9: **else**
10: $\mathcal{P}_i$.score = max(0, $\mathcal{P}_i$.score-1)
11: **end if**
12: **if** $\mathcal{P}_i$.score $<$ min_score **then**
13: min_score = $\mathcal{P}_i$.score
14: min_idx = i
15: **end if**
16: **end for**
17: sr = IScore $(\tilde{c}, W)$ ▷ calculate the interesting score of $\tilde{c}$
18: **if** sr $>$ min_score **then**
19: $\mathcal{P}_i = \tilde{c}$ ▷ substituting the least interesting seed contract in the pool with $\tilde{c}$
20: $\mathcal{P}_i$.score = sr
21: **end if**

Table 2 Performance of DeSCDT in generating syntactic valid contracts.

Model	STG_1	STG_2	STG_3	STG_1^+	STG_2^+	STG_3^+	STG_1^-	STG_2^-	STG_3^-
Bi-GRU	81.4%	**84.6**%	80.2%	80.9%	84.0%	79.8%	62.5%	68.7%	65.9%
Bi-LSTM	82.3%	**85.2**%	81.7%	81.8%	84.5%	81.3%	63.7%	70.1%	67.1%
Transformer	87.6%	**90.8**%	86.8%	87.1%	90.4%	86.3%	68.9%	75.1%	72.3%

4 Experimental Evaluation

In this section, we present the experimental evaluation details, answering the following Research Questions (RQs):

- RQ1: **Performance in generating valid contracts**. How does DeSCDT behave in generating syntactic-valid smart contracts?
- RQ2: **Performance in boosting the testing coverage.** Can DeSCDT effectively generate diverse contracts that continuously improve the testing coverage?
- RQ3: **Bug detection capability**. How about DeSCDT's capability in revealing Solidity compiler bugs?

4.1 Implementation and Experimental Settings

DeSCDT is implemented mainly with Python and Java. The construction and the training of the contract generation model are implemented on top of the PyTorch framework, while the contract generation point localization and fine-grained mutations, which require manipulating on the concrete syntax tree, are implemented using Java. To obtain the concrete syntax tree of a contract, we use ANTLR[1] to build the parser with the grammar for Solidity[2]. The fuzzing experiments are conducted on Solidity 0.4.24, the most widely used version as investigated in the existing work [16]. For estimating the test coverage performance, we leverage `afl-gcc` and `gcov` for instrumenting the solc compiler and the specific profiling of code coverage metrics, respectively.

4.2 RQ1. Performance in Valid Contract Generation

This section evaluates the capability of DeSCDT in producing syntactic valid contracts, which has been confirmed crucial for effective compiler fuzzing [15, 17]. Therewith, we run the fuzzing loops of DeSCDT until 10,000 contracts have been generated, and deem the ones that pass through the compilation as syntactic valid. Table 2 summarizes the pass rates, which are calculated by dividing the quantity of the successfully compiled contracts by the quantity of all the generated ones under each corresponding alternative setting of DeSCDT.

As shown by the values in the first three columns, where merely the learnt generative model is utilized to yield new contracts, the models adopting STG_2 as their generation strategy outperform those using the other two kind strategies. Besides, when comparing across different neural architecture backbones, the generative model with Transformer backbone exhibits the best performance, with its pass rate reaches

[1] https://github.com/antlr/antlr4/tree/master
[2] https://github.com/antlr/grammars-v4/tree/master/solidity

90.8% when using STG_2 as its generation strategy. This indicates the sufficiency of our learnt Transformer-based generative model in synthesizing syntactically valid contracts, as well as its superiority over the alternative neural structures in better comprehending the grammatical nuances of the Solidity language.

Further, we investigated the impact of fine-grained mutation on the syntactic validity of newly generated contracts. The STG_x^+ columns exhibits the pass rate achieved when the contract generation process is completed in its entirety, encompassing both the use of the contract generative model and the fine-grained mutators. The results reveal that these pass rates are nearly identical to those in the first three columns. This indicates that the carefully picked mutators employed in DeSCDT do not necessarily compromise the syntactic validity of the generated contracts. At the same time, they could offer additional opportunities to alter the code in ways that expose elusive bugs.

As explained in Section 3.2.2, regardless of the particular generation strategy employed with the generative model, we consistently confine the generation to occur within the functions, with the aim of increasing the probability of synthesizing syntactic correct contracts. To verify the significance of this limitation, we also run DeSCDT by liberating it from the restriction. The results, as shown by the significantly reduced values in the STG_x^- columns, indicate that maintaining this restriction in DeSCDT is crucial. Without it, there is an average decline of approximately 16.3% in the pass rate.

Answer to RQ1: DeSCDT can effectively produce syntactically correct contracts for fuzz testing the Solidity compiler, with the Transformer-based generator achieving the highest pass rate under generation strategy STG_2. Integrating the generative model further with the carefully picked fine-grained mutators hardly affect the compilation pass rate, while releasing the generation location restriction incurs significant degradation on the pass rate of DeSCDT.

4.3 RQ2. Performance in Boosting Test Coverage

4.3.1 Diversity of the Generated Contracts

The significance of test case diversity in uncovering software bugs has long been acknowledged in the software testing literature [18]. To evaluate whether DeSCDT can effectively generate diversified contract code, we run DeSCDT for 24 hours and calculate the similarity between the newly derived contracts and their original ancestors. At each time node that corresponds to every 2 hours execution of the DeSCDT's fuzzing loops, 500 more new contracts are further generated and compared against their ancestral versions in the original contract seed pool. The SmartEmbed method, which has been discussed in Section 3.2.1, is utilized for the specific similarity calculation between the contracts.

Figure 2 depicts the experimental results, where the horizontal axis outlines the time nodes and the vertical axis marks out the averaged similarity scores. It can observed that, the similarity scores consistently decrease as the fuzzing duration increases. Especially, at the last time node, the similarity scores that corresponds to

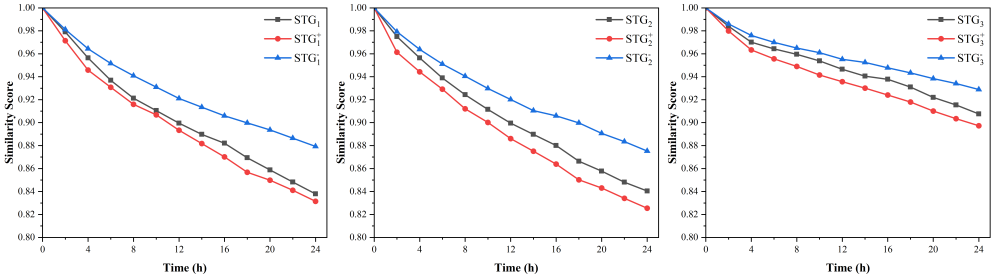

Fig. 2 Diversity of newly generated contracts at different time nodes

generation strategies STG_2 and STG_2^+ drop to 0.841 and 0.825, respectively. This trend suggests that the contracts generated by DeSCDT progressively diverge from their original versions, showcasing DeSCDT's effectiveness in producing diversified contract cases. Besides, the steeper descent curve corresponding to the results of STG_2^+ indicates the added advantage of incorporating fine-grained mutations to enrich code diversity. Additionally, running DeSCT without the within-function restriction does not seem to yield significantly more diverse contracts compared to those generated with the restriction.

4.3.2 Test Coverage Performance

Code coverage is another important indicator reflecting the sufficiency of software testing. Thereby, we run DeSCDT for 24 hours with the different settings, and statistic three widely used coverage metrics, including line coverage, branch coverage and function coverage, during this process.

Figure 3 illustrates the variation tendency of these coverage metric values obtained under different DeSCDT settings. The steadily ascending curves indicate an increasing exploration of Solidity compiler code units as the fuzzing duration extends. Among the evaluated settings, DeSCDT adopting STG_x^+s generally outperforms the others with better coverage metrics at each time node, despite the differences are not that obvious. This suggests the superiority of integrating the generative model with fine-grained mutations in exploring the deeper program states. Besides, the curves corresponding to DeSCDT with the STG_2^xs exhibit steeper upward trend compared to others. This aligns with the findings in Figure 2 that STG_2^xs help yield more diverse code. Especially, DeSCDT with STG_2^+ achieves the highest coverage of 61.89%, 53.82%, and 32.97% in terms of the line, branch and function coverage metrics, respectively.

Moreover, to give a more intuitive sense of the code coverage improvements brought by the newly generated contracts, we compute for each DeSCDT setting a coverage gain with respect to each coverage metric. The gain with respect to each metric is calculated with $\text{covGain} = (\mathcal{V}_{aug} - \mathcal{V}_{org})/\mathcal{V}_{org}$, where $\mathcal{V}_{org}$ denotes the coverage values obtained by feeding to the compiler merely the contracts within the original seed pool, and $\mathcal{V}_{aug}$ denotes the coverage values by additionally feeding with all the new contracts generated during the fuzzing iterations.

Table 3 Coverage gains obtained with DeSCDT in terms of the coverage metrics

Metrics	STG_1	STG_2	STG_3	STG_1^+	STG_2^+	STG_3^+	STG_1^-	STG_2^-	STG_3^-
Line	6.93%	11.24%	6.33%	8.24%	12.53%	7.82%	4.96%	9.36%	5.36%
Branche	5.57%	8.76%	5.33%	7.09%	9.91%	6.71%	4.24%	7.52%	5.04%
Function	4.73%	6.83%	4.38%	6.51%	7.63%	5.65%	3.33%	5.53%	3.65%

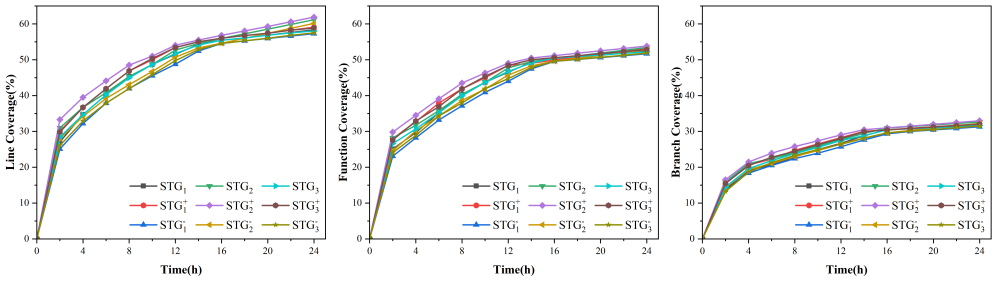

Fig. 3 Trend of test coverage in terms of the fuzzing timeline

Table 3 summarizes the corresponding gains. As the values show, DeSCDT adopting STG_x^+ generally win over the rest settings, again highlighting the merits of augmenting the generative model with fine-grained mutations. Particularly, within 24 hours, DeSCDT working with STG_2^+ achieves 12.53%, 9.91% and 7.63% improvements in terms of the line, branch and function coverage metrics, respectively.

Answer to RQ2: DeSCDT behaves highly effective in producing a range of diversified new contracts, contributing to the progressive increase in code coverage of the tested Solidity compiler. Besides, the DeSCDTs which combine the generative model and fine-grained mutations stands out the alternatives in their ability to generate the most diverse contracts, effectively testing a broader spectrum of the compiler's code units.

4.4 RQ3. Bug Detection Capability

As observed from the evaluations in the preceding sections, DeSCDT with STG_2^+ in overall outperforms its alternatives, in producing new contracts that not only have a high pass rate but also exhibit sufficient diversification. In consideration of this, we determined running DeSCDT under STG_2^+, and conduct experiments in this section to evaluate its capability in compiler bug detection.

4.4.1 Results on Compile-time Defect Detection

By the time of writing, after approximately three months execution, DeSCDT has successfully identified six unique bugs of BT1, which cause notable crashing of the Solidity compiler when fed with specific contracts. Following elaborates on two of

these bugs, including their specifics and the corresponding simplified contract cases that trigger them.

The first is a bug associated with the mapping feature supported in the Solidity language, which can be triggered with the listed code snippet below. Compiling the contract with the optimization either on or off could crash the compiler, reporting an internal compiler error with the message "Invalid non-value type for assignment". The root cause lies in the code between line 4 and 7 generated by DeSCDT, where mapping type data selected according to the parameter are appended to the tail of a mapping type array. The incomprehensive consideration and handling to such kind rare use cases of the mapping structure in the compiler's code implementation cause its crashing.

```
1. contract C {
2.    mapping(address=>unit)[] a;
3.    function f(uint[] x) public {
4.       mapping(address=>uint)[] b;
5.       for (uint i = 0; i< x.length; i++) {
6.          mapping(address=>uint) t = a[x[i]];
7.          b.push(t);
8.       }
9.    }
10. }
```

The second bug to discuss is relevant to Solidity compiler's inadequate handling of ambiguous identifiers appearing in the contract code. In the following contract case, DeSCDT generated a return statement containing a weird sub-expression of `a(c)`. It confuses the compiler in comprehending the precise type of the identifier as well as the intention of this expression, i.e., to call function `a()` or to reference variable `a`. The compilation of this contract code, regardless of the optimization switch on or off, also crashes the compiler with issuing an error message of "Requested type not present".

```
1. contract C {
2.    uint public a;
3.    function a(uint c) public returns (uint256)
4.    {
5.        return 1 * 10 ** a(c);
6.    }
7. }
```

4.4.2 Results on Misoptimization Bug Detection

During the fuzz testing duration, we've observed numerous contract cases that activate the misoptimization bugs. To have a glance of this situation, Figure 4 depicts the distribution of the contract cases produced during a 24-hour running of DeSCDT, categorized by the types of bugs they trigger.

Of the 14,237 newly generated contract cases that are differentially tested, 5,324 (about 37.4%) contracts successfully triggered at least one of the four types of misoptimization bugs. The majority, which is about 37%, of these bug-triggering cases, occurred between BT3 and BT5, where the compiler's intended optimization objectives of simultaneously reducing both the code size and gas consumption is violated, when tested with these specific contracts. 51 cases behave abnormally with producing divergent outcomes under consistent inputs, indicating potential BT2 bugs within the

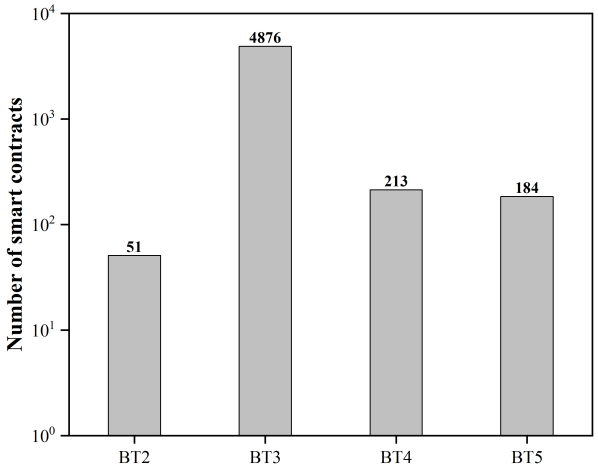

Fig. 4 Quantity distribution of the misoptimization bug triggering contracts.

optimization module. Besides, since bugs of BT2 do not conflict with other misoptimization bugs, a number of the generated contract cases are found to trigger them both. These findings point to concerns about the reliability of the Solidity compiler's optimizer in effectively optimizing contracts as intended by users.

To better understand the issue of gas consumption misoptimization, which is economic-critical both to the contract developers and the users, we further delve into the opcodes of the generated contracts, whose gas consumption of the optimized bytecode is larger than that of its non-optimized version. Figure 5 presents the differences on their opcode frequency distribution. The x-axis denotes all the involved opcodes, and the y-axis shows the overall frequency differences, calculated by subtracting the occurrence of each opcode in the optimized version from its count in the non-optimized version. As the results suggest, the optimization leads to a notably different opcode distribution. Especially, for these gas consumption misoptimization triggering contracts, their optimized bytecode appear to contain opcodes that consume more gas[3], such as "EXP", "SLOAD", "JUMPI" "EXTCODESIZE" and "SHA3". We hope the findings on opcode distributions in both optimized and non-optimized versions of these contracts could aid in redesigning and improving the underlying algorithms in the compiler's optimization module.

Answer to RQ3: DeSCDT is capable of revealing both compile-time and misoptimization bugs within the Solidity compiler. The high number of cases where misoptimization bugs are triggered suggests the immaturity of the optimization module. Considerable efforts are still required to refine and enhance its underlying algorithms to finally meet the intended objectives of the optimization process.

[3]https://ethereum.org/en/developers/docs/evm/opcodes/

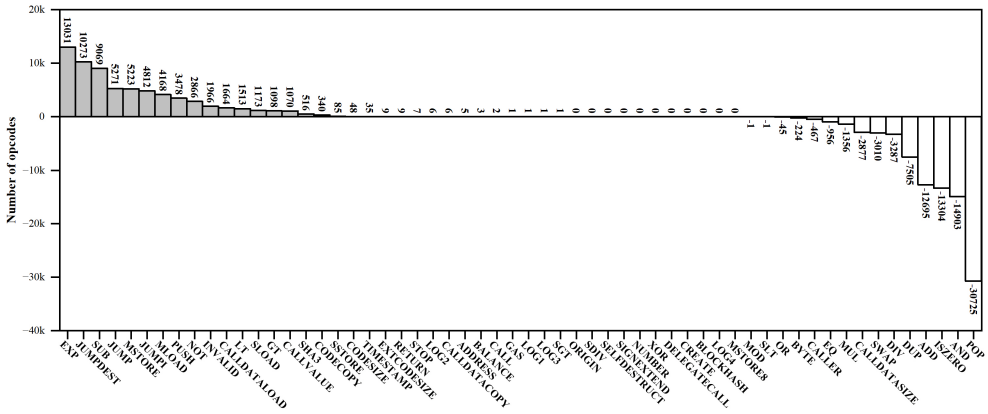

Fig. 5 Differences on the opcode frequency distribution of the gas consumption misoptimization triggering contracts

5 Related Work

Given the pivotal role of compilers in the software ecosystem, extensive researches [3, 19–21] have been conducted on fuzz testing them for validating and enhancing their correctness. Depending on the differences in the way of generating test inputs, which is considered to be the most influential part that directly controls the likelihodd of triggering bugs [1], existing works can majorly be categorized into generation-based, mutation-based and learning-based fuzzers.

5.1 Mutation-based Approaches

Mutation-based fuzzers [4, 22–25], or model-less fuzzers, produce new inputs cases by performing random or heuristic code mutations on existing test cases. The effectiveness of such fuzzers largely rely on the quality of the seed pool, the richness of mutators, as well as the ways to guide the mutation and selection of interesting cases. Generally, if a newly generated test case triggers previously untested program states, measured by metrics such as new paths or statements, it will be considered interesting and saved for later round mutations.

AFL [22], which performs random bitflip mutation and adopts coverage feedback or target sites [26] as guidance, stands as a prominent example of this category. However, without comprehending the grammar aspects of the inputs, these general-purpose fuzzers [22, 26] become much less effective in producing highly structured syntactic correct programs for fuzz testing the compilers. As such, approaches like LangFuzz [27], JavaTailor [23] and CodeAlchemist [25] break down test programs into code fragments and weave ingredients extracted from historical bug-revealing test programs into them to fulfill mutation. Other methods like Superion [28], DIPROM [24] and Fuzzilli [29] rather perform mutation either on the AST or the lifted IRs [4] to create syntactic valid test programs. Besides these works that primarily focus on fuzzing conventional

compilers, like GCC and LLVM for C/C++, Javascript engines [30] and the JVMs [31], EVMFuzz for the first time explore EVMs defects with a mutation-based contract generator. Compare with this work, we blend DL-based generative model with code mutation to produce varied syntactic valid contracts written in the emerging Solidity programming language. Also, we target the bug detection of Solidity compiler rather than EVMs.

5.2 Generation-based Approaches

Generation-based or model-based compiler fuzzers [32, 33], are distinguished by their use of models that implicit the grammar rules of the target language. This enables them to generate syntactic highly correct test programs from scratch without relying on any existing test programs.

CSmith [32], which targets fuzz testing the C compilers including GCC and LLVM, is a representative method in this category. It hardcodes a subset of C grammar to ensure valid test program generation and maintains a probability table to choose code elements to enhance the diversity of the programs. To avoid the testing saturation issue, YARPGen [34] generates expressive C and C++ programs with a novel mechanism skewing the probability distribution systematically. Besides fuzzing the C/C++ compilers, CSmith also inspired the designing of fuzzers targeting other compilers like CUDA C/C++ compiler [35], and compilers for other languages like OpenCL [36] and Simulink [37]. Despite the CSmith-like fuzzers are adept at creating highly valid test programs, their approach of hardcoding grammar lacks flexibility and requires extensive engineering efforts and language-specific domain knowledge for adapting to updated language characteristics or different languages.

5.3 Deep Learning-based Approaches

With the great success of deep learning (DL) driven program analysis researches, there are also some pioneering works [15, 17, 38, 39] that explore building deep learning models to generate programs for compiler fuzz testing. DeepSmith [17] is a trailblazer in applying deep learning for fuzz testing OpenCL compilers, utilizing an LSTM-based model to generate OpenCL programs. This model surpassed CLSmith, a traditional fuzzer implemented with extensive engineering efforts according to expert-defined grammars. DeepFuzz [15] also learns an LSTM-based seq2seq model, but uses it to create C programs for testing GCC and LLVM. However, it always generates new test programs based on the initially collected seed program pool. Without the designing of seed pool update process, the diversity of the newly produced programs and the code coverage they can achieve can be greatly limited. Similarly, Montage [39] trains an LSTM model to manipulate seed JavaScript programs with newly generated code snippets to fuzz test the JS engines. To improve the syntactic correctness of generated code, Dsmith [38] further incorporates a context attention mechanism into the LSTM to enhance its capability in capturing the long-range syntax dependencies in C programs. It achieved as high as 78.79% compilation pass rate, which is about 10.5% higher than the rate achieved by DeepFuzz with its basic LSTM model.

Compared with these DL-based compiler fuzzers, our method leverages the advanced Transformer architecture, renowned for its multihead attention mechanism. This choice significantly enhances the pass rate of the programs we generate. To add further diversity to the generated programs, our method introduces seed pool update phase to avoid always manipulating on the original seed programs as existing DL-based fuzzers do, and incorporates fine-grained mutations with the generative model. To the best of our knowledge, DeSCDT represents the first endeavor to apply deep generative modeling in creating Solidity smart contracts, specifically aimed at fuzz testing the emerging Solidity compiler.

6 Conclusion

This study has presented DeSCDT, an innovative DL-based fuzzer for testing the emerging Solidity compiler, a critical component in the development and deployment of smart contracts in the Ethereum blockchain system. DeSCDT's main component is a sophisticated contract generator utilizing the Transformer model. It incorporates advanced features like semantic encoding and clustering for initial seed pool construction, generation strategies with location restriction, fine-grained mutations, and strategic seed pool updates. These designs enable DeSCDT to generate Solidity smart contracts that are not only syntactically correct but also diverse in functionality. The extensive experiments conducted highlighted a remarkable compilation pass rate of 90.8% for the newly generated contracts with high diversity, which significantly contributing to increased code coverage. DeSCDT behaved adept at identifying both compile-time errors and misoptimization defects in the Solidity compiler. Six unique bugs that could cause the compiler to crash during contract compilation have been discovered during a three-month running of DeSCDT. Also, the numerous instances that expose optimization issues point to the immaturity of the optimization module in the compiler. This underscores the necessity for considerable efforts to refine and enhance the underlying algorithms to meet the optimization goals fully. Some of our future works include exploring other contract generation strategies, designing and incorporating more specialized mutators considering the unique language features of Solidity, as well as optimizing the seed pool update mechanism, to further enhance the bug revealing capability of DeSCDT for Solidity compiler fuzz testing.

Funding. This work was supported in part by the National Natural Science Foundation of China (62272387, 61702414), the Natural Science Basic Research Program of Shaanxi (2022JM-342, 2018JQ6078), the Youth Innovation Team of Shaanxi Universities "Industial Big Data Analysis and Intelligent Processing", the Special Funds for Construction of Key Disciplines in Universities in Shaanxi, and Graduate Innovation Fund of Xi'an University of Posts and Telecommunications (CXJJDL2022013).

Data availability. The dataset and source code implementation of the proposed method are available at the GitHub repository: https://github.com/Xutp-F/DeSCDT.

Author contributions. Zhenzhou Tian: conceptualization, methodology. Fanfan Wang: data curation, writing-original draft, software. Yanping Chen: validation, supervision. Lingwei Chen: methodology, writing-review and Editing.

Declarations

Competing interest. The authors declare no competing interests

References

[1] Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2021) https://doi.org/10.1109/TSE.2019.2946563

[2] Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W.: A systematic review of fuzzing techniques. Computers & Security, 118–137 (2018)

[3] Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing **53**(1) (2020) https://doi.org/10.1145/3363562

[4] Chen, Y., Zhong, R., Hu, H., Zhang, H., Yang, Y., Wu, D., Lee, W.: One engine to fuzz 'em all: Generic language processor testing with semantic validation. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 642–658 (2021). https://doi.org/10.1109/SP40001.2021.00071

[5] Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 442–446 (2017). https://doi.org/10.1109/SANER.2017.7884650

[6] Zhao, Z., Li, J., Su, Z., Wang, Y.: Gasaver: A static analysis tool for saving gas. IEEE Transactions on Sustainable Computing **8**(2), 257–267 (2023) https://doi.org/10.1109/TSUSC.2022.3221444

[7] Wang, Y., Li, K., Tang, Y., Chen, J., Zhang, Q., Luo, X., Chen, T.: Towards saving blockchain fees via secure and cost-effective batching of smart-contract invocations. IEEE Transactions on Software Engineering **49**(4), 2980–2995 (2023) https://doi.org/10.1109/TSE.2023.3237123

[8] Chen, J., Xia, X., Lo, D., Grundy, J.: Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. ACM Trans. Softw. Eng. Methodol. **31**(2) (2021) https://doi.org/10.1145/3488245

[9] List of Known Bugs: List of Known Bugs. https://docs.soliditylang.org/en/latest/bugs.html. Accessed Nov. 23rd, 2023 (2023)

[10] The Optimizer: The Optimizer. https://docs.soliditylang.org/en/latest/internals/optimizer. Accessed Nov. 23rd, 2023 (2023)

[11] M, M.W.: Differential testing for software. Digital Technical Journal **10**(1) (1998)

[12] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I.: Attention is all you need. Neural Information Processing Systems (2017)

[13] Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 861–875. USENIX Association, San Diego, CA (2014). https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[14] Gao, Z., Jiang, L., Xia, X., Lo, D., Grundy, J.: Checking smart contracts with structural code embedding. IEEE Transactions on Software Engineering, 2874–2891 (2021) https://doi.org/10.1109/tse.2020.2971482

[15] Liu, X., Li, X., Prajapati, R., Wu, D.: Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. Proceedings of the AAAI Conference on Artificial Intelligence **33**(01), 1044–1051 (2019) https://doi.org/10.1609/aaai.v33i01.33011044

[16] Tian, Z., Tian, J., Wang, Z., Chen, Y., Xia, H., Chen, L.: Landscape estimation of solidity version usage on ethereum via version identification. International Journal of Intelligent Systems, 450–477 (2022) https://doi.org/10.1002/int.22633

[17] Cummins, C., Petoumenos, P., Murray, A., Leather, H.: Compiler fuzzing through deep learning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (2018). https://doi.org/10.1145/3213846.3213848 . http://dx.doi.org/10.1145/3213846.3213848

[18] Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T.H.: Adaptive random testing: The art of test case diversity. Journal of Systems and Software **83**(1), 60–66 (2010) https://doi.org/10.1016/j.jss.2009.02.022 . SI: Top Scholars

[19] Ma, H.: A survey of modern compiler fuzzing. IEEE Transactions on Sustainable Computing **1**(1), 1–25 (2023)

[20] Mallissery, S., Wu, Y.-S.: Demystify the fuzzing methods: A comprehensive survey. ACM Comput. Surv. **56**(3) (2023) https://doi.org/10.1145/3623375

[21] Chen, J., Hu, W., Hao, D., Xiong, Y., Zhang, H., Zhang, L., Xie, B.: An empirical comparison of compiler testing techniques. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16, pp. 180–190. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884878 . https://doi.org/10.1145/2884781.2884878

[22] AFL: American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed Nov. 23rd, 2023 (2019)

[23] Zhao, Y., Wang, Z., Chen, J., Liu, M., Wu, M., Zhang, Y., Zhang, L.: History-driven test program synthesis for jvm testing. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, pp. 1133–1144. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510059 . https://doi.org/10.1145/3510003.3510059

[24] Tang, Y., Jiang, H., Zhou, Z., Li, X., Ren, Z., Kong, W.: Detecting compiler warning defects via diversity-guided program mutation. IEEE Transactions on Software Engineering **48**(11), 4411–4432 (2022) https://doi.org/10.1109/TSE.2021.3119186

[25] HyungSeok, H., DongHyeon, O., Kil, C.S.: Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In: Network and Distributed Systems Security (NDSS) Symposium 2019, San Diego, USA, pp. 1–15 (2019). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[26] Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS '17, pp. 2329–2344. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134020 . https://doi.org/10.1145/3133956.3134020

[27] Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: 21st USENIX Security Symposium (USENIX Security 12), pp. 445–458. USENIX Association, Bellevue, WA (2012). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[28] Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: Grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 724–735 (2019). https://doi.org/10.1109/ICSE.2019.00081

[29] Samuel, G., Simon, K., Lukas, B., Thorsten, H., Martin, J.: Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In: Network and Distributed Systems Security (NDSS) Symposium 2019, San Diego, USA, pp. 1–17 (2023). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[30] Wang, J., Zhang, Z., Liu, S., Du, X., Chen, J.: FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler. In: 32nd USENIX Security Symposium (USENIX Security 23), pp. 1865–1882. USENIX Association, Anaheim, CA (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junjie

[31] Chaliasos, S., Sotiropoulos, T., Spinellis, D., Gervais, A., Livshits, B., Mitropoulos, D.: Finding typing compiler bugs. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 183–198. Association for Computing Machinery, New York,

NY, USA (2022). https://doi.org/10.1145/3519939.3523427 . https://doi.org/10.1145/3519939.3523427

[32] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. SIGPLAN Not. **46**(6), 283–294 (2011) https://doi.org/10.1145/1993316.1993532

[33] Even-Mendoza, K., Cadar, C., Donaldson, A.F.: Csmithedge: More effective compiler testing by handling undefined behaviour less conservatively. Empirical Softw. Engg. **27**(6) (2022) https://doi.org/10.1007/s10664-022-10146-1

[34] Livinskii, V., Babokin, D., Regehr, J.: Random testing for c and c++ compilers with yarpgen. Proc. ACM Program. Lang. **4**(OOPSLA) (2020) https://doi.org/10.1145/3428264

[35] Jiang, B., Wang, X., Chan, W.K., Tse, T.H., Li, N., Yin, Y., Zhang, Z.: Cudasmith: A fuzzer for cuda compilers. In: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 861–871 (2020). https://doi.org/10.1109/COMPSAC48688.2020.0-156

[36] Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15, pp. 65–76. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737986 . https://doi.org/10.1145/2737924.2737986

[37] Chowdhury, S.A., Mohian, S., Mehra, S., Gawsane, S., Johnson, T.T., Csallner, C.: Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, pp. 981–992. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180231 . https://doi.org/10.1145/3180155.3180231

[38] Xu, H., Wang, Y., Fan, S., Xie, P., Liu, A.: Dsmith: Compiler fuzzing through generative deep learning model with attention. In: 2020 International Joint Conference on Neural Networks (IJCNN), pp. 1–9 (2020). https://doi.org/10.1109/IJCNN48605.2020.9206911

[39] Lee, S., Han, H., Cha, S.K., Son, S.: Montage: A neural network language Model-Guided JavaScript engine fuzzer. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2613–2630. USENIX Association, ??? (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung