Syntax-Aware Mutation for Testing the Solidity Compiler

Charalambos Mitropoulos,[1] Thodoris Sotiropoulos,[2] Sotiris Ioannidis[1], and Dimitris Mitropoulos[3]

[1] Technical University of Crete {cmitropoulos@isc.tuc.gr, sotiris@ece.tuc.gr}
[2] ETH Zurich, theodoros.sotiropoulos@inf.ethz.ch
[3] University of Athens, dimitro@uoa.gr

Abstract. We introduce FUZZOL, the first syntax-aware mutation fuzzer for systematically testing the security and reliability of `solc`, the standard Solidity compiler. FUZZOL addresses a challenge of existing fuzzers when dealing with structured inputs: the generation of inputs that get past the parser checks of the system under test. To do so, FUZZOL introduces a novel *syntax-aware mutation* that breaks into three strategies, each of them making different kind of changes in the inputs. Contrary to existing mutations, our mutation is able to change constructs, statements, and entire pieces of code, in a fine-grained manner that conforms to the syntactic rules of the Solidity grammar. Moreover, to explore new paths in the compiler's codebase faster, we introduce a *mutation strategy prioritization algorithm* that allows FUZZOL to identify and apply only those mutation strategies that are most effective in exercising new interesting paths. To evaluate FUZZOL, we test 33 of the latest `solc` stable releases, and compare FUZZOL with (1) *Superion*, a grammar-aware fuzzer, (2) *AFL-compiler-fuzzer*, a text-mutation fuzzer and (3) two grammar-blind fuzzers with advanced test input generation schedules: *AFLFast* and *MO*PT*-AFL*. FUZZOL identified 19 bugs in total (7 of which were previously unknown to Solidity developers), while the other fuzzers missed half of these bugs. Also, FUZZOL outperforms all fuzzers in terms of line, function, and branch coverage (from 3.75% to 408.8% improvement), while it is the most effective one when it comes to test input generation. Finally, our experiments indicate that our prioritization algorithm makes FUZZOL explore new paths roughly one day (∼24h) faster.
Keywords: Fuzzing · compilers · smart contracts · structured inputs.

1 Introduction

Smart contracts are programs that are stored on a distributed ledger (i.e., blockchain), and are used for automating the execution of agreements and transactions between crypto-currency parties. *Solidity* [5] is an object-oriented programming language designed for developing smart contracts that run on several blockchain platforms [1,2], including the *Ethereum*'s EVM (Ethereum Virtual Machine) [50]. Ethereum is an open-source blockchain with *Ether* being its native crypto-currency, which is the second-largest by market capitalization [16].

Although there are several research endeavors to identify bugs in smart contracts written in Solidity [30,15,24,27,46], there are no thorough studies focusing

on `solc`, the *standard* Solidity compiler. `solc` is a relatively new compiler that counts ∼100 releases since 2015 [5]. Given the intricate nature of Solidity, `solc` offers various special constructs related to smart contract functionalities including formal software verification and inline assembly. Due to this complexity, `solc` has exhibited a variety of bugs related to data structure mishandling, inadequate sanity checks, and unsound optimizations [6].

For the last two decades, *fuzzing* has become a standard technique for assessing software reliability and security [26,14,25]. Fuzzing has been used to identify bugs in miscellaneous entities such as system libraries [35], web and cloud applications [10], data-oriented systems [39,40], and compilers [48,34,19].

When it comes to programs whose inputs follow specific grammars (e.g. compilers), grammar-blind fuzzers (such as AFL [37]) struggle to get past syntax checks and explore deeper code. To this end, researchers have introduced a number of grammar-based fuzzing strategies [44,43,9], and have applied them to various domains, from PHP and Lua interpreters to JavaScript engines.

However, current grammar-based fuzzers have a number of disadvantages. For instance, *Superion* [44], performs some mutations that fail to preserve a correct syntax for the test cases it generates. In addition, many of these fuzzers produce test inputs completely from scratch without considering any promising and interesting language features.

Syntax-aware Mutation. We introduce *syntax-aware mutation* for fuzzing the Solidity compiler. Unlike other grammar-based techniques, our mutation processes test inputs (seeds) without breaking the syntax rules. Our mutation comes with three different strategies operating on the Abstract Syntax Tree (AST) of a smart contract written in Solidity. We apply our strategies to the programs found in the compilers' test suite. Such programs are interesting and complex, as they exercise different language features and functionalities. Our mutations then result in valid programs by making small changes to the existing, complex seeds. This helps exercise new behaviors in the compiler while preserving much of the structure and characteristics of the given seed programs. The first strategy aims to change the control-flow of the input program by mutating statements, operators and data types. To combine diverse characteristics coming from different test inputs, our second strategy selects two contracts and performs permutations on their AST leafs. Finally, our third strategy detects parts written in inline assembly and changes them in a way that stresses `solc`'s inline optimizer.

Mutation Strategy Prioritization. Based on syntax-aware mutation, we have realized FUZZOL, a practical AFL-based fuzzer. Notably, FUZZOL also incorporates existing grammar-blind and grammar-aware strategies. We further boost the effectiveness of FUZZOL by leveraging the insight that only a *small* number of mutation strategies is effective in exploring new paths [31]. To this end, FUZZOL comes with a novel *mutation strategy prioritization* algorithm that identifies and applies only those strategies that are effective for a particular seed. Given a seed smart contract c, our algorithm associates every strategy with an *effectiveness* score. The next time when FUZZOL processes c, our algorithm picks and executes only those strategies whose effectiveness score is greater than a specific threshold value, which is updated and computed dynamically.

Testing Campaign. We evaluate FUZZOL by testing 33 of the latest `solc` releases (>5,5M LoC). Further, we compare FUZZOL against Superion [44], a grammar-based fuzzer, *AFL-compiler-fuzzer* [28], a text-mutation fuzzer that has been used to test `solc` among other compilers, and two grammar-blind fuzzers with advanced seed-generation schedules: *AFLFast* [13] and *MO*PT*-AFL* [35]. Our results indicate that our approach is effective in finding bugs in `solc`. Specifically, our method led to the identification of 19 unique bugs, in total, 7 of which were related to previously unknown issues to the Solidity developers. Also, our campaign helped the developers to identify two performance issues [4,3]. Notably, the other three fuzzers failed to identify half of the bugs (10 / 19) found by FUZZOL. Our findings also show that FUZZOL outperforms the other four fuzzers both in terms of bug-revealing capability, code coverage, and test input generation. Moreover, FUZZOL achieves higher levels of coverage on average: FUZZOL was able to cover ×1.05 more LoC than Superion, ×1.08 more LoC than AFL-compiler-fuzzer, ×4.49 more LoC than AFLFast, and ×5.80 more LoC than MOPT-AFL. Finally, our prioritization algorithm makes FUZZOL exercise new compiler codebase, significantly faster (∼24h) compared to the-state-of-the-art.

Contributions. We make the following contributions.

- We introduce a novel syntax-aware mutation with three distinct strategies that performs fine-grained changes within an input program by taking into account the nature and rules of a corresponding grammar.
- We design a prioritization algorithm that is able to distinguish the most effective strategies for each seed, and speed up the fuzzing process.
- We implement our approach in an AFL-based greybox fuzzer, which we call FUZZOL. We provide in-depth evaluations for understanding the effectiveness of FUZZOL (and its key components) when compared to four state-of-the-art fuzzers in the context of a large-scale study including 33 releases of `solc`.

2 Background

We provide a brief overview of the Solidity compiler and present a number of illustrative examples of `solc` bugs that our approach can help reveal. Furthermore, we discuss the limitations of previous approaches in the context of compiler testing.

2.1 The Solidity Compiler

`solc` [7] is the standard Solidity smart contract compiler. To handle variables and function arguments, Solidity employs particular mechanisms such as *storage* (a persistent memory that every Ethereum account incorporates), and *memory* (a byte-array that holds the data until the execution of the function terminates).

Important components of `solc` include an *Application Binary Interface (ABI)*, the built-in *formal verification module*, and an *inline assembler*. ABI is a standard way to interact with contracts in the Ethereum ecosystem. Interactions can be both external (i.e., from outside of the blockchain) and contract-to-contract. Note that data is always encoded according to its type, as described in the specification of ABI. Further, the encoding is not self-describing and as a result,

it requires a schema to decode. The verification module of `solc` utilizes Microsoft's Z3 theorem prover [8,23]. Specifically, `solc` translates a contract into an SMT (Satisfiability Modulo Theory) formula, and then it attempts to prove the correctness of the contract and warn users about potential arithmetic overflows, unreachable code and more. Finally, through the inline assembler, Solidity provides a way for contracts to interact with EVM at a low level.

2.2 Bugs in the Solidity Compiler

To motivate the design of our fuzzing approach, we discuss two indicative bugs.

Bug in SMTChecker. To enable formal verification within `solc`, developers must include the `SMTChecker` via the `pragma` keyword at the beginning of their contract (in general, the `pragma` keyword can be used to employ diverse compiler features or checks). To verify a given contract and detect property violations, `solc` applies *Bounded Model Checking (BMC)* [22] to all contract functions, including `free` functions. Free functions are defined at a file-level and are not part of a contract. As a result, they cannot directly access state variables and internal functions of contracts. Nevertheless, they can call other contracts, emit events and send Ether. When BMC (through the `SMTChecker` module, line 1) examines the following simple, free function, `solc` produces an internal compiler (version 0.7.3) error:

```
1 pragma experimental SMTChecker;
2 function f() { }
```

This happens because the `SMTChecker` implementation does not reason about `free` functions–`f` in our case (a known issue among several 0.7.x versions).

Bug in array handling. The Solidity compiler may also contain bugs related to the way it handles its various structures such as arrays. Consider the code fragment below:

```
1 contract C {
2   uint[7**90][500] ids;
3 }
```

Contract `C` defines an array of integers named `ids`. Note that the size of arrays in Solidity have an upper bound. When `solc` (v0.6.0) compiles this contract an internal error occurs. This is because the compiler fails to catch that the size of `ids` is beyond the maximum size and produce a corresponding error message to the developer. As a result, the compiler crashes notably at a later stage (i.e., code generation) when trying to statically allocate memory for `ids`.

2.3 Limitations of State-of-the-Art Fuzzers

A grammar-aware fuzzer, could affect both the parsing phase and the semantic analysis process of the compiler. For this reason, grammar-aware mutation strategies have been utilized to test scripting languages [44,9]. However, previous grammar-aware strategies fail to form well-structured inputs efficiently.

Consider two recent mutations: the *enhanced dictionary-based* mutation, employed by Superion [44], and the *tree-based* mutation, used by both Superion

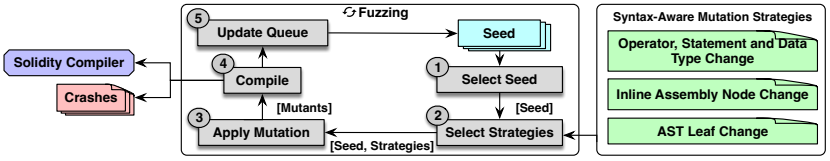


Fig. 1: Overview of our fuzzing approach for testing the Solidity compiler.

and the grammar-aware fuzzer *NAUTILUS* [9]. The basic concept behind the enhanced dictionary-based mutation strategy is a dictionary containing a list of tokens, e.g., reserved identifiers, coming from a specified grammar. Initially, the fuzzer will tokenize the test input. After locating the token boundaries, the mutation either places a new token from the dictionary in between two others, or overwrites an existing one with another also coming from the dictionary. This procedure takes place for each token in the dictionary. Unfortunately, the resulting test cases do not always conform to the syntax of the grammar. We provide an illustrative example later on, in Section 3.

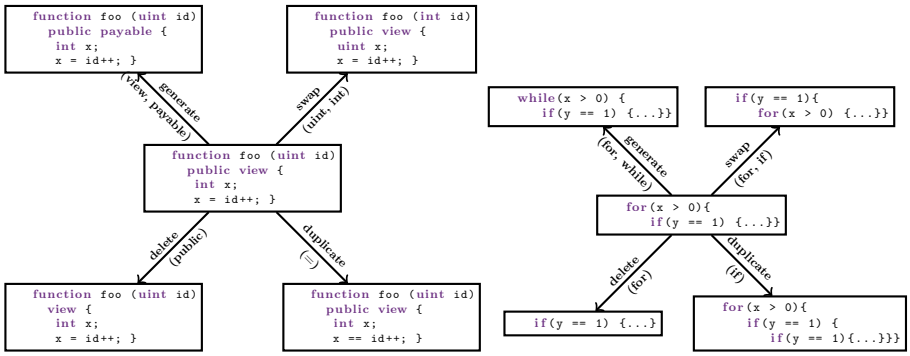
The tree-based mutation strategy selects two test inputs and attempts to parse them and generate the corresponding ASTs based on the target grammar. Note that in case of a parsing error the strategy stops. The strategy collects all sub-trees coming from both inputs and stores them in a set (S). Given the AST of the first test case, the strategy replaces every sub-tree with a random sub-tree taken from S. Each replacement leads to a new test case. By design, the tree-based mutation strategy respects the grammar of the language, as it is based on actions that process AST sub-trees. However, as we noted above, there are many cases where parsing will fail. This is going to happen if other mutations have already changed the test input that the tree-based strategy currently handles in a way that it does not conform to the grammar rules.

The AFL-compiler-fuzzer [28] offers a *text-mutation* strategy that detects specific string instances in a test case and replaces it with new text taken from an existing set. Also, it can add specific code fragments inside a program in an arbitrary manner (e.g. include a generic `if` statement such as `if(0==1)`). Such changes can be made in test cases written in different programming languages and explore compilers in a unified manner. Nevertheless, they will not be able to take into account the specifics of the language and exercise the different components of a compiler such as Solidity.

3 Fuzzing Approach

We introduce a syntax-aware mutation that aims to reveal complex bugs in `solc`. Our mutation consists of three strategies operating on the AST of a smart contract written in Solidity (Section 3.1). To further boost the effectiveness of fuzz testing, we present a prioritization algorithm that identifies and applies the strategies that are most effective in exploring new interesting paths for a specific seed program (Section 3.2). Finally, we explain some technical details behind FUZZOL, the implementation of the proposed approach (Section 3.3).

Overview. Figure 1 presents the overview of our approach. The input of our approach is a set of test programs written in Solidity. Our initial corpus con-



(a) Substitutions performed on operators and data types. (b) Substitutions performed on statements.

Fig. 2: Example of substitutions applied to operators, statements and data types.

sists of small test cases coming from the test suites of various Solidity releases. Notably, these test cases are designed to exercise all the different compiler features. First, we select a test input (seed) from the fuzzing queue. Then, our approach applies both grammar-blind strategies (such as *bit/byte flips* [37]) and our syntax-aware mutation to the selected seed. This *syntax-aware mutation* comes with three different mutation strategies. Each strategy has a different role in exercising Solidity's codebase. In particular, the first strategy performs changes to the control-flow of the input program by updating statements, operators and data types found in each smart contract. Our second strategy selects a random leaf from a contract's AST and place it in another contract. In this manner, we combine different characteristics (e.g. variable names) stemming from multiple contracts to create seeds that are more likely to trigger bugs. The third strategy detects the parts of a given AST written in inline assembly and replaces assembly code opcode arguments with other opcodes. The main goal of this strategy is to yield programs that contain more complicated inline assembly operations, and consequently involve more opportunities for `solc`'s inline optimizer.

To make our approach faster and explore deeper code, when a seed is selected from the queue, we employ a *mutation strategy prioritization* algorithm. As a result, we are able to identify, select, and apply the strategies that are most effective in exploring new paths for that specific seed. Our algorithm relies on an *effectiveness function* that leverages details from previous iterations of the fuzzing process (Section 3.2).

3.1 Syntax-Aware Mutation

Our syntax-aware mutation consists of different strategies. In the following, we analyze the proposed strategies. Given a smart contract c, each strategy performs a different change in c's AST altering the contract's behavior accordingly.

3.1.1 Operator, Statement and Data Type Change Strategy. Our first strategy applies changes in either an operator, a statement, or a data type of a

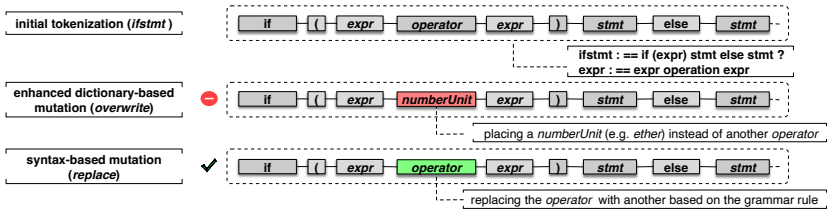


Fig. 3: The *overwrite* substitution of the *enhanced dictionary-based* strategy can potentially break the syntax rules. This is not the case in our *operator, statement, and data type change* strategy.

given contract's AST. To do so, it replaces the selected item with another node of the same type (e.g., an operator is substituted by another operator). Thus, the strategy does not violate the syntax of the contract even though its behaviour and control flow can be significantly changed.

Definition 1 (Operator, Statement and Data Type change). *Let c be a smart contract and let $a \in \{ops, stm, datatype\}$ be a node in c's AST that is either an operator, a statement or a data type. Given a node $a' \in \{ops, stm, datatype\}$, we say that $mut(c) = c[mut(a)] = c[a'/a]$ is an operator, statement and data type change of the contract c that substitutes either an operator, a statement or a datatype node with another similar node, preserving the syntax of the language.*

Specifically, a substitution $c[a'/a]$ may involve (1) the replacement of a token (operator, datatype) or a statement a with another token / statement a' found in the AST, or (2) the generation or deletion (i.e., represented by an empty node ϵ) of valid tokens and program statements. Specifically, our strategy employs four distinct types of substitutions namely: *generate*, *swap*, *delete*, and *duplicate*. By performing such substitutions on Solidity tokens is of particular importance. This is because Solidity has a number of special tokens related to smart contract functionalities such as Ether units (e.g., `finney`, `wei` and `szabo`) and payment addresses (e.g., `address`) that can change the course of compilation. For each substitution we make sure that we maintain a correct grammar syntax. Note that if a substitution violates the syntax we abort it.

Figure 2 demonstrates how each substitution works using two code fragments as target examples. The fragments are depicted at the center of Figures 2a and 2b respectively. In Figure 2a, we include a `view` function (`foo`) (note that a `view` function can read but cannot write to the variables that process the persistent memory), while Figure 2b illustrates a simple `if` statement inside a `for` loop.

Note that the *generate* substitution works in a way similar to the *overwrite* method of the enhanced dictionary-based strategy implemented in Superion [44]. However, the existing *overwrite* strategy may violate the syntax rules of the grammar, as it chooses a random token of the program and overwrites it with a randomly-generated token of the language without checking whether this replacement breaks the syntax of the program (see also Section 2.3). This will not happen with our *generate* method because the strategy will enforce a correct syntax. Figure 3, highlights The distinct difference between *overwrite* and *generate*.

3.1.2 AST Leaf Node Change Strategy. Our *AST leaf node change* strategy takes the contract c_1 that is currently first on the queue, and another randomly-selected contract c_2 from the queue. Then, it parses the contracts and generates the corresponding ASTs. Given an AST leaf node of the first contract, the strategy replaces it with a random leaf node that stems from the second contract. Such a replacement leads to a new test case that involves unexpected characteristics (e.g., variable names), which in turn examine new compiler behaviours. Notably, the strategy considers changes only in the tree leafs and not in the sub-trees, making our strategy efficient and fast. This is because moving sub-trees across ASTs leads to large test cases that slow down the process.

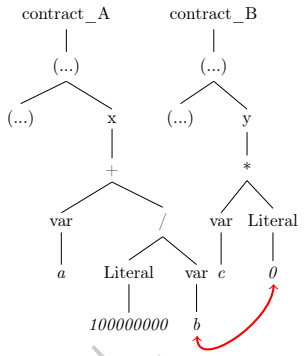


Fig. 4: *AST leaf node change.*

Definition 2 (AST Leaf Node Change). *Let c_1 and c_2 be two contracts and let l_1 be a leaf node of c_1, and l_2 be a leaf node of c_2. Then the we say that $mut(c_1) = c_1[l_2/l_1]$ is an AST leaf node change of the contract c_1 that replaces a leaf l_1 with another leaf l_2 from another AST c_2, preserving language syntax.*

This strategy again results in well-formed programs because l_2 of c_2 will replace l_1 of c_1, only if this change respects the grammar rules of the language. An example is depicted in Figure 4. Contract `A`, contains the following expression: `x = a + (100000000 / b)` while contract `B` includes: `y = c * 0`. Our strategy takes the leaf node `0` from contract `A` and replaces it with the leaf node `b` from contract `B`. Such a change can produce unexpected behaviours, e.g., triggering the compiler check that verifies whether the program is free from divisions by zero. Note that the strategy is designed to preserve the syntax, contrary to the tree-based strategy discussed in Section 2.3.

3.1.3 Inline Assembly Node Change Strategy. In the context of Solidity, developers are able to employ blockchain-specific opcodes only available through inline assembly. However, malformed inline assembly code can affect the optimizations that can be applied to programs by the compiler, leading to crashes [18]. Our *inline assembly node change* strategy identifies inline assembly nodes, and makes changes in the corresponding assembly's opcodes depth.

Definition 3 (Inline Assembly Node Change). *Let c be a smart contract, $o_1 \in$ opcodes be an opcode node of c, and n be a child node of o_1. Given another opcode $o_2 \in$ opcodes, we say that $mut(c) = c[o_1[o_2/n]/o_1]$ is an inline assembly node change of the contract c that replaces an argument of an opcode with another opcode according to the grammar rules of the language. This change increases the depth of the opcodes in the AST.*

Consider the following example. In a smart contract `A` that involves the opcode $o =$ `add(x, y)`, the strategy operates as follows: First, it selects a random

Algorithm 1: Mutation Strategy Prioritization

```
1  Function Prioritization(t, strategies, scores, k, bound):
2      if scores = nil then // first time we process t
3          for s ∈ strategies do
4              apply strategy s
5              scores_s ← eff(s, t)
6          bound ← GetKthScore(scores, k)
7      else
8          for s ∈ strategies do
9              if scores_s ≥ bound then
10                 apply strategy s
11                 scores_s ← eff(s, t)
12                 if scores_s ≤ bound then
13                     bound ← scores_s
14         bound ← GetKthScore(scores, k)
15     return bound
16 End Function
```

opcode o' (e.g.., `mul`) from the set of available *opcodes* supported by the Solidity's inline assembly language. Then, it chooses a random child node of the initial opcode o (i.e., either `x` or `y`) and replaces it with the new opcode o' with the same arguments as o's (e.g. `add(mul(x, y), y)`).

Overall, the *inline assembly node change* strategy produces complicated code and makes it hard for the compiler to solve some formulas used for verifying program correctness. Further, changing the inline assembly code can lead to discrepancies among the optimized code and the regular one. The reason behind this is that a program that manifests more complex opcodes in inline assembly triggers more paths in the `solc`'s inline assembly optimizer, as the code now involves more optimization opportunities.

3.2 Mutation Strategy Prioritization

The key idea of our algorithm is that for every seed, instead of applying all strategies in a deterministic manner (as all AFL-based fuzzers do), we choose to perform *only* the top-k strategies that are most effective in producing test cases that explore new paths. In this way, testing does *not* waste time and resources in applying strategies that are deemed to be ineffective for a particular seed.

To achieve this, we introduce a function that evaluates the *effectiveness* of a strategy s on a test case t based on the fraction of the number of new explored paths ($\#newpaths$) and the number of times the strategy s is applied to t ($\#executions$).

$$eff(s,\ t) = \frac{\#newpaths}{\#executions}$$

Intuitively, the greater the $eff(s,t)$ is, the more effective the mutation strategy s is on this test case.

Algorithm 1 summarizes the details of the concept. The inputs of the algorithm are: (1) one seed program (t), (2) the set of mutation strategies that can

be potentially applied to t, (3) an integer constant k indicating the number of top strategies exploring new paths, (4) the effectiveness scores of the strategies from the last time the t was processed, and (5) a *bound* value. Based on these inputs, our algorithm operates as follows. If it is the first time we process the given test case (which indicates that we do not have the effectiveness scores from previous runs, i.e., *scores* = **nil**, line 1), the algorithm applies all available strategies and computes their scores (lines 2–4). Then, the algorithm computes the *bound* value, which is used as an indicator of whether a strategy should be selected or not the next time we will process the seed. This bound value is the result of the `GetKthScore` function, which sorts the list of effectiveness scores in a descending order and then returns the score of the k^{th} strategy.

If the given test case has been previously processed, the algorithm iterates all mutation strategies and applies only those whose effectiveness score is greater or equal to the bound (line 8). Practically, this means that the algorithm performs the top-k mutation strategies with the greatest effectiveness scores as computed in the previous run of the given seed. To prevent our algorithm from applying the same top-k strategies all the time, when the current effectiveness score *eff(s, t)* of an executed mutation strategy is lower than the value of *bound*, the algorithm updates *bound* as *eff(s, t)* (lines 12-13). Conceptually, updating and lowering *bound* gives the opportunity to other strategies to take the place of a strategy currently included the top-k list (assuming the condition at line 9 holds).

3.3 Fuzzol

We have implemented FUZZOL, an AFL-based fuzzer to test the Solidity compiler. We plan to make our fuzzer publicly available, concurrently with this paper's publication. We have developed our novel syntax-aware mutation together with its three distinct strategies in C/C++. Furthermore, we have adapted Superion's [44] tree-based and enhanced dictionary-based mutation strategies (also written in C/C++) to handle smart contracts and included them in our implementation. Beyond that, FUZZOL also employs other common grammar-blind strategies [37] such as *bit/byte flips* and *interesting values*. Finally, FUZZOL follows our prioritization algorithm to identify and apply strategies that are effective for a particular seed in the way we discussed in the previous section.

We built the Solidity grammar using ANTLR 4. Even though an ANTLR grammar for Solidity exists, it is incomplete and does not support the latest versions of the Solidity compiler. Thus, we have enriched the grammar adding more than 200 lines of code containing new grammar rules.

4 Evaluation

We evaluate FUZZOL by examining multiple releases of the Solidity compiler, seeking answers to the following research questions:

- **RQ1** Is FUZZOL effective in finding bugs in the Soldity compiler?
- **RQ2** How effective is our syntax-aware mutation when compared to grammar-blind strategies?
- **RQ3** How effective is FUZZOL when compared to the state-of-the-art fuzzers?
- **RQ4** Does our prioritization algorithm speed up the fuzzing process?

Table 1: Total bugs discovered in all `solc` versions by FUZZOL. Bugs are grouped in categories based on their root cause.

Category	Total	Fixed	Confirmed (Unfixed)
Verification	5	5	0
ABI encoding	2	2	0
Inline assembly	3	3	0
Data structures & functions	8	7	1
Optimization	1	0	1
Total	**19**	17	2

4.1 Evaluation Setup

We focused on the last 33 Solidity versions, i.e., from `solc-v0.5.11` to `solc-v0.8.17`. We excluded `solc-v0.8.1`, `solc-v0.8.2`, and `solc-v0.8.14` because we were not able to properly set them up due to configuration problems. Each compiler version contains 230k LoC on average.

Our initial corpus of seeds was populated by the test cases coming from the aforementioned versions. We extracted small test cases (less than 1 kB – recall that using small and targeted seeds is preferred in compiler testing [42]) that explore all the different functionalities from every version we tested. We gathered 1.5k test cases in total, each containing 10 LoC on average.

4.2 RQ1: Discovering Bugs

FUZZOL triggered several crashes. We examined the crashes to identify their source and find potential bugs in the Solidity compiler. Table 1 summarizes our results. FUZZOL identified 19 bugs in total, which we reported to the development team of Solidity. The team was already aware of some bugs. For the unknown bugs (enlisted in Appendix A), there were prompt fixes (∼6 hours after our report). Note also, that our testing campaign helped identify two performance issues [4,3]. We further classified the discovered bugs based on their root cause. In the following, we describe the categories that we have identified.

Verification-related bugs. As we discussed in Section 2.1, `solc` enables formal verification through the `SMTChecker` module. We have found that several contracts that invoke this module can lead to compiler crashes. By examining these cases we have identified five distinct bugs. As an example of this bug category, consider again the first issue discussed in Section 2.2.

ABI Encoding Bugs. Using the `ABIEncoder` module, `solc` encodes and decodes various elements of a contract (e.g., structs) into other formats such as JSON. FUZZOL was able to identify two bug instances related to ABI encoding. As an example, consider the following test case:

```
1 function f() public {
2   mapping(uint=>uint)public memory x;
3 }
```

This test case calls the `mapping` function, which can be used to store data in the form of key-value pairs (both `uint` in our case). Our *AST leaf node change* strategy replaced the second leaf of `uint` with a new leaf `uint[1000000]`, which comes from another contract. The corresponding mutant triggered a *"mapping used outside of storage"* error in `solc`. This happens because when the

`ABIEncoder` attempts to encode the elements of the contract, it does not prevent the processing of an out-of-bounds array.

The bug above highlights that combining individual characteristics of two contracts (i.e., through the *AST leaf node change* strategy) can result in test cases with unique features that are more likely to trigger bugs. For example, it is highly unlikely for a generator to produce the construct `uint[1000000]`.

Inline Assembly-related Errors. As discussed in Section 2.1, contracts can have direct access to the EVM through `solc`'s inline assembler. In Solidity, inline assembly is marked by the `assembly { ... }` statement. Inside the curly braces, developers can utilize variable declarations, literals, opcodes and more. We observed that in three occasions the compiler did not handle such features in a correct manner. As an example, consider a contract that assigns one integer variable to another in inline assembly:

```
1 assembly {
2   uint x; uint y;
3   x := y
4 }
```

Our *operator, statement and data type change* strategy, changed the type of `x` from `uinit` into a `calldata` type. When `solc` versions 0.6.4 and 0.6.8 attempted to compile the code above they both crashed. This is because there was a bug in the assignment implementation of the `calldata` types.

Bugs in Data Structures and Functions. We have discovered bugs in the implementations of various Solidity data structures and modules. Overall, FUZZOL found eight bugs coming from this category.

We have already discussed one of theses issues in Section 2.2. (bug in array handling). Our *AST leaf node change* strategy helped reveal this bug in the following manner. It collected a large integer number from a leaf of another contract and substituted the boundary of the array with this number. When processing the corresponding test case during the code generation stage, the compiler crashed because there were no checks regarding array limits.

Optimization Bugs. We identified one bug related to compiler optimizations. The code that led to the identification of the issue contained a *hex* value, that was replaced with another, large, *hex* value, i.e., `hex"344383800E6110...`. In this case, our *AST Leaf Node Change* strategy replaced the leaf node of the *hex* value, and replaced it with another *hex* value node, existed in another contract.

4.3 RQ2: Comparing Syntax-Aware and Grammar-Blind Strategies

We compare our strategies (described in Section 3.1) with standard grammar-blind strategies. We focus on the state-of-the-art strategies offered by AFL, namely: *bit/byte flips*, *arithmetics* and *interesting values*. To do so, we compare the number of unique test cases each strategy generates, i.e., the test cases that trigger new paths, over a 48 hour window. Note that comparing test cases is a standard way to evaluate strategies [36,47,31]. Also, collecting seeds for 48 hours is consistent with previous work where the time window for the experiments was roughly 24 hours [13,35]. Furthermore, we compare the strategies in

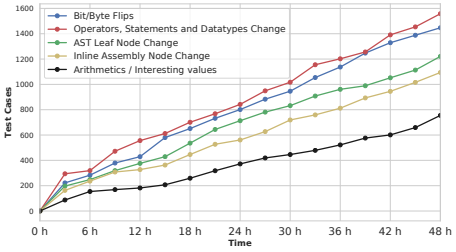


Fig. 5: Test cases produced for `solc` 0.8.13 by each strategy.

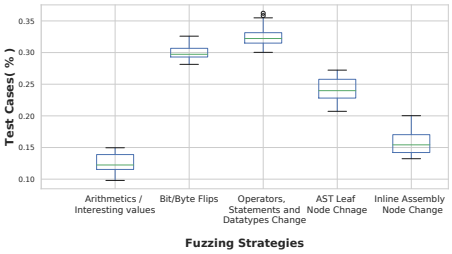


Fig. 6: The ratio of interesting test cases per fuzzer to the total number of generated test cases.

terms of effectiveness. We define the effectiveness of a mutation strategy as the ratio of unique test cases to the total number of test cases it generates [31,32,9].

Figure 5 presents the evolution of the generated test cases by each strategy for `solc` v0.8.16. We observed very similar trends in other compiler versions and omit the corresponding results for brevity. Our results indicate that all mutation strategies show a linear growth with different coefficients. Our *operator, statement and data type change* strategy turns out as the most productive one at all times. Notably, after 48 ours it has generated 250 test cases more than the *bit/byte flips* strategy (the second most productive), and 1000 more than the *arithmetics* strategy (the least productive). Our two other syntax-aware strategies come in the third and fourth place respectively.

Our results indicate that our strategies offer an increasing rate of producing interesting test cases. Another observation is that grammar-blind strategies can be productive when fuzzing a compiler, an observation made also by the authors of Superion [44], who examined different interpreters.

Focusing on effectiveness we observed that our *operator, statement and data type change* strategy is the most effective one. Figure 6 shows box-plots that present the effectiveness of each strategy for all 32 `solc` versions. The green line inside every box plot indicates the corresponding median value. The *operator, statement and data type change* strategy has the highest ratio overall (30–40%). Then, *bit/byte flipping* is the second best strategy with an overall ratio of 28–32%. Our *AST leaf node change* strategy has a (23–28%) ratio, and the *inline assembly node change* strategy comes next with a 15–20% ratio. Finally, the *arithmetic* strategy ratio is the lowest (10–15%).

4.4 RQ3: Comparison with State-of-the-Art Fuzzers

We compare FUZZOL with four AFL-based fuzzers, namely: Superion [44], the AFL-compiler-fuzzer [28], AFLFast [13] and MOPT-AFL [35]. Appendix B presents the design differences between FUZZOL and the fuzzers above. Unfortunately, given the time restrictions, we were not able to compare FUZZOL with other grammar-aware fuzzers, such as NAUTILUS [9], *IFuzzer* [43], *GRIMOIRE* [11]. This is because these fuzzers are not AFL-based, thus it requires much engineering effort and sufficient time to make them run for Solidity.

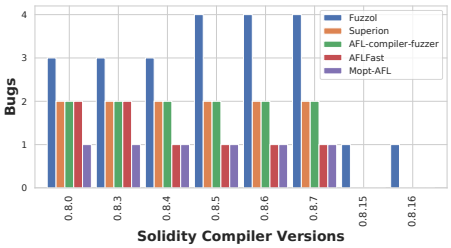


Fig. 7: Bugs across 14 of the last solc versions.

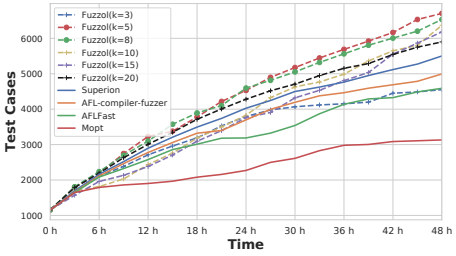


Fig. 8: The ratio of interesting test cases per strategy to the total number of generated test cases.

To perform our comparison we focus on two dimensions: (1) the bug finding capabilities of each tool and (2) code coverage. To gather our results, we run all fuzzers for 48 hours. All experiments were run on a machine with an Intel Xeon CPU E5-2650v3 2.30GHz processor with 6 logical cores and 64 GB of RAM.

Figure 7 presents the bugs discovered by each tool for the last 12 `solc` versions. From versions 0.8.0 to 0.8.7, all tools reported crashes related to existing bugs. In all cases, FUZZOL found more bugs than any other fuzzer. From versions 0.8.8 to 0.8.13 there are no bugs found by the fuzzers. While in versions v0.8.15 and v0.8.16 FUZZOL identified one optimization issue, while the other four fuzzers were not able to detect any bugs.

We measured how much code is exercised by each tool by examining three `solc` versions. To do so, we used *afl-cov* [38]. Table 2 presents the line, function, and branch coverage per fuzzer – version.

Overall, we found that on average, FUZZOL was able to cover ×1.05 (i.e., 5.4% code coverage improvement) more lines than Superion, ×1.08 (i.e., 8.5% code coverage improvement) more lines than AFL-compiler-fuzzer, ×4.40 more lines than AFLFast (i.e., 230.6% code coverage improvement) and ×5.80 more lines than MOPT (i.e., 408.8% code coverage improvement). Notably, given the compiler's large codebase, an 1% code coverage improvement translates to covering 2,220 more lines of code. The situation is similar in the case of functions and branches where FUZZOL outperformed all fuzzers. In particular, our results show that on average, FUZZOL invoked ×1.04 more functions than Superion, ×1.08 more functions than AFL-compiler-fuzzer, ×2.4 more than AFLFast and MOPT. Finally, in terms of branch coverage, FUZZOL was ×1.03 better than Superion, ×1.07 better than AFL-compiler-fuzzer, ×2.56 better than AFLFast and ×3.2 better than MOPT.

All the above clearly indicate that the techniques implemented in FUZZOL lead to better results compared to the-state-of-the-art, in terms of both bug-finding capabilities and code coverage improvement.

4.5 RQ4: Mutation Strategy Prioritization Algorithm

To evaluate our mutation strategy prioritization algorithm (Section 3.2), we run different FUZZOL instances with different settings, i.e., we tried out different

Table 2: Line, function and branch coverage for three of the latest `solc` versions.

	Line Coverage (%)			Function Coverage (%)			Branch Coverage (%)		
Tool	**v0.8.16**	**v0.8.15**	**v0.8.13**	**v0.8.16**	**v0.8.15**	**v0.8.13**	**v0.8.16**	**v0.8.15**	**v0.8.13**
FUZZOL	48.1	48.3	48.0	21.6	21.8	21.1	28.5	28.5	28.3
Superion [44]	46.3	45.0	46.1	20.6	20.5	20.6	28.1	28.0	27.5
AFL-compiler-fuzzer [28]	45.2	44.3	45.6	19.6	20.5	20.1	25.7	26.1	26.5
AFLFast [13]	15.2	15.0	15.4	<10	<10	<10	11.2	<10	11.3
MOPT [35]	<10	<10	<10	<10	<10	<10	<10	<10	<10

values of k, which is an input of our algorithm. Recall that k indicates the number of top strategies exploring new paths (see Section 3.2). Focusing on performance, we examined the number of unique test cases generated over time. Further, we compared FUZZOL's performance against the corresponding ratios of the other four tools mentioned earlier.

Apart from the three strategies discussed in Section 3, FUZZOL also incorporates all grammar-blind strategies of AFL and the grammar-based strategies implemented in Superion [44], namely, *enhanced dictionary-based* mutation and *tree-based* mutation, counting 20 strategies in total. Therefore, running our algorithm with $k = 20$ is equivalent to running FUZZOL with the default, AFL-based prioritization algorithm, i.e., running all the strategies in the same order.

We run all fuzzers on `solc` version 0.8.16 for 48 hours. In the case of FUZZOL we used 6 instances with different k's. Figure 8 illustrate our results. For the first four hours, all fuzzers add interesting test cases in the queue. From that point and on, all FUZZOL 's instances, except for FUZZOL 's instance with $k = 3$, generate more interesting test cases than all the other fuzzers. After 24 hours, five FUZZOL instances take the lead as they generate 1,100 test cases than Superion, 1,500 test cases than AFL-compiler-fuzzer, and 2,000 test cases than both AFLFast and MOPT-AFL. Observe that $k = 5$ and $k = 8$ instances are the most effective ones as they yield 1,100 more test cases than the baseline, i.e., $k = 20$. We observed similar trends in all the remaining compiler versions.

Our results indicate that our prioritization algorithm further boosts the fuzzing process. First, as we observe in Figure 8 the baseline FUZZOL instance (the black thick line) is faster than all the other fuzzers, something that is consistent with our RQ3 findings (Section 4.4). However, it is slower than the instances that employ the algorithm (except the one with $k = 3$). This indicates that when our algorithm is used with values of k that are neither too high nor too low (e.g., $k = 5$, $k = 8$), there is a significant benefit in the performance of the fuzzing process, because FUZZOL produces unique test cases much faster.

5 Related Work

Grammar-aware mutation and generation. We have already discussed the basic limitations of the strategies employed by Superion [44] in Section 2.3. *IFuzzer* [43] is a grammar-aware fuzzer that uses genetic programming [45] to compose new seeds for the JavaScript interpreter. Holler et al. [29] have proposed a similar approach. Specifically, they extract code fragments form sample code and use them to mutate test cases. On the grammar-aware generation front, *NAUTILUS* [9] can generate seeds containing valid code and then perform tree-

based mutations on them (see also Section 2.3). Then, the corresponding mutants can be used to test languages such as PHP and JavaScript. Notably, NAUTILUS works without an initial set of test cases and generates inputs from scratch without taking into account different language characteristics. Recall that utilizing the existing test cases of a compiler helps exercising interesting compiler features (see Section 3). *GRIMOIRE* [11] extends NAUTILUS adding more mutations including string replacements, recursive replacements, and more.

Compared to this body of work, FUZZOL is the first fuzzer for the Solidity language, which implements novel syntax-aware strategies that takes into account Solidity's grammar and syntax rules.

Testing compilers. Compiler testing approaches have been extensively surveyed [21]. We enumerate a number of methods related to our work. *Csmith* [49] automatically generates C programs that are free from undefined behavior. Randomized differential testing has also been used to examine production compilers such as GCC and Clang [41,34]. The AFL-compiler-fuzzer [28] uses a text-based mutation to test different compilers, including `solc` (as we discussed in Section 2.3). Our evaluation indicated that our approach is more effective and achieves better results in terms of both bug-finding capabilities and code coverage improvement than the AFL-compiler-fuzzer.

Advanced scheduling. There are several approaches that provide more dynamic and effective power schedules for seeds prioritization. MOPT [35] employs a modified *particle swarm optimization* algorithm to make an effective use of the mutation scheduler. To further improve scheduling, *Cerebro* [33] takes into account elements such as coverage, and execution time. Furthermore, Cha et al. [17] employ symbolic analysis on execution traces to maximize effectiveness. *AFLGo* [12] and *Hawkeye* [20] introduce power schedules able to direct the fuzzing process towards specific locations of a programs (*directed* fuzzing), based on distance metrics. AFLFast [13] and *fair-fuzz* [32], include a scheduling algorithm that prioritizes rarely-exercised branches to achieve higher coverage.

FUZZOL implements a novel prioritization algorithm that is able to identify mutations that can achieve better results in terms of exploring new paths. Conceptually, our algorithm instead of prioritizing seeds, it prioritizes mutations.

6 Conclusion

We have presented FUZZOL, a greybox fuzzer for the Solidity compiler. FUZZOL comes with two key components for boosting the effectiveness of Solidity compiler fuzzing: (1) a syntax-aware mutation for producing syntactically-valid mutants that get past the syntactic checks of the compiler (and thus exploring deeper code), and (2) a mutation strategy prioritization algorithm that treats each seed differently, according to the mutations that are most suitable for that specific seed. Our in-depth evaluation on 33 compiler releases indicates that FUZZOL is superior to four state-of-the-art fuzzers in terms of bug-finding capability, improved code coverage, and test input generation. Finally, our prioritization algorithm makes FUZZOL generate unique test inputs almost one day faster.

References

1. The Counterparty financial platform, `https://counterparty.io/`, [Online; accessed 15-January-2023]
2. Hedera hashgraph, [Online; accessed 15-January-2023]
3. Optimized contract crash. https://github.com/ethereum/solidity/issues/12840, [Online; accessed 05-January-2023]
4. Optimized contract freeze. https://github.com/ethereum/solidity/issues/12848, [Online; accessed 03-January-2023]
5. Solidity, `https://docs.soliditylang.org/en/v0.8.0/`, [Online; accessed 03-January-2023]
6. Solidity compiler - issues catalog, `https://github.com/ethereum/solidity/issues`, [Online; accessed 15-January-2023]
7. The Solidity contract-oriented programming language Github repository, `https://github.com/ethereum/solidity`, [Online; accessed 05-January-2023]
8. Z3 GitHub repository (2021), `https://github.com/Z3Prover/z3`, [Online; accessed 20-January-2023]
9. Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A., Teuchert, D.: NAUTILUS: fishing for deep bugs with grammars. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS) (2019)
10. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: Proceedings of the 41st International Conference on Software Engineering. p. 748–758. ICSE '19, IEEE Press (2019)
11. Blazytko, T., Aschermann, C., Schlögel, M., Abbasi, A., Schumilo, S., Wörner, S., Holz, T.: Grimoire: Synthesizing structure while fuzzing. In: Proceedings of the 28th USENIX Conference on Security Symposium. p. 1985–2002. USENIX Association, USA (2019)
12. 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. p. 2329–2344. CCS '17, Association for Computing Machinery, New York, NY, USA (2017)
13. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. p. 1032–1043. CCS '16, Association for Computing Machinery, New York, NY, USA (2016)
14. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: Proceedings of the 2013 International Conference on Software Engineering. p. 122–131. ICSE '13, IEEE Press (2013)
15. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: A smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 454–469. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020)
16. Browne, R.: Ether, the world's second-biggest cryptocurrency, is closing in on an all-time high (2021), `https://www.cnbc.com/2021/01/19/bitcoin-ethereum-eth-cryptocurrency-nears-all-time-high.html`, [Online; accessed 20-January-2023]
17. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. p. 725–741. SP '15, IEEE Computer Society, USA (2015)

18. Chaliasos, S., Gervais, A., Livshits, B.: A study of inline assembly in solidity smart contracts. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022)
19. 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. p. 183–198. PLDI 2022, ACM, New York, NY, USA (2022)
20. Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., Liu, Y.: Hawkeye: Towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 2095–2108. CCS '18, Association for Computing Machinery, New York, NY, USA (2018)
21. Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing. ACM Comput. Surv. **53**(1) (Feb 2020)
22. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. p. 137–148. ASE '09, IEEE Computer Society, USA (2009)
23. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
24. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 415–427. ISSTA 2020, ACM, New York, NY, USA (2020)
25. Godefroid, P.: Fuzzing: Hack, art, and science. Commun. ACM **63**(2), 70–76 (Jan 2020)
26. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. Queue **10**(1), 20–27 (Jan 2012)
27. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Analyzing the out-of-gas world of smart contracts. Commun. ACM **63**(10), 87–95 (Sep 2020)
28. Groce, A., van Tonder, R., Kalburgi, G.T., Le Goues, C.: Making no-fuss compiler fuzzing effective. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. p. 194–204. CC 2022, Association for Computing Machinery, New York, NY, USA (2022)
29. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proceedings of the 21st USENIX Conference on Security Symposium. p. 38. Security'12, USENIX Association, USA (2012)
30. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 259–269. ASE 2018, Association for Computing Machinery, New York, NY, USA (2018)
31. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 2123–2138. CCS '18, Association for Computing Machinery, New York, NY, USA (2018)
32. Lemieux, C., Sen, K.: Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 475–485. ASE 2018, Association for Computing Machinery, New York, NY, USA (2018)
33. Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., Liu, Y.: Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In: Proceed-

ings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 533–544. ESEC/FSE 2019, ACM, New York, NY, USA (2019)
34. Livinskii, V., Babokin, D., Regehr, J.: Random testing for C and C++ compilers with YARPGen. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020)
35. Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R.: MOpt: Optimized mutation scheduling for fuzzers. In: Proceedings of the 28th USENIX Conference on Security Symposium. p. 1949–1966. USENIX Association, USA (2019)
36. Lyu, C., Ji, S., Zhang, X., Liang, H., Zhao, B., Lu, K., Wang, T., Beyah, R.: Ems: History-driven mutation for coverage-based fuzzing. In: 29th Annual Network and Distributed System Security Symposium) (2022)
37. M. Zalewski: American fuzzy lop. `https://lcamtuf.coredump.cx/afl/` (2013), online accessed; 13-January-2023
38. Rash, M.: afl-cov - AFL fuzzing code coverage (2021), `https://github.com/mrash/afl-cov`, [Online; accessed 06-January-2023]
39. Rigger, M., Su, Z.: Testing database engines via pivoted query synthesis. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 667–682. USENIX Association (Nov 2020)
40. Sotiropoulos, T., Chaliasos, S., Atlidakis, V., Mitropoulos, D., Spinellis, D.: Data-oriented differential testing of object-relational mapping systems. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1535–1547 (2021)
41. Sun, C., Le, V., Su, Z.: Finding and analyzing compiler warning defects. In: Proceedings of the 38th International Conference on Software Engineering. p. 203–213. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016)
42. Sun, C., Le, V., Zhang, Q., Su, Z.: Toward understanding compiler bugs in GCC and LLVM. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. p. 294–305. ISSTA 2016, Association for Computing Machinery, New York, NY, USA (2016)
43. Veggalam, S., Rawat, S., Haller, I., Bos, H.: Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In: Proceedings of the 21st European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 9878, pp. 581–601. Springer (2016)
44. Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: Grammar-aware greybox fuzzing. In: Proceedings of the 41st International Conference on Software Engineering. p. 724–735. ICSE '19, IEEE Press (2019)
45. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering. p. 364–374. ICSE '09, IEEE, USA (2009)
46. Wüstholz, V., Christakis, M.: Harvey: A greybox fuzzer for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1398–1409. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020)
47. Yan, S., Wu, C., Li, H., Shao, W., Jia, C.: Pathafl: Path-coverage assisted fuzzing. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. p. 598–609. ASIA CCS '20, Association for Computing Machinery, New York, NY, USA (2020)
48. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. SIGPLAN Not. **46**(6), 283–294 (Jun 2011)

49. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 283–294. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011)
50. Zubairy, R.: Create a blockchain app for loyalty points with Hyperledger Fabric Ethereum Virtual Machine (2018), [Online; accessed 06-January-2023]

A Bugs Previously Unknown to Solidity Developers

In the table below, we enumerate all bugs that (1) FUZZOL identified and (2) were unknown to Solidity developers.

Table 3: Category and references of the bugs that were unknown to Solidity developers.

Category	**URL**
Data structures & functions	`github.com/ethereum/solidity/issues/11677`
Data structures & functions	`github.com/ethereum/solidity/issues/10502`
Data structures & functions	`github.com/ethereum/solidity/issues/7550`
Inline assembly	`github.com/ethereum/solidity/issues/9936`
Inline assembly	`github.com/ethereum/solidity/issues/11680`
Verification	`github.com/ethereum/solidity/issues/10798`
Verification	`github.com/ethereum/solidity/issues/7546`

B Differences Between FUZZOL and Fuzzers Included in Our Evaluation

In the following table, we present the main design differences between FUZZOL and the related fuzzers included in our evaluation. Note that all fuzzers are AFL-based.

Table 4: Point-to-point comparison between FUZZOL and the fuzzers included in our evaluation. GB: grammar-blind, GA: grammar-aware, TM: text-mutation

Fuzzer	**Mutation**	**Advanced Schedule**	**Target Program**
Superion [44]	GA, GB	✗	JavaScript interpreter
AFL-compiler-fuzzer [28]	TM	✗	Solidity, Move, Fe, Zig
AFLFast [13]	GB	✓	Binaries
MOPT [35]	GB	✓	Binaries
FUZZOL	GA, GB	✓	Solidity