SolForge: A Large Language Model-Based Approach for the Differential Testing of Solidity Compilers

Gabriel Zortea Salvi
Universidade Federal da Fronteira Sul
Chapecó, Santa Catarina, Brazil
gabriel.salvi@estudante.uffs.edu.br

Andrei Braga
Universidade Federal da Fronteira Sul
Chapecó, Santa Catarina, Brazil
andrei.braga@uffs.edu.br

Guilherme Dal Bianco
PPGPCA - UFFS
Chapecó, Santa Catarina, Brazil
guilherme.dalbianco@uffs.edu.br

Samuel Feitosa
PPGPCA - UFFS
Chapecó, Santa Catarina, Brazil
samuel.feitosa@uffs.edu.br

Abstract

Solidity is the primary language used to develop smart contracts on Ethereum. Because deployed contracts are immutable, compiler reliability is particularly important, as compilation defects can introduce persistent vulnerabilities into generated bytecode. Traditional testing approaches often struggle to explore the semantic complexity required to expose edge-case compiler failures. This paper presents SolForge, an automated fuzzing and differential testing framework driven by Large Language Models (LLMs). Rather than relying on handcrafted templates or model fine-tuning, SolForge uses combinatorial prompt generation to produce structurally diverse Solidity contracts. These contracts are then compiled across different compiler configurations, including *solc* (standard and optimized) and *solang*, to identify behavioral inconsistencies and internal failures. The experimental evaluation produced 1,357 syntactically valid contracts and revealed 1,041 compilation disagreements across compiler targets and configurations, most of which corresponded to architectural incompatibilities between EVM-oriented Solidity code and the Solana backend of *solang*. In addition, the approach exposed 29 internal compiler errors, including 9 unhandled exceptions in the official *solc* compiler. The results suggest that prompt-engineered LLMs can serve as a complementary strategy for differential compiler testing in blockchain environments.

Keywords

Solidity, Smart Contracts, Code Generation, Fuzzing, Differential Testing

1 Introduction

Blockchain technology and smart contracts enable the execution of distributed agreements without relying on centralized intermediaries [12, 14]. In Ethereum-based ecosystems, smart contracts are predominantly implemented in Solidity. Because deployed smart contracts are immutable, meaning their bytecode cannot be modified after deployment, defects introduced during compilation can persist and may result in financial losses [3, 11]. The reliability of the Solidity compiler (*solc*) is therefore an important component of smart contract security. Compilation defects may introduce inconsistencies between source code and generated bytecode, potentially leading to unintended behavior [1, 5, 15].

Fuzzing is widely used to identify failures in language infrastructure and compiler implementations. The technique automates the execution of a target program using large volumes of invalid, random, or semantically complex inputs to trigger unexpected behavior or crashes [7]. In compiler testing, the input consists of source code rather than conventional data. The effectiveness of fuzzing therefore depends on generating programs that are syntactically valid enough to reach deeper compiler stages, including optimization and code generation components [10].

A common challenge in compiler testing is the absence of a perfect oracle capable of determining the correct output for arbitrary programs. To address this limitation, prior work frequently adopts differential testing [8]. In this approach, the same input program is compiled or executed across multiple equivalent systems, such as different compiler versions, optimization configurations, or independent compiler implementations. Divergences in compilation results, execution behavior, or exit codes may indicate the presence of implementation defects in at least one of the evaluated targets.

While traditional fuzzing and differential testing techniques are widely used to evaluate compiler infrastructure, existing Solidity code generation approaches still face practical limitations. Many approaches rely on handcrafted templates, mutation strategies, or model fine-tuning. These constraints can reduce the structural diversity of generated contracts and limit the exploration of less common language interactions and corner cases.

This raises the question of whether prompt-engineered LLMs can generate Solidity contracts suitable for differential compiler testing. To investigate this question, we present SolForge, an automated fuzzing framework that uses combinatorial prompt generation to produce Solidity contracts for cross-compiler evaluation. The generated contracts are compiled across different compiler configurations to identify behavioral inconsistencies and internal compiler errors (ICEs).

The main contributions of this work are:

- A combinatorial prompt-generation strategy for producing Solidity contracts using pre-trained LLMs without fine-tuning;
- A cross-compiler differential testing pipeline targeting *solc* (standard and optimized) and *solang*;
- An empirical evaluation of the generated contracts in the context of compiler behavioral inconsistencies and ICE discovery;

- A qualitative analysis of failures related to memory-to-storage translation scenarios in the code generation pipeline of the official *solc* compiler.

The remainder of this paper is organized as follows. Section 2 discusses related work in compiler testing and LLM-assisted code generation. Section 3 describes the SolForge architecture and methodology. Section 4 presents the experimental results and qualitative analysis. Finally, Section 5 concludes the paper and discusses future work.

2 Related Works

Research on smart contract compiler testing generally follows two directions: traditional structured generation techniques and, more recently, LLM-assisted generation approaches. This section summarizes these categories and situates SolForge within the current compiler testing landscape.

2.1 Traditional Fuzzing and Structured Generation

Compiler testing in the Solidity ecosystem has traditionally relied on syntax-aware mutation and algorithmic generation. *Fuzzol* [9], for example, applies structural transformations such as AST manipulation and control-flow mutations to existing contract seeds in order to stress the *solc* compiler. Similarly, *Solsmith* [2] uses algorithmic construction to generate structurally valid Solidity programs. Both approaches evaluate *solc* by comparing compilation behavior between standard and optimized configurations.

Ma et al. [4] proposed *Erwin*, which extends the analysis to multiple compilers and static analyzers by exhaustively generating predefined templates. Unlike mutation-based strategies, *Erwin* relies on template completion to construct valid inputs. However, these approaches still rely on seed contracts or manually designed templates, which can limit structural diversity and exploration of less common semantic behaviors.

2.2 LLM-Assisted Generation in Smart Contracts

More recently, Large Language Models (LLMs) have been explored as input-generation mechanisms for smart contract testing. Tian et al. [13] introduced *DeSCDT*, a Transformer-based architecture for generating contracts for optimization-oriented differential testing in *solc*. The approach requires an offline training and semantic clustering stage to adapt the model to the target domain.

At a different abstraction level, *OpDiffer* [6] uses LLMs to synthesize EVM opcodes directly for differential testing across Ethereum Virtual Machine implementations rather than Solidity compilers themselves.

2.3 Positioning SolForge

Existing approaches reveal a separation between template- or mutation-based generators and LLM-driven testing strategies. Traditional approaches rely on handcrafted structures or predefined seeds, while current LLM-based methods either require additional training stages or focus primarily on bytecode execution.

SolForge combines elements from both directions by using combinatorial prompt generation with pre-trained LLMs to synthesize complete Solidity contracts without offline fine-tuning. In contrast to tools restricted to optimization-based comparisons within a single compiler, such as *fuzzol*, *Solsmith*, and *DeSCDT*, SolForge performs cross-compiler differential testing between *solc* and *solang* at the source-code level.

Table 1: Comparison of SolForge with existing compiler testing approaches.

Approach	Generation	Targets	Fine-Tuning
fuzzol [9]	AST Mutation	*solc* (std/opt)	No
Solsmith [2]	Algorithmic	*solc* (std/opt)	No
Erwin [4]	Templates	Multi-Compiler	No
DeSCDT [13]	Transformer LLM	*solc* (std/opt)	Yes
OpDiffer [6]	Opcode LLM	EVMs	No
SolForge	Prompt-Based LLM	*solc* / *solang*	No

3 The SolForge Approach

SolForge is a cross-compiler differential testing tool designed to automate the validation of Solidity compilers. Implemented in Haskell and containerized with Docker, the framework operates across three stages: (i) combinatorial prompt generation, (ii) contract synthesis using Large Language Models, and (iii) differential classification through an execution oracle. For code synthesis, SolForge integrates the DeepSeek Coder V2 model (16 billion parameters) [16], executed locally through an Ollama server.

Figure 1 summarizes the SolForge testing pipeline. The process begins with the Combinatorial Prompt Generator, which combines domain-specific tokens to produce semantic instructions for the LLM. These instructions are sent through a REST API to DeepSeek Coder V2, configured to generate Solidity contracts. The generated artifacts are first processed by a preliminary syntactic filter based on the official *solc* compiler. Contracts rejected during this stage are discarded before entering the differential testing phase. Validated contracts are then compiled concurrently across three configurations: *solc* in standard mode, optimized *solc*, and *solang*. The resulting outputs are analyzed by the classification oracle, which identifies divergences and internal compiler failures. *DIVERGENCE* and *CRASH* events are stored in persistent logs for later inspection. Next, we detail each stage of the *SolForge* pipeline.

3.1 Combinatorial Prompt Generation - Stage 1

To increase the diversity of generated contracts, SolForge implements a structured prompt generator in Haskell. Instead of relying on fixed code templates, the framework dynamically constructs natural language prompts by combining tokens from four semantic categories designed to exercise different compiler behaviors:

- **Functional Objectives:** Defines the high-level contract logic or architectural pattern (e.g., “a diamond proxy facet system”, “a deterministic deployment factory (CREATE2)”, or “a minimal optimistic rollup state transition”).

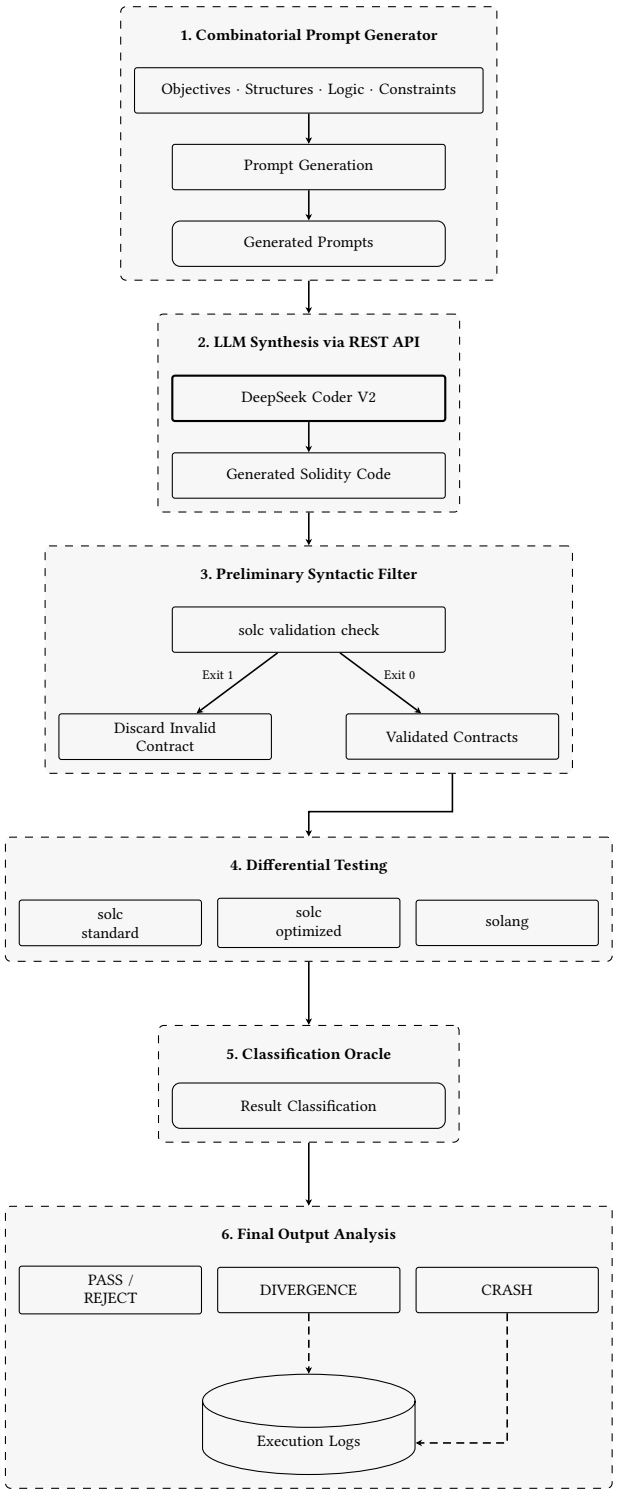

Figure 1: Overview of the *SolForge* pipeline for LLM-driven differential testing of Solidity compilers.

- **Data Structures:** Specifies storage and memory layouts intended to stress type resolution and state management mechanisms (e.g., "a nested mapping(uint => mapping(address => bool))", "a deeply nested struct", or "a sparse array emulated using mappings").
- **Computational Logic:** Introduces low-level or error-prone execution patterns frequently associated with optimizer and code-generation edge cases (e.g., "manipulates standard storage slots via assembly (sload/sstore)", "handles signed integer overflows intentionally", or "uses inline assembly to allocate memory manually").
- **Constraints:** Applies additional implementation restrictions or execution conditions (e.g., "uses the 'memory' keyword efficiently", "overrides internal functions across three inheritance layers", or "makes the constructor revert sometimes intentionally").

Table 2 presents examples of the domain tokens used during prompt generation.

Table 2: Examples of domain tokens used for combinatorial prompt generation.

Category	Example Token
Objective	a minimal optimistic rollup state transition
Structure	a mapping(bytes32 => mapping(uint256 => address))
Logic	encodes and decodes function selectors manually
Constraint	relies on Solidity's default value semantics

The prompt synthesis engine uses a recursive generation function that randomly selects one token from each category and interpolates them into a predefined instruction template:

> "Create a Solidity contract that implements **[Objective]**. The contract MUST use **[Structure]**. Crucially, the logic **[Logic]**. Finally, ensure the contract **[Constraint]**."

To avoid duplicate prompts and redundant inference cycles, the implementation stores generated combinations in `Data.Set`. The recursive process continues until the target corpus size is reached.

Given the current token pools, the Cartesian product of these categories yields 384,384 distinct semantic combinations. This combinatorial space increases input diversity and may expose additional compiler edge cases, low-level memory interactions, and implementation flaws.

3.2 Synthesis and Syntactic Validation - Stage 2 and 3

The synthesis stage operates continuously, with the Haskell core sending generated prompts to the LLM through a local REST API. DeepSeek Coder V2 is executed locally via Ollama with constrained inference parameters to improve reproducibility and reduce syntax degradation. Because of that, the model uses a context window (`num_ctx`) of 8,192 tokens and a sampling temperature fixed at 0.2.

Inference behavior is constrained through a dedicated System Prompt. The model is instructed to generate only Solidity source

code without conversational text or Markdown formatting. To standardize testing across compiler targets, all generated contracts must begin with the `// SPDX-License-Identifier: MIT` identifier and target `pragma solidity ^0.8.0;`. The prompt also explicitly prohibits the use of `import` statements, ensuring that generated contracts remain fully self-contained during differential testing.

Following code extraction, SolForge applies a preliminary syntactic validation step using the official *solc* compiler. Contracts rejected due to syntax errors (exit code 1) are immediately discarded without attempting automatic repair. This design choice avoids the overhead associated with repair heuristics and favors higher test throughput. In practice, the pipeline relies on repeated generation attempts from the model rather than implementing additional repair stages for invalid contracts. Consequently, only syntactically valid programs proceed to the differential testing stage.

Listing 1: System Prompt configuration wrapper within the SolForge orchestration layer.

```
You are a Solidity Smart Contract Generator.
Input: A requirement for a smart contract.
Output: ONLY the Solidity code.
IMPORTANT:
- Always start with // SPDX-License-Identifier: MIT
- Always use pragma solidity ^0.8.0;
- The code must be 100% SELF-CONTAINED.
  NO IMPORT STATEMENTS.
- Do NOT use 'import'.
```

Despite the constrained inference parameters and the restrictive system prompt, the preliminary syntactic filter rejected approximately 44% of the generated contracts. Inspection of the execution logs showed that these failures were usually associated with inconsistent language usage rather than malformed syntax.

One recurrent failure pattern involved hallucinated or deprecated EVM properties. When prompts contained low-level execution constraints, the model occasionally attempted to access unsupported Solidity members such as `tx.gaslimit`, which is not available in Solidity `^0.8.0`. Listing 2 presents an example of this behavior.

Listing 2: Example of LLM syntax rejection due to hallucinated global variables in contract_008.sol.

```
// Rejected by solc: Member "gaslimit"
// not found or not visible after
// argument-dependent lookup in tx.
require(
    tx.gaslimit > 0,
    "Transaction gas limit must be greater than zero"
);
```

Another common rejection pattern involved incorrect library reconstruction and scope handling. Since the system prompt prohibits external `import` statements, the model occasionally attempted to recreate architectural abstractions internally but produced invalid type associations. Listing 3 illustrates a case where the `'using ... for'` directive is incorrectly applied to an interface instead of a valid library declaration.

Listing 3: Example of LLM syntax rejection due to structural hallucination of library scopes in contract_103.sol.

```
// Rejected by solc: Library name expected.
using LegacyInterface for LegacyInterface.Node;
```

Instead of implementing AST-level repair heuristics to correct these invalid artifacts, SolForge discards them and continues the generation process. This decision prioritizes throughput and allows the framework to focus on contracts that successfully reach the differential testing oracle.

3.3 Differential Testing and Classification Oracle - Stage 4 and 5

The central testing module evaluates validated contracts across three compilation configurations: *solc* in standard mode, optimized *solc*, and *solang*. The classification oracle implements a deterministic consensus function over these targets, as formalized in Algorithm 1.

For each execution, SolForge records the exit code together with the standard output and error streams. Each compiler result is mapped to one of three statuses: Ok (successful compilation), Rej (syntax rejection via exit code 1), or Err (internal failures or abnormal termination).

The final verdict is derived from the combination of these statuses, as summarized in Table 3. Internal compiler failures immediately trigger a Crash verdict. If all targets compile successfully, the oracle returns Pass. If all targets reject the contract, the result is classified as Reject. Any disagreement among the evaluated compilers is classified as Divergence.

This rule set normalizes heterogeneous compiler outputs into a unified verdict. The oracle stores Crash and Divergence events together with execution traces, prompts, and compiler error outputs for later inspection and qualitative analysis.

Table 3: Decision matrix governing the SolForge classification oracle.

solc (std)	*solc* (opt)	*solang*	Oracle Verdict
OK	OK	OK	PASS
REJ	REJ	REJ	REJECT
OK	OK	REJ	DIVERGENCE
OK	REJ	OK	DIVERGENCE
REJ	OK	OK	DIVERGENCE
OK	REJ	REJ	DIVERGENCE
REJ	OK	REJ	DIVERGENCE
REJ	REJ	OK	DIVERGENCE
ERR	*	*	CRASH
*	ERR	*	CRASH
*	*	ERR	CRASH

Note: * denotes any status ∈ {OK, REJ, ERR}. The CRASH rule maintains absolute priority over all other conditions.

4 Experimental Evaluation and Results - Stage 6

The experimental evaluation was conducted in a containerized environment using an NVIDIA RTX 6000 Ada GPU (96GB VRAM) to host the DeepSeek Coder V2 model. The generated corpus was evaluated against three compilation targets: *solc* v0.8.26 (standard and optimized configurations) and *solang* v0.3.3.

Algorithm 1 SolForge Differential Testing Oracle

Require: Solidity contract C
Ensure: Oracle verdict $V \in \{\text{Pass}, \text{Reject}, \text{Divergence}, \text{Crash}\}$

$r_{std} \leftarrow \text{Run}(\texttt{solc std}, C)$
$r_{opt} \leftarrow \text{Run}(\texttt{solc opt}, C)$
$r_{solang} \leftarrow \text{Run}(\texttt{solang}, C)$
$s_{std} \leftarrow \text{ClassifyStatus}(r_{std})$
$s_{opt} \leftarrow \text{ClassifyStatus}(r_{opt})$
$s_{solang} \leftarrow \text{ClassifyStatus}(r_{solang})$
if $\text{Err} \in \{s_{std}, s_{opt}, s_{solang}\}$ **then**
    return Crash
else if $s_{std} = s_{opt} = s_{solang} = \text{Ok}$ **then**
    return Pass
else if $s_{std} = s_{opt} = s_{solang} = \text{Rej}$ **then**
    return Reject
else
    return Divergence
end if

function ClassifyStatus(r)
    if ContainsCrashIndicator($r.stderr$) **then**
        return Err
    else if $r.exitCode = 0$ **then**
        return Ok
    else if $r.exitCode = 1$ **then**
        return Rej
    else
        return Err
    end if
end function

function ContainsCrashIndicator($stderr$)
    $K \leftarrow$ {"internal compiler error", "panic", "segmentation fault", "stack overflow", "Unimplemented feature"}
    return $\exists k \in K : k \subseteq stderr$
end function

4.1 Generation and Validation Performance

Starting from a corpus of 3,000 combinatorial prompts, the framework executed 2,425 generation attempts in approximately 1 hour and 37 minutes (~ 2.4s per attempt). Of this total, 1,357 contracts (55.96%) passed the syntactic validation stage.

This result indicates that pre-trained LLMs can generate a sufficient volume of syntactically valid Solidity contracts for differential testing workflows, even without fine-tuning. In this setting, continuous generation cycles compensate for invalid outputs by maintaining a steady throughput of valid contracts for the testing stage.

4.2 Architectural Compatibility Evaluation

Submitting the validated corpus of 1,357 contracts to the differential testing oracle produced 1,070 anomalous execution events. To avoid conflating compiler defects with expected platform differences, the observed anomalies were categorized by root cause. Table 4 summarizes the resulting classification.

Table 4: Detailed classification of discovered anomalies and execution divergences.

Category	Root Cause / Description	Qty.
Unanimous Consensus	Successful compilation across all targets (Pass).	287
Unanimous Rejection	Syntax/semantic failure across all targets.	0
Architectural Incompatibility	EVM-specific logic rejected by *solang*.	1,040
Optimization Inconsistency	Divergence between *solc* (std) and *solc* (opt).	1
Internal Compiler Error (ICE)	Unhandled exceptions and assertion failures.	29
Total Contracts		**1,357**

Most anomalous cases (1,040 contracts) were classified as architectural incompatibilities. Inspection of these results indicated that they primarily reflect differences between Ethereum's EVM model and Solana's BPF-based execution environment rather than implementation defects. These cases involved EVM-specific features, including Inline Assembly/Yul constructs, that do not have direct equivalents in *solang*'s Solana backend.

One representative incompatibility was identified during the evaluation of `contract_1893.sol`. While both standard and optimized versions of *solc* compiled the contract successfully, *solang* rejected the same artifact when targeting Solana. The divergence originated from the use of low-level Yul instructions, specifically `sstore` and `sload`. Listing 4 shows the generated code fragment responsible for the incompatibility.

Listing 4: EVM-specific inline assembly accepted by solc but rejected by solang due to architectural incompatibility.

```
contract StorageRentDemo {
    mapping(address => uint256)
        private storageSlots;

    function setStorageSlot(uint256 value) public {
        assembly {
            sstore(storageSlots.slot, value)
        }
    }

    function getStorageSlot() public view
      returns (uint256) {
        uint256 value;
        assembly {
            value := sload(storageSlots.slot)
        }
        return value;
    }
}
```

This behavior reflects differences between the EVM execution model and Solana's account-based storage architecture. In *solc*, Yul instructions such as `sstore` map directly to EVM storage operations.

In contrast, *solang* targets Solana, where contract state is managed through external data accounts rather than a unified storage trie. As a result, low-level EVM assembly instructions cannot be translated directly into the Solana execution model.

Additional incompatibilities were associated with EVM-specific features such as `msg.value` and consensus-related properties including `block.difficulty`.

4.3 Compiler Defect Discovery

After excluding expected architectural incompatibilities, the oracle identified 30 potentially relevant compiler anomalies: one optimization inconsistency and 29 internal compiler errors (ICEs). The optimization inconsistency occurred in *solc*, where a contract accepted by the standard compiler configuration was rejected by the optimized configuration due to the use of the `msize` instruction. The optimized configuration reported that `msize` cannot be used when the Yul optimizer is enabled because the optimizer may change its semantics. Consequently, this case was classified as a configuration-dependent rejection rather than an internal compiler error.

The 29 ICE cases were further analyzed to determine the affected compiler components. Among them, 20 were isolated *solang* failures, in which both *solc* configurations successfully compiled the contract while *solang* terminated with an internal error. Eight cases were isolated *solc* failures, where both the standard and optimized *solc* configurations failed with the same internal error while *solang* either rejected the program normally or compiled it successfully. One additional case affected both compiler families simultaneously, with *solc* failing in both configurations and *solang* terminating with a stack overflow. Notably, no ICE was observed in only one *solc* configuration; whenever *solc* failed internally, the failure was consistently reproduced in both standard and optimized compilation.

The *solc*-related ICEs were dominated by a single code-generation pattern. Eight contracts triggered an `UnimplementedFeatureError` in `ArrayUtils.cpp`, caused by copying dynamically sized memory arrays of structs into storage. The remaining *solc* ICE involved fixed-point types, failing in `YulUtilFunctions.cpp` with the diagnostic "Fixed point types not implemented." In contrast, the *solang* ICEs were more heterogeneous, spanning multiple compiler components, including Solana-target code generation, semantic analysis, ABI encoding, constant folding, and stack overflow failures. Although the largest cluster consisted of eleven contracts that reached a "not implemented" panic in `emit/solana/target.rs:1937`, the remaining failures were distributed across several distinct components and failure modes.

At the implementation level, the dominant *solc* issue appears related to code generation for assignments between EVM memory structures and `storage`-backed dynamic types. In these cases, the compiler aborts with an internal `UnimplementedFeatureError` instead of producing bytecode or issuing a regular source-level diagnostic.

The following case studies were selected because they represent the dominant *solc* ICE pattern observed during the evaluation. Of the nine contracts that triggered an internal error in *solc*, eight failed in `ArrayUtils.cpp` during the lowering of dynamically sized arrays of structs from memory to storage. Contracts #613 and #1026 were chosen as representative examples because they reach this same compiler failure through different semantic constructions. The first case implements a compact state-reset operation over a dynamic array of `Player` structs, whereas the second combines nested structs, temporary in-memory manipulation, and a subsequent write-back to persistent storage. Together, they illustrate that the observed ICE is not tied to a specific contract template, but to a broader unsupported code-generation pattern in the *solc* compiler.

4.3.1 Case Study 1: State Resetting in a Lottery System. In attempt #613, the model generated a lottery contract containing a dynamic storage array of `Player` structs. After selecting a winner, the contract attempted to reset the lottery state by assigning a newly instantiated memory array to the persistent storage variable, as illustrated in Listing 5.

Listing 5: Semantic snippet of Contract #613 illustrating a state reset pattern that crashes the compiler.

```
contract Lottery {
    struct Player {
        address addr;
        uint256 amount;
    }
    Player[] public players; // Dynamic Storage Array

    function pickWinner() public {
        // ... (Winner selection and
        // transfer logic omitted) ...

        // Trigger: Memory-to-Storage assignment
        // of an array of structs
        players = new Player[](0);
    }
}
```

Although syntactically valid, processing line 12 of Listing 5 causes the *solc* compiler to terminate with an internal `UnimplementedFeatureError`, shown in Listing 6.

Listing 6: solc standard error stream (stderr) capturing the unhandled C++ exception for Contract #613.

```
Unimplemented feature:
/solidity/libsolidity/codegen/ArrayUtils.cpp(227): Throw in
function solidity::frontend::ArrayUtils::copyArrayToStorage(...)
Dynamic exception type: boost::wrapexcept<solidity::langutil::
UnimplementedFeatureError>
std::exception::what: Copying of type struct Lottery.Player
memory[] memory to storage not yet supported.
```

4.3.2 Case Study 2: In-Memory Sorting and State Updates. In attempt #1026, the generator produced a more complex contract implementing an optimistic rollup state transition combined with an in-memory sorting routine. To avoid repeated storage operations, the generated logic copied the state into memory, manipulated the temporary buffer, and attempted to write the resulting structure back into storage.

Listing 7 shows the relevant code fragment.

Listing 7: Semantic snippet of Contract #1026 executing in-memory sorting before a storage update.

```
contract OptimisticRollupStateTransition {
    struct NestedStruct {
        uint256 value;
        NestedStruct[] subStructs;
    }
    mapping(address => NestedStruct) public stateMap;

    function transitionState(
        address user, uint256 newValue
    ) public {
        NestedStruct storage currentState =
            stateMap[user];

        // Array instantiated in memory
        NestedStruct[] memory tempSubStructs =
            new NestedStruct[](
                currentState.subStructs.length
            );

        // ... (Bubble sort logic applied to
        // tempSubStructs omitted) ...

        // Trigger: Complex nested assignment
        // to persistent storage
        currentState.subStructs = tempSubStructs;
    }
}
```

As observed in the previous case study, the assignment operation involving `currentState.subStructs = tempSubStructs;` triggers an internal failure in the *solc* code generation pipeline, producing the same abrupt termination pattern.

These results indicate that the generation pipeline can produce contract structures complex enough to reach backend compiler failures in Solidity compilers. Multiple independently generated contracts converged on the same implementation weakness through different semantic constructions. While this evaluation focuses on feasibility rather than comparative superiority against existing fuzzers, the discovered ICEs demonstrate that prompt-engineered LLMs can generate relevant inputs for differential compiler testing.

4.4 Threats to Validity

To support the interpretation of the experimental results, this section discusses potential threats to validity across four dimensions.

Internal Validity: One limitation originates from the use of the official *solc* compiler as a preliminary syntactic filter. Because invalid contracts are discarded before differential testing, the resulting corpus is naturally biased toward Ethereum-compatible structures. Consequently, incompatibilities observed in *solang* may be amplified by the composition of the filtered corpus. In addition, although the sampling temperature was fixed at 0.2, LLM inference remains probabilistic, and repeated executions may generate slightly different contract distributions.

Construct Validity: The evaluation emphasizes qualitative defect discovery rather than formal diversity metrics such as AST coverage or opcode distribution. As a result, the approach's effectiveness is assessed primarily by its ability to expose compiler failures and behavioral divergences.

External Validity: The evaluation was limited to *solc* v0.8.26 and *solang* v0.3.3. The observed behaviors may not generalize to other smart contract compilers or execution environments. In addition, the synthesis pipeline relied exclusively on DeepSeek Coder V2 16B. Different language models or training corpora may produce different syntactic acceptance rates and contract structures.

Conclusion Validity: Since LLM-based generation is inherently probabilistic, repeated executions may expose different defect distributions and behavioral patterns. The current evaluation is exploratory and focuses primarily on feasibility and qualitative analysis rather than statistical generalization.

5 Conclusion and Future Works

This paper presented SolForge, a generative AI-assisted fuzzing approach for the comparative validation of Solidity compilers. The empirical evaluation suggests that the combinatorial use of prompts together with pre-trained models, without requiring fine-tuning, provides a viable strategy for differential compiler testing. From a corpus of 1,357 syntactically valid contracts, the approach identified 1,040 cross-target disagreements between *solc* and the Solana backend of *solang*, one optimization-dependent inconsistency in *solc*, and 29 internal compiler errors (ICEs), including 9 unhandled exceptions in the official Ethereum Foundation compiler.

Although most divergences were due to expected architectural incompatibilities between EVM- and Solana-oriented compilation targets, the generated contracts were still capable of isolating genuine compiler defects. In particular, the discovered ICEs indicate that prompt-engineered LLMs can generate programs that reach backend compiler components and IR-generation paths, exposing edge cases that can complement traditional mutation-based strategies.

The current architecture still presents operational limitations. The strategy of immediately discarding syntactically invalid contracts simplifies the pipeline and preserves testing throughput, but the observed 44% rejection rate introduces nontrivial computational overhead. Additionally, the absence of a persistent execution state limits the scalability and fault tolerance of long-running testing campaigns. Overall, the results suggest that LLM-assisted synthesis can serve as a complementary input-generation strategy for differential compiler testing in blockchain compiler ecosystems.

Future work will focus on improving both the robustness and the analysis capabilities of the framework. At the infrastructure level, we intend to introduce incremental checkpoint mechanisms to support interruption recovery and large-scale distributed executions. We also plan to investigate Retrieval-Augmented Generation (RAG) techniques based on formal Solidity specifications and compiler documentation to improve syntactic validity and expand the generation of semantically complex contracts. Finally, future studies should include comparative evaluations against grammar-based and mutation-based fuzzers to better characterize the input diversity and defect-discovery capabilities of LLM-generated programs.

Artifact Availability

The complete implementation of SolForge, including the prompt generation pipeline, generated corpus, and evaluation artifacts is publicly available at https://github.com/gabrielsalvi/solforge to support the reproducibility of the experiment presented in this paper.

References

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[2] Lantian Li, Zhihao Liu, and Zhongxing Yu. 2025. Solsmith: Solidity Random Program Generator for Compiler Testing. *arXiv preprint arXiv:2506.03909* (2025).

[3] Ruichao Liang, Jing Chen, Cong Wu, Kun He, Yueming Wu, Ruochen Cao, Ruiying Du, Ziming Zhao, and Yang Liu. 2025. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing. *IEEE Transactions on Information Forensics and Security* (2025).

[4] Haoyang Ma, Alastair F Donaldson, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2025. Bounded Exhaustive Random Program Generation for Testing Solidity Compilers and Analyzers. *arXiv preprint arXiv:2503.20332* (2025).

[5] Haoyang Ma, Wuqi Zhang, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2024. Towards Understanding the Bugs in Solidity Compiler. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM, 1312–1324. doi:10.1145/3650212.3680362

[6] Jie Ma, Ningyu He, Jinwen Xi, Mingzhe Xing, Haoyu Wang, Ying Gao, and Yinliang Yue. 2025. OpDiffer: LLM-Assisted Opcode-Level Differential Testing of Ethereum Virtual Machine. *arXiv preprint arXiv:2504.12034* (2025).

[7] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[8] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[9] Charalambos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis, and Dimitris Mitropoulos. 2023. Syntax-Aware Mutation for Testing the Solidity Compiler. In *European Symposium on Research in Computer Security*. Springer, 327–347.

[10] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[11] Eugenia Politou, Fran Casino, Efthimios Alepis, and Constantinos Patsakis. 2019. Blockchain mutability: Challenges and proposed solutions. *IEEE Transactions on Emerging Topics in Computing* 9, 4 (2019), 1972–1986.

[12] Nick Szabo. 1996. Smart contracts: building blocks for digital markets. *Extropy* 16, 2 (1996), 18.

[13] Zhenzhou Tian, Fanfan Wang, Yanping Chen, and Lingwei Chen. 2024. Differential testing solidity compiler through deep contract manipulation and mutation. *Software Quality Journal* 32, 2 (2024), 765–790.

[14] Gautami Tripathi, Mohd Abdul Ahad, and Gabriella Casalino. 2023. A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges. *Decision Analytics Journal* 9 (2023), 100344.

[15] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[16] Qihao Zhu et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).