Bounded Exhaustive Random Program Generation for Testing Solidity Compilers

Haoyang Ma[*]
The Hong Kong University of Science and Technology
China
haoyang.ma@connect.ust.hk

Alastair F. Donaldson
Imperial College London
United Kingdom
alastair.donaldson@imperial.ac.uk

Qingchao Shen
Tianjin University
China
qingchao@tju.edu.cn

Yongqiang Tian
Monash University
Australia
yongqiang.tian@monash.edu

Junjie Chen
Tianjin University
China
junjiechen@tju.edu.cn

Shing-Chi Cheung
The Hong Kong University of Science and Technology
China
scc@cse.ust.hk

Abstract

By July 2025, smart contracts collectively manage roughly $120 billion in assets. With Solidity remaining the dominant language for smart contract development, the correctness of Solidity compilers has become critically important. However, Solidity compilers are prone to bugs, with a recent study revealing that combinations of qualifiers in Solidity programs are the primary cause of compiler crashes, accounting for 40.5% of all historical crashes. While random program generators are widely used for compiler testing, they may be less effective at finding Solidity compiler bugs because they explore the unbounded space of possible programs rather than concentrating on the specific subspace related to bug-prone qualifiers. A promising idea for finding qualifier-related bugs is to bound the search space based on empirical evidence of where such bugs are likely to occur, specifically focusing test generation to target subspaces with rich combinations of qualifiers. To address this, we propose *bounded exhaustive random program generation*, a novel approach that dynamically bounds the search space, enhancing the likelihood of uncovering Solidity compiler bugs. Specifically, our method bounds the search space by generating valid program templates that abstract programs that use bug-prone qualifiers, and then uses these templates as a basis for compiler testing through exhaustive enumeration of suitable qualifiers. Mechanisms are devised to address technical challenges regarding validity and efficiency.

We have implemented our novel generation approach in a new tool, Erwin. We have used Erwin to find and report 26 bugs across two Solidity compilers, `solc` and `solang`, and one Solidity static analyzer, `slither`. Among these, 23 were previously unknown, 18 have been confirmed, and 10 have been fixed. Evaluation results demonstrate that Erwin outperforms state-of-the-art Solidity fuzzers in bug detection and complements developer-written test suites by covering 4,599 edges and 14,824 lines of the `solc` compiler that were missed by `solc`'s unit tests.

[*]This work was completed during Haoyang's visit to the FastPL group at Imperial College London led by Prof. Alastair F. Donaldson.

ACM Reference Format:
Haoyang Ma, Alastair F. Donaldson, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2026. Bounded Exhaustive Random Program Generation for Testing Solidity Compilers. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773237

1 Introduction

Solidity compilers, which translate the widely used smart contract language Solidity into executable bytecode for the Ethereum Virtual Machine, are essential tools for the blockchain and smart contract ecosystem: Solidity is the predominant language for smart contracts [29], with around $120 billion in assets [2] managed via Solidity smart contracts as of July 2025. Therefore, ensuring the correctness of Solidity compilers by generating Solidity test programs is of utmost importance.

However, recent research [29] reveals that Solidity compilers are susceptible to bugs. In particular, combinations of qualifiers in Solidity are responsible for triggering 40.5% of all recorded compiler crashes, making them the leading cause of such failures. These compiler crashes can result in system malfunctions, underscoring the importance of developing effective compiler testing techniques.

Random program generators [14, 16, 19, 22, 24–26, 28, 36–38, 40] are prevalent in compiler testing. They produce syntactically valid and well-typed test programs from scratch in a randomised fashion. Because these well-formed programs pass the compiler's front-end checks, they are effective at exercising the optimization and code generation phases of the compiler. However, the inherent randomness of these tools causes program generators to produce programs without targeted focus within the vast, unbounded search space defined by the language. This leads to a phenomenon known as *opportunism* [41]: generators depend on chance encounters with bug-triggering code rather than systematically exploring all possibilities within a relevant subspace. Because compiler bugs tend to cluster in specific subspaces (*e.g.*, the subspace defined by qualifier combinations in Solidity), such opportunistic generation can significantly delay the discovery of bug-triggering test programs, or make the probability of finding certain bugs vanishingly small.

The template-based testing method [38–41] demonstrates that meticulously crafted templates have bug-detection potential. The template is an incomplete test program with placeholders to be instantiated. Each template corresponds to a significantly smaller search space, thereby largely mitigating the opportunism. Nevertheless, this approach relies on the manual construction of templates, or the extraction of templates from high-quality seed programs. The scarcity of high-quality templates therefore limits the effectiveness of the approach.

An intuitive way to address this limitation is to generate templates dynamically during random program generation and then enumerate all valid test programs from these templates. This process naturally strikes a balance between broad exploration of random generation and the exhaustive enumeration of valid test programs, effectively mitigating the weaknesses of pure randomness and combining the strengths of both approaches. We call this *bounded exhaustive random program generation.*

Effective template design is crucial because low-quality templates define subspaces that are unlikely to reveal bugs, leading to inefficient enumeration. One way to improve template quality is to focus on language features that are closely associated with compiler bugs. For example, prior studies have shown that type combinations and operations in Java [13], data types and tensor shapes in deep learning models [32], and qualifier combinations in Solidity [29] strongly correlate with compiler bugs. Notably, as discussed above, certain qualifier combinations in Solidity account for up to 40.5% of all historical Solidity compiler crashes [29]. To effectively tailor bounded exhaustive random program generation for Solidity while preserving generality, we represent these qualifiers as placeholders within templates.

With a well-designed generator of useful templates, the new generation approach comprises two stages: (1) Random generation of templates and (2) Bounded exhaustive enumeration across the set of valid values for each placeholder within the generated template. To guarantee the validity of the generated templates, we maintain a solvable constraint set related to placeholders during template generation. To enhance the efficiency of the exhaustive search and boost the generator's throughput, we have developed a constraint reduction algorithm. This approach significantly reduces the number of constraints by eliminating those that do not affect the overall semantics, thereby improving the efficiency of the exhaustive enumeration process.

We have implemented our approach in the context of Solidity, a mainstream smart contract language for the Ethereum blockchain, in a tool called Erwin. Over six months of intermittent testing, Erwin discovered 26 bugs in two Solidity compilers (`solc`, `solang`) and one static analyzer (`slither`), with 23 previously-unknown, 18 confirmed, and 10 fixed. Within 20 days, Erwin found 18 compiler bugs, more than twice the number detected by the leading fuzzers ACF and `Fuzzol`, with 16 unique to Erwin. Compared to Solidity-Smith, a configuration of Erwin which does not use bounded exhaustive generation, Erwin detected six additional bugs under the same conditions, providing evidence that the bounded exhaustive generation component is valuable for bug detection. Furthermore,

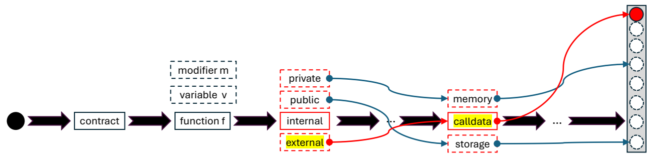


Figure 1: Random program generation

programs generated by Erwin covered 4,599 edges and 14,824 lines not reached by `solc`'s unit tests, demonstrating its effectiveness in producing high-quality and complementary test cases.

Contributions. Our contributions are as follows: **1** The concept of bounded exhaustive random program generation, which combines the strengths of random program generators and template-based methods and addresses the inefficiency of random program generators caused by opportunism. **2** The design and implementation of Erwin, a bounded exhaustive random program generator for Solidity, which has successfully identified 23 previously unknown bugs across two Solidity compilers and one Solidity static analyzer. Erwin is available at https://github.com/haoyang9804/Erwin. **3** An extensive and thorough evaluation of Erwin, including bug-finding capability, code coverage, comparison with state-of-the-art Solidity fuzzers, throughput analysis, and ablation study.

2 Motivation and Background

2.1 Motivation

Figure 1 illustrates the random program generation process. The black arrows show a sequence of generation steps leading to a bug-missing test program, represented by a white circle with a dashed outline. In Solidity test program generation, several steps related to qualifier selection, highlighted by red blocks in the figure, are particularly important. At these steps, the generator must choose among multiple options. Traditional random program generation treats these key steps the same as others, selecting choices at random. As a result, the probability of generating a test program that will trigger a particular bug can be extremely low, as the necessary bug-triggering conditions are often difficult to satisfy.

The important generation steps together form a focused search space for relevant bugs. Bounded exhaustive random program generation dynamically concentrates on this subspace during runtime, thoroughly exploring the bug-relevant region. Put differently, it can exhaustively generate all valid qualifier combinations in this example, meeting the bug-triggering conditions (illustrated as the thin red arrows) and ensuring that the bug-triggering test program represented by the red circle is not missed.

Figure 1 visualizes several generation steps for the bug-triggering test program for a documented bug [4] shown in Figure 2. This bug occurs when a public function g calls an external function f that includes a return statement stored in calldata. Even minor valid modifications to these qualifiers, such as replacing calldata with memory, will prevent the bug from being triggered. Relying solely on grammar integration in the random generator may result in only a minimal chance of producing bug-triggering programs. As shown in the evaluation (§5.3), traditional random generation

```
contract CalldataTest {
  function f() external returns (int[] calldata x) {
    x = x;
    return x;
  }
  function g() public {
    this.test();
  }
}
```

Figure 2: Bug-triggering test program for GitHub issue #9134 [4]

failed to trigger this bug even after 20 days, whereas the bounded exhaustive approach succeeded.

2.2 Solidity

Solidity is a high-level programming language for developing smart contracts on the Ethereum blockchain. Solidity features inheritance, user-defined types, and various storage locations (memory, storage, calldata), making it suitable for complex smart contract development. Key Solidity tools include the solc compiler [9], solang compiler [8], and slither static analyzer [5].

A comprehensive study of Solidity compiler bugs [29] reveals that Solidity compilers are prone to bugs with severe consequences due to the immutable nature of smart contracts. Importantly, the study identifies several highly bug-related qualifiers in Solidity programs, including visibility, storage locations, data types, and mutability. These qualifiers can be used systematically to form bug-triggering templates, as they frequently appear in bug-triggering conditions and significantly influence the behavior of the compiler. Therefore, we implement Erwin as a bounded exhaustive random program generator for Solidity to study the effectiveness of bounded exhaustive random program generation in detecting bugs in Solidity compilers and analyzers.

3 Approach

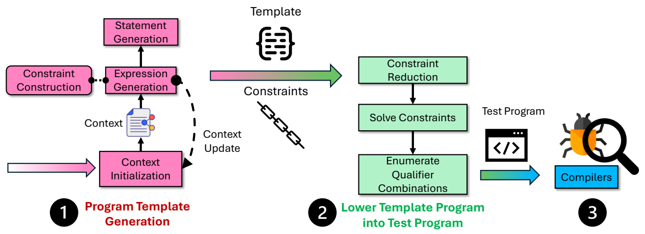


Figure 3: Workflow of Erwin

The workflow of Erwin is depicted in Figure 3. It contains three components: (1) Program Template generation; (2) Program Template lowering; and (3) Testing.

Program Template generation. Program Template generation begins by initializing a template with randomly-created valid declarations, such as class and function declarations, while leaving their qualifiers unassigned for later completion. The main motivation for emphasizing qualifiers in this paper is that they are both broadly applicable and closely tied to bugs, spanning general-purpose languages (e.g., Java [13]) as well as domain-specific languages (e.g., deep learning models [32] and Solidity [29]). Subsequently, expressions are generated within appropriate scopes, concurrently building constraints for the qualifier placeholders. The process concludes with the generation of statements. We delve into the details of the program template generation in §3.2.

Program Template lowering. A program template defines a highly bug-related subspace of the unbounded search space of all test programs. Erwin then systematically explores this subspace, ensuring no opportunities are missed for detecting related bugs. Specifically, Erwin begins by reducing constraints. It then enumerates all valid qualifier combinations that satisfy all remaining constraints to transform the program template into concrete test programs. This step exemplifies the spirit of *bounded exhaustiveness*. We further discuss the details in §3.3.

Testing. Finally, the generated test programs are fed to compilers and analyzers to trigger crashes and hangs.

3.1 Preliminaries

$$
\begin{aligned}
\langle p \in \mathit{Program}\rangle &::= \overline{d} \\
\langle d \in \mathit{Decl}\rangle &::= \overline{\mathcal{Q}}\ v \mid \text{error } v(\overline{\overline{\mathcal{Q}}\ v}) \mid \text{event } v(\overline{\overline{\mathcal{Q}}\ v}) \mid \text{struct}\{\overline{\overline{\mathcal{Q}}\ v}\} \\
&\mid \text{modifier } v(\overline{\overline{\mathcal{Q}}\ v})\{\overline{s}\} \mid \text{function } (\overline{\overline{\mathcal{Q}}\ v})\ \overline{\mathcal{Q}}\ \{\overline{s}\} \mid \text{class } \{\overline{d}\} \\
\langle s \in \mathit{Stmt}\rangle &::= e \mid \overline{\mathcal{Q}}\ v = e \mid \text{if } (e)\ \{\overline{s}\} \text{ else } \{\overline{s}\} \mid \text{for } (e; e; e)\ \{\overline{s}\} \\
&\mid \text{do } \{\overline{s}\} \text{ while } (e) \mid \text{while } (e)\ \{\overline{s}\} \mid \text{return } e \mid \text{emit } e \mid \text{revert } e \\
\langle e \in \mathit{Exp}\rangle &::= \text{literal} \mid v \mid v\ op_a\ e \mid e\ op_b\ e \mid op_u\ e \mid \text{new } v \mid e\ ?\ e\ :\ e \mid v(\overline{e}) \mid e[e] \mid e.v \\
\overline{\mathcal{Q}} &::= \text{a sequence of qualifier placeholders, each of which is denoted as } \mathcal{Q} \\
v &::= \text{an identifier representing the name of a variable, method or contract} \\
op_a,\ op_b,\ op_u &::= \text{identifiers representing assignment, binary, unary operators}
\end{aligned}
$$

Figure 4: Syntax of templates, where $\overline{x}$ denotes a sequence of elements each of the form x

Figure 4 defines the syntax of templates. A variable or method can be qualified by a sequence of *qualifier placeholders*, each of which can take on a qualifier during the template lowering process (§3.3). Specifically tailored for Solidity, a qualifier placeholder $\mathcal{Q}$ can be $\mathscr{T}$, $\mathscr{S}$, $\mathscr{V}$, and $\mathscr{M}$, each of which represents the qualifier placeholder for data type, storage location, visibility, and mutability, respectively. The collection of qualifiers that can be assigned to a qualifier placeholder is referred to as the *qualifier placeholder codomain*, symbolized by τ. Figure 5 lists the initial codomain of each kind of qualifier placeholder.

$$
\begin{aligned}
\tau_{\mathscr{T}} &::= \{\text{bool, address, address payable, string, integer type, struct type,} \\
&\quad\ \text{contract type, array type, mapping type}\} \\
\tau_{\mathscr{S}} &::= \{\text{memory, storage pointer, storage reference, calldata}\} \\
\tau_{\mathscr{V}} &::= \{\text{public, private, internal, external}\} \\
\tau_{\mathscr{M}} &::= \{\text{pure, view, payable, nonpayable}\}
\end{aligned}
$$

Figure 5: Qualifier codomain initialization in Solidity

3.2 Program Template Generation

Context Initialization. Erwin's generation approach is hierarchical, starting with the creation of contract declarations. This is followed by the declarations of contract members, such as member variables and member functions, as well as parameter declarations, and so forth. These declarations, along with their scopes and associated qualifier placeholders, define the context of the program.

Expression Generation and Constraint Construction. Expression generation proceeds top-down. Specifically, if Erwin intends to generate an expression e from the context, it first analyzes the necessary sub-expressions $\overline{e_s}$ and gathers the required constraints among e's qualifier placeholders and those of $\overline{e_s}$. Subsequently, Erwin generates $\overline{e_s}$ based on the constraints. Finally, Erwin collects all the associated constraints in the generation of e and pushes them into the constraint set $\mathbb{C}$. We present this constraint-based approach in a manner inspired by work on constraint-based type inference [31, Chapter 22] adapted to include subtype constraints [17].

Definition 3.1 (Constraint Set). A constraint set $\mathbb{C}$ comprises:

A collection of relations $\{\mathcal{Q}_i \bowtie \mathcal{Q}_j\}$, where $\mathcal{Q}_i$ and $\mathcal{Q}_j$ are qualifier placeholders and $\bowtie \in \{\doteq, \lessdot\}$. Here, $\doteq$ and $\lessdot$ signify "is the same as" and "is more restricted than", respectively.

A collection of codomain restrictions of qualifier placeholders such as $\tau_{\mathcal{Q}} \leftarrow \{\ldots\}$ or $Q_i \lessdot \mathcal{Q} \lessdot Q_j$ where Q_i and Q_j are qualifiers. The first restriction assigns a new codomain to $\tau_{\mathcal{Q}}$ where the second sets the upper bound and lower bound of $\tau_{\mathcal{Q}}$. This kind of constraint directly modifies the codomain of a qualifier placeholder.

Like the traditional subtyping relation, $\lessdot$ is both reflexive and transitive. It conveys different meanings across qualifier kinds. **1** *Data types*: $\mathcal{T}_1 \lessdot \mathcal{T}_2$ means that $\mathcal{T}_1$ is a subtype of $\mathcal{T}_2$, *e.g.*, $\text{int}_{16} \lessdot \text{int}_{32}$. **2** *Storage locations*: $\mathcal{S}_1 \lessdot \mathcal{S}_2$ means that data stored in $\mathcal{S}_1$ can be transferred to $\mathcal{S}_2$, *e.g.*, calldata $\lessdot$ memory. **3** *Visibilities* ($\mathcal{V}$) and *mutabilities* ($\mathcal{M}$) are not governed by relations.

The constraint set $\mathbb{C}$ is initialized by propagating the initial codomains of all kinds of qualifier placeholders (as defined in Figure 5) to their respective instances after context initialization. The scope can refine $\tau_{\mathcal{S}_i}$ during constraint set initialization. For example, $\tau_{\mathcal{S}_i}$ ::= {storage reference} if $\mathcal{S}_i$ resides in a contract's member scope. Such language-specific refinements are intentionally excluded from formalization to preserve simplicity and readability.

We formalize the process of generating expressions and constructing constraints independently for every participating qualifier placeholder. To structure this formally, we define $\mathsf{GEN}^s_{\mathbb{C}}(e : \mathcal{Q}_e)$ as the generative procedure for expression e in scope s under constraints related to a newly defined $\mathcal{Q}_e$ in the constraint set $\mathbb{C}$ while pushing the initialization of $\tau_{\mathcal{Q}_e}$ to $\mathbb{C}$, and the fraction line as the operator for breaking down a generation process.

$$\frac{\mathsf{GEN}^s_{\mathbb{C}}(e)}{\mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\textit{constraints for } \mathcal{Q}_{e_1})}(e_1 : \mathcal{Q}_{e_1}) \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\textit{constraints for } \mathcal{Q}_{e_2})}(e_2 : \mathcal{Q}_{e_2}) \quad \ldots} \tag{1}$$

Specifically, Rule 1 expresses that generating e involves decomposing it into the generation of all its constituent subexpressions $e_1, e_2, \ldots$. In the demonstrated formula, $\mathbb{C}$.push serves to incrementally update the global constraint set $\mathbb{C}$. Subexpressions are generated sequentially, ensuring constraints derived for e_1 become visible to the subsequent generator of e_2, as shown in the example.

$$\frac{\mathsf{GEN}^s_{\mathbb{C}}(v = e : \mathcal{T})}{\mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{T} \doteq \mathcal{T}_v)}(v : \mathcal{T}_v) \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{T}_e \lessdot \mathcal{T})}(e : \mathcal{T}_e)} \ \boxed{\mathcal{T}\text{-Assign}} \tag{2}$$

$$\frac{\mathsf{GEN}^s_{\mathbb{C}}(e_1 \mathbin{?} e_2 : e_3 : \mathcal{S})}{\mathsf{GEN}^s_{\mathbb{C}}(e_1 : \mathcal{S}_{e_1}) \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{S}_{e_2} \lessdot \mathcal{S})}(e_2 : \mathcal{S}_{e_2}) \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{S}_{e_3} \lessdot \mathcal{S})}(e_3 : \mathcal{S}_{e_3})} \ \boxed{\mathcal{S}\text{-Cond}} \tag{3}$$

Example 3.1. Rule 2 describes an example of generating assignments while maintaining data-type-related constraints. The expression $v = e$ is qualified by data type placeholder $\mathcal{T}$. Specifically, Erwin assigns a new data type placeholder $\mathcal{T}_v$ to ready-to-generate v, pushes the related constraint $\mathcal{T} \doteq \mathcal{T}_v$ to the constraint set $\mathbb{C}$, and generates v under the updated $\mathbb{C}$. Then Erwin generates e according to the same principle. Rule 3 delineates how to generate conditional expressions while ensuring storage-location constraints are preserved. Specifically, Erwin generates the boolean expression (e_1) first and then the two branches (e_2 and e_3) subsequently. Following Solidity rules, the storage locations of e_2 and e_3 are more restricted than that of the conditional expression.

Context Update. The top-down approach to expression generation concludes by producing atomic expressions, such as identifier expressions and literal expressions. For literal expressions, the qualifier placeholder restricts the range of possible values. For example, if the data type placeholder excludes the assignment of string, then the literal cannot be a string literal. Identifier expressions, which directly reference variable declarations, carry greater complexity.

$$\frac{}{(v, \mathcal{S}_v) \in \Gamma.\mathsf{query}(s) \quad \mathbb{C} \cup \{\mathcal{S} \lessdot \mathcal{S}_v\} \text{ is solvable} \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{S} \lessdot \mathcal{S}_v)}(v : \mathcal{S})} \ \boxed{\mathcal{S}\text{-Ident}} \tag{4}$$

$$\frac{}{(v, \mathcal{T}_v) \in \Gamma.\mathsf{query}(s) \quad \mathbb{C} \cup \{\mathcal{T} \doteq \mathcal{T}_v\} \text{ is solvable} \quad \mathsf{GEN}^s_{\mathbb{C}.\mathsf{push}(\mathcal{T} \doteq \mathcal{T}_v)}(v : \mathcal{T})} \ \boxed{\mathcal{T}\text{-Ident}} \tag{5}$$

Rule 2 and 4 are constraints related to the generation of identifier expression. They both include a Γ.*query* function to query the existence of variables together with their qualifier placeholders from the scope s. If there is no variable whose qualifier placeholder satisfies the solvability requirement, Erwin should update context by adding new declarations.

Algorithm 1 Algorithm for querying, checking and updating

```
Require: The context $\Gamma$, the constraint set $\mathbb{C}$, the current scope $s$, the qualifier place-
    holder $\mathcal{Q}$ of the identifier.
Function QCU($\Gamma$, $\mathbb{C}$, $s$, $\mathcal{Q}$)
 1: $D$ is the set of declarations collected from $\Gamma$.query($s$).
 2: for $(d, \mathcal{Q}_d) \in \Gamma.\mathsf{query}(s)$ do
 3:     if $\tau_{\mathcal{Q}_d} \cap \tau_{\mathcal{Q}} = \emptyset$ then
 4:         Remove $d$ from $D$
 5:     else
 6:         $\mathbb{C}' \leftarrow \mathbb{C} \bigcup \{\tau_{\mathcal{Q}_d} \leftarrow \tau_{\mathcal{Q}_d} \cap \tau_{\mathcal{Q}}\}$
 7:         if $\mathbb{C}'$ is not solvable then
 8:             Remove $d$ from $D$
 9:         end if
10:     end if
11: end for
12: if $D = \emptyset$ then
13:     Pick $s_d$ randomly from $\Gamma$.findVisibleScopes($s$)
14:     $\Gamma.\mathsf{push}(d, s_d, \mathcal{Q}_d)$
15: end if
End Function
```

Algorithm 1 demonstrates this process. The algorithm is divided into two primary steps: (1) checking for existing available declarations (lines 2-11), and (2) modifying the context by adding new declarations when none are available (lines 12-15). In the first step, Erwin excludes those whose qualifier codomain does not overlap with the qualifier placeholder codomain of the identifier (lines 3-5). Finally, if a declaration is selected as the origin of the identifier, Erwin attempts to modify the constraint set accordingly (line 6). However, if the updated constraint set becomes unsolvable, the update is reverted, and the declaration is excluded (lines 7-9). The process of determining solvability involves enumerating the possible values for each qualifier placeholder from its codomain and verifying whether there are assignments of qualifiers to each qualifier placeholder that fulfill all the given relations. In the second step, if no suitable declaration is found (line 12), Erwin randomly picks a scope from the set of scopes that are visible to s (line 13), and pushes the newly generated declaration d associated with the qualifier placeholder $\mathcal{Q}_d$ into the scope (line 14).

Visibilities and mutabilities are not restrained by relations. Constraints for them are about narrowing down the value codomain. For instance, if Erwin detects a function f calls another function g whose scope is not visible to the scope of f, then Erwin performs $\mathbb{C}$.push($\tau_{\mathscr{V}_g} \leftarrow \tau_{\mathscr{V}_g} \cap \{\text{external}, \text{public}\}$), ensuring g can only be qualified by external or public.

Example 3.2. Figure 6 demonstrates a step-by-step example of generating the first expression immediately following the context initialization, which generates no variable declaration in this case. The dice icon signifies randomness. For example, in step 1 during the generation of e_1, Erwin randomly sets it to $e_3 \mathrel{+}= e_4$. The formulas within brackets denote the constraints built while generating sub-expressions. In steps 2 and 3, Erwin updates Γ with new declarations of x_1 and x_2 because it cannot find suitable variable declarations. Sub-expression generation ends when a literal or an identifier is generated. The overall generation process is depth-first and halts at step 8, yielding a global constraint set comprising five constraints. The generation of expressions and the building of constraints come to an end either when Erwin randomly chooses to stop or when the maximum number of expressions is reached.

$\mathbb{C}$-Solvability. The constraint set $\mathbb{C}$ is inherently solvable due to its construction process. During the creation of the constraint set, qualifier codomain restrictions are verified before being applied to ensure that they do not involve an empty codomain. For relations ($\doteq, \lessdot$), Erwin ensures that these constraints do not result in unsolvability by verifying their solvability during the generation of identifier expressions (e.g., $\mathscr{T}$-Ident, $\mathscr{S}$-Ident).

Statement Generation. Erwin categorizes all statements into two types: (1) expression statements and (2) non-expression statements. For expression statements, Erwin simply appends a semicolon to a generated expression, allowing the compiler to treat the expression as a valid statement. For non-expression statements, Erwin either directly generates the statement within an appropriate scope (e.g., placing a `break` statement inside an `if` scope) or creates a new scope, inserts the statement into it, and populates the new scope with additional expressions and statements (e.g., generating `while` loops within a function body, filling the loop body with statements, and setting the loop condition using expressions).

3.3 Lowering Program Templates to Test Programs

The template defines a subspace of the unbounded search space. The subspace contains all combinations of qualifiers, from which Erwin needs to find valid ones. An intuitive way to accomplish this task is to systematically consider every possible substitution, where each substitution consists of assigning a qualifier to each placeholder in the constraint set $\mathbb{C}$, and then select those substitutions that satisfy all constraints in $\mathbb{C}$.

However, the large volume of constraints can negatively impact efficiency. Figure 7a visualizes all 12 introduced data-type constraints among 11 qualifier placeholders during the generation of a single expression $(x_1 \mathrel{+}= x_2) + (x_1 \times (1 - x_2))$. In the figure, each edge denotes a constraint from a qualifier placeholder associated with a declaration in the block to another qualifier placeholder. In addition to these, each expression's generation may also introduce constraints related to storage locations and codomain restrictions. Generating all substitutions that meet these constraints can be computationally intensive because each potential substitution needs to be verified against every constraint. Additionally, due to the considerable number of qualifier placeholders, the overall pool of substitution candidates is large, which makes identifying valid substitutions more time-consuming.

We observe that a significant portion of the constraints consist of relations between newly introduced qualifier placeholders for expressions (*e.g.*, the $\doteq$ constraint from the qualifier placeholder of $x_1 \mathrel{+}= x_2$ to that of x_1 in Figure 7a), which are generated during expression construction. These expression-level qualifier placeholders do not appear on program templates, but serve to establish indirect constraint relations with the qualifier placeholders associated with declarations, which are the placeholders required to be instantiated. Since our focus is solely on substituting declaration-level qualifier placeholders during program template lowering, we can map the constraint relationships of expression-level qualifier placeholders to the declaration-level counterparts. This enables the removal of expression-level qualifier placeholders and their related constraints from the constraint set, thus boosting efficiency.

To facilitate this transfer process, we begin by examining the relationships between expression-level qualifier placeholders and the declaration-level qualifier placeholders to which they can be connected directly and indirectly. Subsequently, we deduce the constraints between each pair of declarations by identifying the most restrictive relational constraint imposed by their shared expression-level qualifier placeholders.

The initial step involves establishing constraints from expression-level qualifier placeholders to those they are indirectly connected to. Specifically, if $\mathcal{Q}_{e_i} \bowtie_1 \mathcal{Q}_{e_j}$ and $\mathcal{Q}_{e_j} \bowtie_2 \mathcal{Q}_d$, this step introduces a constraint between $\mathcal{Q}_{e_i}$ and $\mathcal{Q}_d$. To achieve this, we define four constraint propagation rules, as shown in Figure 8, where $\mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j)$ is semantically equivalent to the infix notation $\mathcal{Q}_i \bowtie \mathcal{Q}_j$), but the prefix form is used to improve the readability and clarity of complex formulas. These rules introduce a new relation $\gtrless$, meaning that a qualifier placeholder is either more restricted $\lessdot$ or less restricted $\gtrdot$ than another placeholder. Equation (4) illustrates how this new relation arises: when both $\mathcal{Q}_i$ and $\mathcal{Q}_k$ are found to be more restricted than $\mathcal{Q}_j$, the resulting relation between $\mathcal{Q}_i$ and $\mathcal{Q}_k$ is given by $\gtrless$.

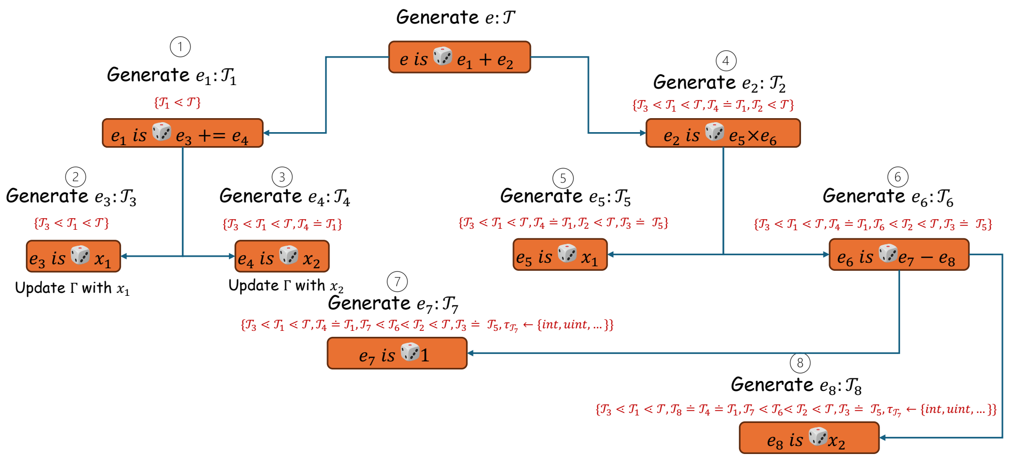
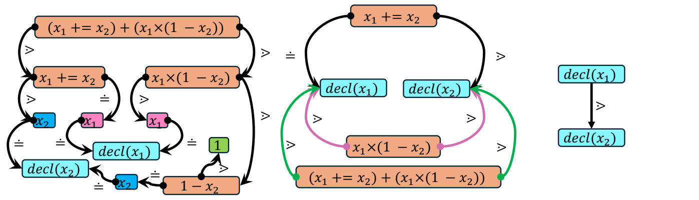

Generate $e : \mathcal{T}$
e is $e_1 + e_2$
1 Generate $e_1 : \mathcal{T}_1$
$\{\mathcal{T}_1 \lessdot \mathcal{T}\}$
e_1 is $e_3 \mathrel{+}= e_4$
2 Generate $e_3 : \mathcal{T}_3$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}\}$
e_3 is x_1
Update Γ with x_1
3 Generate $e_4 : \mathcal{T}_4$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_4 \doteq \mathcal{T}_1\}$
e_4 is x_2
Update Γ with x_2
4 Generate $e_2 : \mathcal{T}_2$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_4 \doteq \mathcal{T}_1, \mathcal{T}_2 \lessdot \mathcal{T}\}$
e_2 is $e_5 \times e_6$
5 Generate $e_5 : \mathcal{T}_5$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_4 \doteq \mathcal{T}_1, \mathcal{T}_2 \lessdot \mathcal{T}, \mathcal{T}_3 \doteq \mathcal{T}_5\}$
e_5 is x_1
6 Generate $e_6 : \mathcal{T}_6$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_4 \doteq \mathcal{T}_1, \mathcal{T}_6 \lessdot \mathcal{T}_2 \lessdot \mathcal{T}, \mathcal{T}_3 \doteq \mathcal{T}_5\}$
e_6 is $e_7 - e_8$
7 Generate $e_7 : \mathcal{T}_7$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_4 \doteq \mathcal{T}_1, \mathcal{T}_7 \lessdot \mathcal{T}_6 \lessdot \mathcal{T}_2 \lessdot \mathcal{T}, \mathcal{T}_3 \doteq \mathcal{T}_5, \tau_{\mathcal{T}_7} \leftarrow \{int, uint, ...\}\}$
e_7 is 1
8 Generate $e_8 : \mathcal{T}_8$
$\{\mathcal{T}_3 \lessdot \mathcal{T}_1 \lessdot \mathcal{T}, \mathcal{T}_8 \doteq \mathcal{T}_4 \doteq \mathcal{T}_1, \mathcal{T}_7 \lessdot \mathcal{T}_6 \lessdot \mathcal{T}_2 \lessdot \mathcal{T}, \mathcal{T}_3 \doteq \mathcal{T}_5, \tau_{\mathcal{T}_7} \leftarrow \{int, uint, ...\}\}$
e_8 is x_2

Figure 6: A step-by-step example of generating the first expression $(x_1 \mathrel{+}= x_2) + (x_1 \times (1 - x_2))$



(a) Data Type Constraints of $(x_1 \mathrel{+}= x_2) + (x_1 \times (1 - x_2))$ **(b) Data type constraints after constraint propagation (c) Data type constraints after constraint inference**

Figure 7: Constraint reduction for $(x_1 \mathrel{+}= x_2) + (x_1 \times (1 - x_2))$

$$
\begin{aligned}
&(1)\quad (\mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k)) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_k, \mathcal{Q}_l) = \\
&\qquad \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j) \land (\mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_k, \mathcal{Q}_l)) \quad \text{(Associative)} \\
&(2)\quad \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \implies \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j) \quad \text{(Commutative)} \\
&(3)\quad \text{for } \bowtie \text{ in } \{\lessdot, \gtrdot, \doteq, \rtimes\}: \\
&\qquad \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \implies \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_k) \quad \text{(Transitive)} \\
&\qquad \mathcal{P}_{\doteq}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \implies \mathcal{P}_{\bowtie}(\mathcal{Q}_i, \mathcal{Q}_k) \\
&\qquad \mathcal{P}_{\rtimes}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_j, \mathcal{Q}_k) \implies \mathcal{P}_{\rtimes}(\mathcal{Q}_i, \mathcal{Q}_k) \\
&(4)\quad \mathcal{P}_{\lessdot}(\mathcal{Q}_i, \mathcal{Q}_j) \land \mathcal{P}_{\gtrdot}(\mathcal{Q}_j, \mathcal{Q}_k) \implies \mathcal{P}_{\rtimes}(\mathcal{Q}_i, \mathcal{Q}_k)
\end{aligned}
$$

Figure 8: Constraint propagation rules

Following constraint propagation rules, we establish relational constraints linking each expression-level qualifier placeholder ($\mathcal{Q}_e$) to the declaration-level qualifier placeholders it can reach. As a result, for every pair of declaration-level qualifier placeholders ($\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$), we can identify all their shared expression-level qualifier placeholders ($\mathfrak{Q}_e$), as well as all relational constraints that connect $\mathfrak{Q}_e$ to the $\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$, as illustrated by Figure 7b.

Subsequently, we infer the relation between all pairs of $\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$. The inference rules are demonstrated in Figure 9. Taking equation (1) as an example, it demonstrates that if the relations between $\mathcal{Q}_e$ and $\mathcal{Q}_{d_i}$ or the relation between $\mathcal{Q}_e$ and $\mathcal{Q}_{d_j}$ is not deterministic, then so is the relation between $\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$. After applying constraint inference rules, the relation between $\mathcal{Q}_{d_i}$ and

$$
\begin{aligned}
&(1)\quad \mathcal{P}_{\rtimes}(\mathcal{Q}_e, \mathcal{Q}_{d_i}) \lor \mathcal{P}_{\rtimes}(\mathcal{Q}_e, \mathcal{Q}_{d_j}) \implies \mathcal{P}_{\rtimes}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j}) \\
&(2)\quad \mathcal{P}_{\bowtie}(\mathcal{Q}_e, \mathcal{Q}_{d_i}) \land \mathcal{P}_{\bowtie}(\mathcal{Q}_e, \mathcal{Q}_{d_j}) \implies \text{if } \bowtie \text{ is } \doteq \text{ then } \mathcal{P}_{\doteq}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j}) \text{ else } \mathcal{P}_{\rtimes}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j}) \\
&(3)\quad \text{if } \bowtie_i \text{ is not } \rtimes \text{ and } \bowtie_j \text{ is not } \rtimes\text{, then:} \\
&\qquad \mathcal{P}_{\bowtie_i}(\mathcal{Q}_e, \mathcal{Q}_{d_i}) \land \mathcal{P}_{\bowtie_j}(\mathcal{Q}_e, \mathcal{Q}_{d_j}) \land \underline{\bowtie_i \prec \bowtie_j} \implies \mathcal{P}_{\gtrdot}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j}) \\
&\qquad \mathcal{P}_{\bowtie_i}(\mathcal{Q}_e, \mathcal{Q}_{d_i}) \land \mathcal{P}_{\bowtie_j}(\mathcal{Q}_e, \mathcal{Q}_{d_j}) \land \underline{\bowtie_j \prec \bowtie_i} \implies \mathcal{P}_{\lessdot}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j})
\end{aligned}
$$

Figure 9: Constraint inference rules, where $\prec$ is defined by $\lessdot \ \prec \doteq \prec \ \gtrdot$

$\mathcal{Q}_{d_j}$ may be more than one. To finalize the constraint relation, we select the most restrictive relation. Formally, this corresponds to choosing the minimal relation with respect to the partial order $\prec$ as defined in Figure 9. Figure 7c illustrates the final constraint reduced from Figure 7b via applying constraint inference rules, which is much simpler than the original one.

Algorithm 2 Algorithm for reducing constraints

```
Function CONSTRAINT_REDUCTION($\mathbb{C}$)
 1: $\mathfrak{Q}_e \leftarrow$ Get expression-level qualifier placeholders from $\mathbb{C}$
 2: for $\mathcal{Q}_e \in \mathfrak{Q}_e$ do
 3:     $\mathfrak{Q}_d \leftarrow$ Get declaration-level qualifier placeholders restrained by $\mathcal{Q}_e$
 4:     for $(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j}) \in \mathfrak{Q}_d \times \mathfrak{Q}_d$ where $i < j$ do
 5:         Calculate $\mathcal{P}_{\bowtie_i}(\mathcal{Q}_e, \mathcal{Q}_{d_i})$ and $\mathcal{P}_{\bowtie_j}(\mathcal{Q}_e, \mathcal{Q}_{d_j})$ by constraint propagation
            rules (figure 8)
 6:         Infer $\mathcal{P}_{\bowtie_{ij'}}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j})$ by constraint inference rules (figure 9)
 7:         if $\mathcal{P}_{\bowtie_{ij}}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j})$ does not exist or $\bowtie_{ij'} \prec \bowtie_{ij}$ then
 8:             $\bowtie_{ij} = \bowtie_{ij'}$, $\mathcal{P}_{\bowtie_{ij}}(\mathcal{Q}_{d_i}, \mathcal{Q}_{d_j})$
 9:         end if
10:     end for
11: end for
End Function
```

Algorithm 2 summarizes and formalizes the constraint reduction process. The algorithm iterates through each expression-level qualifier placeholder $\mathcal{Q}_e$ (line 2) and then examines all pairs of declaration-level qualifier placeholders (*e.g.,* $\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$) linked by the expression node (line 4). For each such pair, the algorithm initially determines the relation between $\mathcal{Q}_e$ and $\mathcal{Q}_{d_i}$ as well as between $\mathcal{Q}_e$ and $\mathcal{Q}_{d_j}$ following the constraint propagation rules (line 5). Then it infers the relation between $\mathcal{Q}_{d_i}$ and $\mathcal{Q}_{d_j}$ based on constraint inference rules (line 6). If there is no relation between $\mathbb{Q}_{d_i}$ and $\mathbb{Q}_{d_j}$ or if the newly inferred constraint is stricter than the existing one, then update the relation to use the freshly inferred constraint (lines 7-9).

Applying Algorithm 2 significantly reduces the size of the constraint set. The final phase involves enumerating all possible substitutions and checking each one against the remaining constraints. Given that $\mathbb{C}$ is solvable, there will always be at least one substitution that satisfies all constraints. Once such substitutions are identified, they are applied to the program template, which is then instantiated as concrete test programs.

4 Experimental Setup

Our experimental setup is designed to answer the following research questions:

RQ1 How effective and efficient is Erwin in generating bug-triggering test programs?

RQ2 How does Erwin compare to the state-of-the-art bug detection tools for Solidity compilers?

RQ3 Do the main components of Erwin enhance bug detection in Solidity compilers?

4.1 Baselines

We select two state-of-the-art bug detection tools for Solidity compilers as baselines. To the best of our knowledge, they are the only publicly available tools described in academic papers that focus on fuzzing Solidity compilers. We also develop SoliditySmith, a variant of Erwin that deliberately avoids bounded exhaustiveness, and include it as another baseline.

AFL-compiler-fuzzer (ACF) [21] is an AFL-based fuzzer enhanced with language-agnostic mutation rules (e.g., modifying conditions, deleting statements) and code fragment assembly to diversify test cases and reuse bug-revealing segments.

`Fuzzol` [30] is a mutation-based fuzzer tailored for Solidity that applies language-specific mutations by analyzing Solidity ASTs and mutating nodes or opcode arguments, enabling more precise and effective fuzzing.

SoliditySmith is a variant of Erwin that excludes the bounded exhaustiveness strategy by stopping constraint solving upon finding the first valid substitution. Because this step incurs non-negligible runtime (shown in Table 2), we remove its cost from all measurements. In other words, SoliditySmith corresponds to Erwin running in gen1 mode (see Table 2) with the constraint-solving time excluded.

4.2 Systems Under Test

`solc` [9] is the primary Solidity compiler that converts Solidity code into EVM bytecode for deployment on Ethereum, ensuring syntax correctness, error checking, and optimization.

`solang` [8] is a multi-platform Solidity compiler supporting Ethereum, Solana, and Polkadot, offering greater flexibility and focusing on performance and modern development compared to `solc`.

`slither` [5] is a static analysis tool for Solidity that detects vulnerabilities by analyzing code structure and flow without execution.

4.3 Metrics

Our evaluation employs the following widely-used metrics:

Code Coverage We trace source-level line and edge coverage of the entire `solc` codebase, aligning with prior research [12, 20, 25]. `solang` is excluded as its differing grammar would hinder baseline test program mutation, leading to an unfair comparison with Erwin.

Bug Count Following previous work [28, 33, 35, 36], we measure the number of bugs detected by Erwin to assess its effectiveness. Potential bugs are identified by unexpected crashes (e.g., segmentation faults), incorrect behavior (e.g., assertion failures without clear error messages), or hangs, and further validated by developer confirmation and code patches. For bugs detected within a Solidity compiler bug dataset [29], we confirm a unique bug match if the detected error message is identical or negligibly different from the documented one.

4.4 Implementation

Our Erwin comprises 15,098 lines of TypeScript code. While Erwin is designed to be easily extensible to cover the full range of Solidity language features, the current version (1.3.1) only supports a limited subset of these features, including contracts, functions, modifiers, events, errors, etc.

Given the differences in the Solidity grammar across various versions and compilers, Erwin was developed using the Solidity grammar from `solc` version 0.8.20. This grammar is not universally applicable but is compatible with a broad range of Solidity versions. Furthermore, we have extended Erwin to support the grammar of `solang` version 0.3.3, enhancing its compatibility across different Solidity compiler ecosystems.

4.5 Experimental Configuration

We performed bug detection across multiple versions, including `solc` 0.8.20–0.8.28, `solang` 0.3.3, and `slither` 0.10.4. Given that Erwin's generation process incorporates randomness, we expose all random variables as configurable flags. For the evaluation, we integrated a random flag selection script into Erwin to avoid bias toward specific random seeds and guarantee the fairness of the assessment. This script is capable of exploring all feasible and valid combinations of random flags, delivering a comprehensive evaluation of Erwin's performance. It is worth noting that SoliditySmith also employs this same script during generation when compared with Erwin.

5 Evaluation

5.1 RQ1: Effectiveness of Erwin

Table 1: Reported bugs

	Bugs	Confirmed	Fixed	Duplicate
solc	21	15	8	3
solang	4	0	0	0
slither	1	1	1	0
Total	26	16	9	3

5.1.1 Case Study of Detected Bugs. After six months of intermittent operation, Erwin has identified a total of 26 bugs across `solc`, solang, and slither. Table 1 provides a detailed breakdown of the bugs detected by Erwin in each system under test. Out of these, 18 have been confirmed as bugs by developers, and 10 of those have already been fixed. The remaining bugs, including six bugs for solc and four bugs for solang, are still under investigation. Among the 18 confirmed bugs, three solc bugs are categorized as duplicates by developers. The 13 non-duplicate confirmed bugs have various symptoms and root causes. In terms of symptoms, two bugs trigger segmentation faults, four bugs lead to incorrect output, one bug results in a hang, five bugs induce internal compiler errors (ICE), and one bug causes the compiler to accept invalid programs. As for root causes, two bugs are caused by type errors, three are related to incorrect error handling, four are due to formal verification errors, one is caused by a code analysis error, one is due to incorrect version control, and two are caused by specification errors.

```
bool internal v;
modifier m() {
  while (v ? false : v) {}
  _;
}
```

Example 1: Test program for GitHub issue #2619 [1]

Code Analysis Error + Hang. The GitHub issue 2619 [1] is a code analysis error that causes slither to hang. Listing 1 shows the bug-triggering fragment inside a contract for this issue. The bug arises from the code analysis process, where slither statically analyzes the code to detect vulnerabilities. The bug-triggering scenario involves a modifier that contains a while loop, where the loop's condition expression is a ternary operation over an uninitialized boolean member variable. If the while loop is moved into a function or the ternary operation expression is altered, the bug will not be detected during analysis.

```
contract C {
  constructor() {
    int128 v;
    (v *= v);
  }
  bool[6] internal arr;
  function f() internal {
    while(arr[(3)] == true) {}
  }
}
```

Example 2: Test program for GitHub issue #15647 [3]

Formal Verification Error + ICE. GitHub issue 15647 causes an Internal Compiler Error (ICE) in solc due to a formal verification error during SMT encoding, specifically when mishandling array access expressions (Listing 2). This formal verification process, which converts contract code to logical formulas for SMT solver analysis [29], fails under precise conditions: a boolean array access within a while/do-while loop condition inside an internal function, combined with a constructor containing a multiplication assignment. Although the root cause is straightforward, these stringent requirements make the bug exceptionally difficult to detect. Erwin identifies it by generating complex test programs that explore diverse data types and function visibilities, enabling comprehensive coverage of the required qualifier combinations.

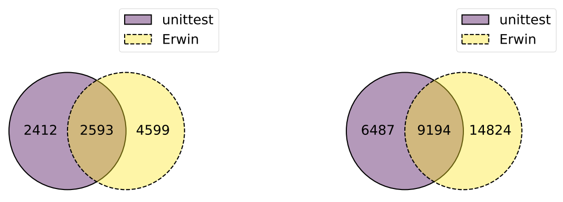
```
contract C {
  struct S {
    bool b;
  }
}
contract D {
  C.S public s;

  function f() public view {
    true ? C.S(true) : this.s();
  }
}
```

Example 3: Test program for GitHub issue #15525 [11]

Specification Error + ICE. GitHub issue 15525 is a specification error causing the solc compiler to crash, as demonstrated by the test program in Listing 3. The bug arises from the compiler's incorrect handling of a getter for a struct state variable: the call C.S(true) creates a new struct instance, but this.s() mistakenly returns a tuple of the struct members instead of the struct itself. This type mismatch triggers a compilation error. Developers acknowledge this long-standing quirk and plan to fix it in a future breaking change, noting that the current Solidity specification is unclear. By formalizing the specification, Erwin can systematically explore language features and their interactions, enabling it to detect such bugs.

5.1.2 Coverage Enhancement. As the most popular Solidity compiler and the official compiler for Solidity language, solc has been widely adopted by smart contract developers. To evaluate the effectiveness of Erwin in generating high-quality bug-triggering test programs, we compare the line and edge coverage of solc using test programs generated by Erwin within 24 hours against the unit test cases of solc.



(a) Edge coverage **(b) Line coverage**

Figure 10: Edge and line coverage difference between Erwin-generated test programs vs. unit tests

Figure 10a and 10b show the Venn diagrams of the edge and line coverage differences among Erwin's generated test programs and unit tests. The results show that Erwin can cover 4,599 edges and 14,824 lines that are not covered by unit test cases.

5.1.3 Throughput and Efficiency. To evaluate Erwin's efficiency in generating test programs, we measure its throughput for producing program templates and corresponding test programs by running

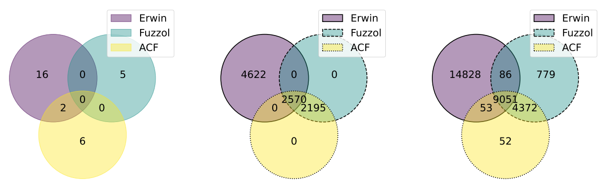
Table 2: Througput of program generation

gen	1	50	100	150	200	250	300
Program Templates / s	65.53	13.78	8.53	5.84	2.13	1.98	1.65
Programs / s	65.53	689.00	853.53	876.17	427.62	496.13	494.76

1000 iterations per generation setting with the `-max` flag enabled. Each experiment is repeated five times, and the median throughput is reported (Table 2). The gen row shows the number of test programs derived from program templates. To ensure sufficient complexity, Erwin is configured to generate two contracts with two member functions each, expanding the search space. In the `gen1` scenario, Erwin stops collecting substitutions after finding the first valid combination, achieving a throughput of 65.53 test programs per second, demonstrating fast program template generation and constraint solving. As the suffix number in gen increases, program template generation throughput decreases due to the extra time needed for substitution collection. However, test program throughput rises from `gen1` to `gen150` before fluctuating between 420 and 560 programs per second, indicating that collecting more substitutions slows down the search.

5.2 RQ2: Comparison with State-of-the-Art Fuzzing Tools

We evaluate Erwin against two cutting-edge fuzzing tools for Solidity compilers, `ACF` and `Fuzzol`, by comparing the number of bugs detected in the Solidity compiler bugs dataset. This dataset includes 104 reproducible bugs that manifest as crashes in the Solidity compiler (`solc`).



(a) Bug detection (b) Edge coverage (c) Line coverage

Figure 11: Effectiveness differences among Erwin, ACF, and Fuzzol

Figure 11a compares the bug detection performance of Erwin, `ACF`, and `Fuzzol` on the Solidity compiler bugs dataset after 20 days of separate execution. The seed pools for `ACF` and `Fuzzol` follow their default settings [21, 30]. Erwin detects 18 bugs in total, 16 of which are missed by `ACF` and `Fuzzol`. Additionally, Erwin finds a segmentation fault in `solc` 0.7.5 not present in the dataset and undetected by the other fuzzers. Among the 13 unique bugs, nine stem from formal verification errors, two from code generation errors, one from a type system error, and one from a memory-related error. Formal verification errors, relying on SMT encoding and sensitive to qualifier values, are notably difficult for `ACF` and `Fuzzol` to detect [29]. Erwin's strength lies in generating complex test programs with intricate control and data flows, exploring diverse qualifier combinations to uncover such errors. For example, bug 8963 causes an ICE in `solc` due to a formal verification error involving a tuple or mapping type assignment within an if condition (Listing 4). Erwin successfully generates the necessary constructs to expose this bug. The dataset's 104 bugs require various features, *e.g.*, `byte` type (12

```
pragma experimental SMTChecker;
contract C {
  function f(int128 v) public{
    if (1 == 1) {
      (v) >>= 14235;
    }
  }
}
```

Example 4: Test program for GitHub issue #8963 [6]

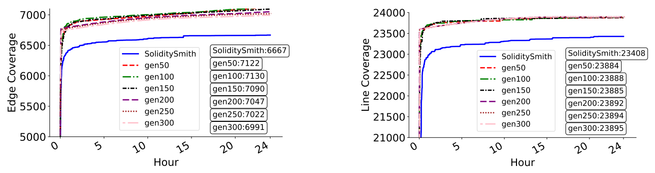
bugs), contract inheritance (10), `library` (8), array `pop`/`push` (6), `fixed` type, `abi.encode`, and grammar violations (5 each), function types and inherent function calls (4 each), inline assembly and type alias (3 each), `try catch`, multiple test files, and enum (2 each), and comments (1). Erwin currently lacks support for these features, explaining why it misses 72 bugs and fails to detect 11 bugs found by `ACF` and `Fuzzol`. In summary, Erwin detects 18 of 32 bugs not dependent on unsupported features, indicating significant potential for more discoveries with extended testing and feature expansion. Beyond bug detection, Figure 11b and 11c present Venn diagrams of edge and line coverage differences after 24 hours. Erwin covers 4,622 edges and 14,828 lines not covered by `ACF` and `Fuzzol`.

5.3 RQ3: Ablation Study

To evaluate the effectiveness of the main components of Erwin in enhancing bug detection in Solidity compilers, we conduct an ablation study by comparing the bug detection performance of Erwin with and without the main components.

5.3.1 *Impact of the Bounded Exhaustive Generation Strategy.*

To evaluate the bounded exhaustive random program generation strategy, we compare Erwin with SoliditySmith, a variant of Erwin that stops constraint solving upon finding the first valid substitution to exclude the bounded exhaustiveness strategy. For fair comparison, we exclude time spent on constraint reduction, measuring only program template generation time, which aligns with direct test program generation. This isolates the effect of program template generation and lowering on SoliditySmith's performance. Because the process is stochastic, we repeat experiments five times and report median coverage values at each timestamp after linear interpolation to ensure result reliability.



(a) Edge coverage (b) Line coverage

Figure 12: Edge and line coverage over time, where percentages refer to coverage ratios

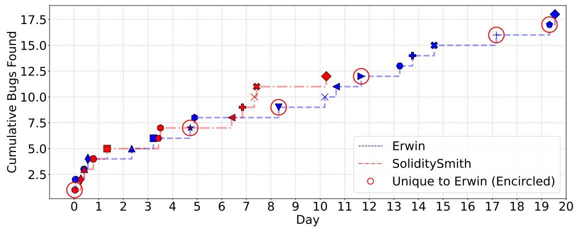


Figure 13: Effectiveness of Erwin vs. SoliditySmith in bug detection on the Solidity compiler bugs dataset

Figure 12a and 12b illustrate the progression of edge and line coverage achieved by Erwin across various configurations, as well as by SoliditySmith. In this diagram, `gen50` indicates that Erwin produces up to 50 test programs by examining qualifier combinations from the program template. This same interpretation extends to other `genXX` parameters. These three configurations establish the upper limit for the exhaustiveness and exhibit different preferences in balancing the time spent exploiting the generated program template against the time spent creating new ones. The number shown on the figures demonstrates the exact coverage achieved by each subject. The figures evidently show that Erwin outperforms SoliditySmith in both edge and line coverage across all selected configurations. On average, Erwin covers about 400 more edges and 480 more lines than SoliditySmith.

To further analyze the impact of coverage differences between Erwin and SoliditySmith, we extend the experiment in §5.2 by running SoliditySmith for 20 days on the Solidity compiler bugs dataset and comparing their bug detection performance. The results in Figure 13 use different shapes to denote bugs found by each tool. Overall, SoliditySmith detects 12 bugs, all of which are also found by Erwin. For example, bug 10105 requires (1) a struct type

```
pragma experimental ABIEncoderV2;
contract C {
  struct S { int[][5] c; }
  S s;
  function f(S calldata c) external {
    s = c;
  }
}
```

Example 5: Test program for GitHub issue #10105 [7]

containing a nested array and (2) a non-internal function with a calldata parameter of that struct type (Listing 5). These strict data type, visibility, and storage location conditions are challenging to satisfy. Although SoliditySmith may generate similar tests differing only in the struct's storage location, it fails to trigger the bug. In contrast, Erwin thoroughly explores all relevant qualifier combinations and interactions, successfully uncovering this bug. Another example is the bug in Figure 2 (§2.1), detected by Erwin on day five but missed by SoliditySmith.

It is worth noting that SoliditySmith failed to detect any new bugs after the 11th day, while Erwin successfully identified eight additional bugs during this period. Of these eight bugs, five were also detected by SoliditySmith. This lag in bug detection stems from the fact that Erwin allocates part of its resources to exploring the qualifiers.

Table 3: Search space size reduction by constraint reduction

	Median	Min	Max
Type	1.1e14	8	6.9e68
Storage	4.5e6	1	1.6e38
Visibility	1	1	1
Mutability	1	1	1

5.3.2 *Impact of the Constraint Reduction.*

Constraint reduction, detailed in Algorithm 2, is a key part of Erwin that reduces the search space and improves the efficiency of bug detection. We evaluate its impact by running Erwin for 1000 iterations and comparing the average size of the search space before and after pruning, as summarized in Table 3. The effectiveness of pruning largely depends on the average number of possible values each expression node can take. Since data types have a larger set of possible values than other qualifiers (see Figure 5), pruning reduces their search space more significantly. However, pruning does not affect visibility and mutability qualifiers because their constraints are not represented in the graph.

6 Discussion

6.1 Threats to Validity

To mitigate potential bias from randomness in bounded exhaustive program generation, we employ two strategies. First, we employ repeated experiments with median reporting, which is applied to `solc` coverage enhancement (Figure 10), throughput analysis (Table 2), Erwin's coverage under various settings (Figure 12), and baseline coverage comparisons (Figure 11b, 11c). Each test is repeated five times, with median values reported. Second, we use extended testing duration, which is applied to Erwin vs. SoliditySmith comparisons (Figure 13) and Erwin vs. baseline bug detection assessments (Figure 11a), with experiments running for 20 days. These approaches minimize randomness effects and enhance result reliability.

6.2 False Alarms

Although Erwin is based on the Solidity language specification and its constraints, some generated programs may still be invalid and rejected by compilers, resulting in false alarms. These can stem from implementation errors in Erwin or, more often, from ambiguities or mistakes in the specification itself. So far, Erwin has identified three suspicious false alarms, with two confirmed as specification errors. For example, GitHub issue 15483 [10] involves a function return declaration in calldata. While `solc` requires that calldata return declarations be assigned before use, this rule is not stated in the specification. Although the `solc` team considers this behavior intentional, they acknowledge the inconsistency, which has now been addressed. These findings demonstrate that Erwin can not only detect compiler bugs but also reveal deficiencies in the Solidity specification, helping to improve its accuracy and alignment with compiler behavior.

6.3 Seed Pool Enhancement

Program generators serve dual roles: they create test programs to find bugs and enrich seed pools for mutation-based fuzzers. For instance, Csmith [37] is commonly used to generate seeds for fuzzers like Athena [23], Hermes [34], and GrayC [20], which test compilers such as GCC and LLVM. These fuzzers mutate Csmith's seeds to reveal bugs that the original programs did not trigger. We propose a similar approach for Solidity compilers, where ERWIN generates test programs to augment mutation-based fuzzers' seed pools, broadening their exploration and bug-finding capabilities. Extending the evaluation in §5.2, we configure ERWIN in `gen1` mode to produce test programs added to the seed pools of ACF and Fuzzol. Running these fuzzers for 24 hours daily over 20 days shows that combined, they cover 112 lines and 14 edges missed by unit tests, ERWIN, and these fuzzers. This enhanced coverage leads to discovering a bug overlooked by ERWIN and the fuzzers without ERWIN's seeds.

6.4 Generalizability of ERWIN

The concept of bounded exhaustive random program generation is applicable to other compilers that are similarly plagued by bug patterns tied to specific language features. This generation strategy helps narrow the vast search space inherent in random program generation, thereby improving bug detection efficiency. However, the design of ERWIN, including its template syntax and constraint types, is specifically tailored for Solidity. Adapting ERWIN to other compilers would likely require a more semantically general template syntax, new types of constraints, and a new code generation component. The modular design of ERWIN significantly reduces the engineering effort of integrating a new code generator, requiring approximately 1,500 lines of TypeScript code, substantially less than building a generator from scratch. We acknowledge the limitations of ERWIN in terms of syntax and constraint generality, and we will discuss potential extensions to address these issues in the Future Work section (§6.5).

6.5 Limitations and Future Work

Because our template syntax omits several Solidity constructs (see Figure 4), ERWIN can overlook compiler faults during testing, as shown in §5.2. To address this, we will expand support for inline assembly, interfaces, and libraries, thereby broadening the variety of generated programs and raising coverage for both compilers and analyzers. We will also introduce test oracles that flag non-crashing bugs. Beyond Solidity, we plan to port ERWIN to additional languages and compilers, fostering more effective compiler testing across diverse environments. Finally, we will relax the current qualifier-only constraints to embrace richer language-level restrictions. Although the present release is purely generative, the existing Solidity compiler test suites can seed template construction. Integrating them is another item on our roadmap.

7 Related Work

7.1 Program Generation

Program generators automatically create programs based on fixed rules or user specifications [15, 27] and are used to test compiler correctness. For example, Csmith [37], built on Randprog [18], generates random C programs featuring structs, pointers, and arrays without undefined behavior. Using differential testing on GCC and LLVM, Csmith discovered 325 previously unknown bugs. Inspired by Csmith's success, related generators like CsmithEdge [19], CUDAsmith [22], nnsmith [25], and MLIRsmith [36] have been developed for various compilers. However, Csmith faces limitations such as saturation [26] and difficulty in adapting to other languages [14]. To address saturation, yarpgen [26] introduces generation policies that bias program ingredient distributions, increasing diversity and delaying saturation. To broaden applicability, Hephaestus [14] proposes an IR lowerable to multiple languages (Java, Kotlin, Scala, Groovy), though this sacrifices language-specific insights and may miss certain bugs. This paper does not aim to extend Csmith to other languages or address its generalizability. Instead, we concentrate on incorporating bug-related templates into the generation process to improve test program quality.

7.2 Template-based Compiler Testing

Template-based compiler testing uses incomplete test programs with placeholders that are filled with random values to generate complete programs. This approach leverages embedded knowledge in templates, reducing the effort of generating programs from scratch. JAttack [38] applies this technique to JIT Java compilers, requiring manually crafted templates populated with random values to ensure validity. SPE [41] extracts templates from C test programs by replacing variables with placeholders and systematically explores variable usage patterns to alter dependencies, effectively probing bug triggers. MLIRsmith [36] generates MLIR programs via a two-phase process—template generation and instantiation—allowing expert knowledge integration and improved extensibility. Unlike JAttack, ERWIN autonomously generates diverse program templates from scratch. Compared to SPE, which explores all variable usage patterns, ERWIN investigates all valid qualifier combinations relevant to bugs. Unlike MLIRsmith, which produces a single test program per template, ERWIN explores the full spectrum of test programs derivable from a program template.

8 Conclusion

We propose bounded exhaustive random program generation, a method that systematically explores a high-quality, bug-relevant space within the vast search space of a programming language. Implemented in ERWIN for Solidity, this approach outperforms state-of-the-art fuzzers in bug detection efficiency. Additionally, ERWIN generates high-quality programs, covering code edges and lines missed by `solc`'s unit tests.

Acknowledgments

This work is supported by National Key Research and Development Program of China (No. 2024YFE0204200) and the Hong Kong Research Grant Council/General Research Fund (No. 16206524).

We are grateful to John Wickerson for his feedback on an earlier draft of this work.

References

[1] 2025. [Bug-Candidate]: slither hangs on a complicated test program. https://github.com/crytic/slither/issues/2619, last accessed on 2/13/2025.

[2] 2025. DefiLlama. https://defillama.com/.

[3] 2025. ICE in SolverInterface.cpp: Trying to create an 'equal' expression with different sorts. https://github.com/ethereum/solidity/issues/15647, last accessed on 2/13/2025.

[4] 2025. ICE on calling externally a function that returns calldata pointers. https://github.com/ethereum/solidity/issues/9134, last accessed on 2/13/2025.

[5] 2025. Slither. https://github.com/crytic/slither, last accessed on 2/13/2025.

[6] 2025. [SMTChecker] ICE in solidity::frontend::SMTEncoder::mergeVariables. https://github.com/ethereum/solidity/issues/8963, last accessed on 2/13/2025.

[7] 2025. [Sol->Yul] ICE in Whiskers render while copying calldata struct to storage. https://github.com/ethereum/solidity/issues/10105, last accessed on 2/13/2025.

[8] 2025. Solang. https://github.com/hyperledger-solang/solang, last accessed on 2/13/2025.

[9] 2025. Solidity. https://github.com/ethereum/solidity, last accessed on 2/13/2025.

[10] 2025. Use before assignment of calldata struct instance inside a function does not throw an error. https://github.com/ethereum/solidity/issues/15483, last accessed on 2/13/2025.

[11] 2025. Using this to get access to a state variable of struct instance causes an bool-type var. https://github.com/ethereum/solidity/issues/15525, last accessed on 2/13/2025.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[13] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485500

[14] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 183–198. https://doi.org/10.1145/3519939.3523427

[15] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2021), 4:1–4:36. https://doi.org/10.1145/3363562

[16] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

[17] Alastair F. Donaldson and Simon J. Gay. 2010. Type inference and strong static type checking for Promela. *Sci. Comput. Program.* 75, 11 (2010), 1165–1191. https://doi.org/10.1016/J.SCICO.2010.05.010

[18] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, Luca de Alfaro and Jens Palsberg (Eds.). ACM, 255–264. https://doi.org/10.1145/1450058.1450093

[19] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empir. Softw. Eng.* 27, 6 (2022), 129. https://doi.org/10.1007/S10664-022-10146-1

[20] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1219–1231. https://doi.org/10.1145/3597926.3598130

[21] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making no-fuss compiler fuzzing effective. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 194–204. https://doi.org/10.1145/3497776.3517765

[22] Bo Jiang, Xiaoyan Wang, Wing Kwong Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. CUDAsmith: A Fuzzer for CUDA Compilers. In *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020.* IEEE, 861–871. https://doi.org/10.1109/COMPSAC48688.2020.0-156

[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[24] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 65–76. https://doi.org/10.1145/2737924.2737986

[25] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 530–543. https://doi.org/10.1145/3575693.3575707

[26] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

[27] Haoyang Ma. 2023. A Survey of Modern Compiler Fuzzing. *CoRR* abs/2306.06884 (2023). https://doi.org/10.48550/ARXIV.2306.06884 arXiv:2306.06884

[28] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 248–260. https://doi.org/10.1145/3597926.3598053

[29] 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 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1312–1324. https://doi.org/10.1145/3650212.3680362

[30] Charalambos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis, and Dimitris Mitropoulos. 2023. Syntax-Aware Mutation for Testing the Solidity Compiler. In *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 14346)*, Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis (Eds.). Springer, 327–347. https://doi.org/10.1007/978-3-031-51479-1_17

[31] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

[32] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 968–980. https://doi.org/10.1145/3468264.3468591

[33] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2025. A Tale of Two DL Cities: When Library Tests Meet Compiler. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025.* IEEE, 2201–2212. https://doi.org/10.1109/ICSE55347.2025.00025

[34] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[35] Chenyao Suo, Junjie Chen, Shuang Liu, Jiajun Jiang, Yingquan Zhao, and Jianrong Wang. 2024. Fuzzing MLIR Compiler Infrastructure via Operation Dependency Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1287–1299. https://doi.org/10.1145/3650212.3680360

[36] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. 2023. MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023.* IEEE, 1555–1566. https://doi.org/10.1109/ASE56229.2023.00120

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[38] Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 23:1–23:13. https://doi.org/10.1145/3551349.3556958

[39] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT Testing with Template Extraction. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1129–1151. https://doi.org/10.1145/3643777

[40] Zhiqiang Zang, Fu-Yao Yu, Nathan Wiatrek, Milos Gligoric, and August Shi. 2023. JATTACK: Java JIT Testing using Template Programs. In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 6–10. https://doi.org/10.1109/ICSE-COMPANION58688.2023.00014

[41] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 347–361. https://doi.org/10.1145/3062341.3062379