# TENSURE: Fuzzing Sparse Tensor Compilers

Kabilan Mahathevan, Yining Zhang, Muhammad Ali Gulzar, Kirshanthan Sundararajah
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
Emails: {kabilan, yiningz, gulzar, kirshanthans}@vt.edu

***Abstract*—Sparse Tensor Compilers (STCs) have emerged as critical infrastructure for optimizing high-dimensional data analytics and machine learning workloads. The STCs must synthesize complex, irregular control flow for various compressed storage formats directly from high-level declarative specifications, thereby making them highly susceptible to subtle correctness defects. Existing testing frameworks, which rely on mutating computation graphs restricted to a standard vocabulary of operators, fail to exercise the arbitrary loop synthesis capabilities of these compilers. Furthermore, generic grammar-based fuzzers struggle to generate valid inputs due to the strict rules governing how indices are reused across multiple tensors.**

**In this paper, we present TENSURE, the first extensible black-box fuzzing framework specifically designed for the testing of STCs. TENSURE leverages Einstein Summation (Einsum) notation as a general input abstraction, enabling the generation of complex, unconventional tensor contractions that expose corner cases in the code-generation phases of STCs. We propose a novel constraint-based generation algorithm that guarantees 100% semantic validity of synthesized kernels, significantly outperforming the ∼3.3% validity rate of baseline grammar fuzzers. To enable metamorphic testing without a trusted reference, we introduce a set of semantic-preserving mutation operators that exploit algebraic commutativity and heterogeneity in storage formats. Our evaluation on two state-of-the-art systems, TACO and Finch, reveals widespread fragility, particularly in TACO, where TENSURE exposed crashes or silent miscompilations in a majority of generated test cases. These findings underscore the critical need for specialized testing tools in the sparse compilation ecosystem.**

## I. INTRODUCTION

Over the past decade, the demand for efficient execution of scientific computing and deep learning workloads has surged, pushing tensor compilers to the centre of modern systems. Compiler frameworks such as TVM [1], TensorRT [2], Triton [3], and XLA [4] aggressively optimise tensor computations by lowering high-level programs into hardware-specific code. Concurrently, tensor sizes are expanding rapidly while hardware scaling plateaus [5]. This divergence necessitates performing computation directly on compressed tensor storage formats.

However, it is non-trivial to construct optimized tensor computations, or kernels, for compound operations on various compressed tensor data storage formats. Unlike dense arrays, compressed formats do not support efficient random access; consequently, the resulting kernels require sophisticated loop nests to synchronize access across multiple sparse operands. Therefore, a compiler-based approach is widely advocated for sparse tensor operations. This need has driven the emergence of Sparse Tensor Compilers (STCs) as a dedicated class of tools for performing computations on compressed storage formats [6]–[11].

Unfortunately, the complexity of sparse traversal schedules significantly exceeds that of dense counterparts [6], [9], [10], [12]. The need to support diverse compressed data structures creates a large surface area for subtle correctness bugs—including iteration faults and memory safety violations—particularly when chaining multiple tensor operations.

Recent real-world failures demonstrate that compiler bugs can cause severe damage even in simpler dense settings. For instance, Anthropic traced a degradation in Claude's responses to an XLA miscompilation [13]. Diagnosing such issues is notoriously difficult because model outputs reveal little about the underlying compiled program, and developers implicitly trust the compiler. Sparse tensor pipelines amplify this challenge: the control flow is far more intricate, making defects significantly harder to isolate.

In this paper, we introduce an extensible automated testing framework specifically designed to verify the functional correctness of STCs. Existing approaches based on differential testing [14], [15] to validate general-purpose compilers are fundamentally incompatible with tensor compilers because of the non-negligible number of false positives due to variations in numerical accuracy across different execution environments [16]. Therefore, TENSURE adopts an approach similar to POLYJUICE, leveraging the same execution environment for metamorphic testing.

Conventional semantic-preserving mutation techniques used in traditional metamorphic testing, such as dead-code injection [17], instruction padding [18], and control-flow restructuring [14], [19]—rely on mutating imperative control structures (*e.g.,* if blocks, loops, and statement ordering). However, STCs typically operate on high-level declarative specifications, such as *Einsum* (Einstein summation) notation [20], where explicit control flow is absent. Consequently, these standard imperative mutations are inapplicable, as the control flow is not defined in the source but is synthesized during compilation, leaving no target for traditional code transformations.

Furthermore, the landscape of STCs is characterized by

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23006
www.ndss-symposium.org

significant syntactic and semantic fragmentation. Prominent frameworks like TACO [6], Finch [7], and the MLIR Sparse Dialect [9] each expose distinct declarative DSLs and support varying subsets of the einsum standard. This divergence is compounded by heterogeneous underlying infrastructures—ranging from C++ template libraries to LLVM-based lowering pipelines. Therefore, creating a unified testing framework is non-trivial; a test case valid for one tool is syntactically incompatible with another. To bridge this interoperability gap, an intermediate representation is required to abstract the core semantic features of STC programs, decoupling test generation from implementation-specific syntax.

Previous work on testing dense tensor compilers, such as NNSmith [21] and PolyJuice [22], relies on constructing computation graphs—directed acyclic structures where nodes represent fixed high-level operators (*e.g.,* MatMul, Conv2d) connected by data dependencies. This approach is inherently limited by the fixed vocabulary of the standard operator library; it tests the compiler's ability to optimize known patterns but fails to stress-test its capability to synthesize loop nests for arbitrary, unconventional tensor operations. Crucially, these frameworks also lack any representation of compressed storage formats, the backbone of STCs, thereby missing the complex traversal constraints required for sparse compilation.

To the best of our knowledge, no existing testing framework specifically targets functional correctness for STCs. This leaves a substantial portion of the tensor-compiler ecosystem effectively untested. We address this gap by developing a fuzzer that generates valid tensor kernels, translates them into the target sparse-tensor DSL, and creates semantically equivalent program variants for metamorphic testing. Unlike general-purpose fuzzing, where large sets of rewrite rules yield many equivalent variants, single-kernel tensor programs offer limited opportunities for mutation.

We evaluated TENSURE on two state-of-the-art systems: the C++-based TACO [6] and the Julia-based Finch [7]. Our experiments revealed significant robustness issues in TACO, where the fuzzer exposed crash-inducing inputs or miscompilations in over 60% of generated test cases. Furthermore, the successful integration with Finch validates the framework's extensibility to diverse compiler architectures. Collectively, these preliminary results highlight the pervasive fragility of the current sparse tensor compilation infrastructure and demonstrate the efficacy of TENSURE in detecting latent defects.

This paper makes the following contributions:

- **Extensible STC Fuzzer**: To the best of our knowledge, we present TENSURE, the first language-agnostic black-box fuzzing framework specifically designed to validate STCs.
- **Domain-Specific Mutation Operators**: We define a set of mutation operators that exploit *Storage Format Heterogeneity* and *Algebraic Commutativity* to generate semantically equivalent STC programs.
- **Constraint-Based Generation Algorithm**: We formalize and implement a generation algorithm that solves context-sensitive dimensional constraints to synthesize einsum expressions. Unlike standard grammar-based fuzzers, which achieve a validity rate of only ∼3.3%, our approach guarantees 100% semantic validity, enabling high-throughput testing of deep compilation passes.

The remainder of this paper is structured as follows. Section II provides background on sparse compilation, while Section III motivates the need for specialized fuzzing. The details of the design of TENSURE are provided in the Section IV. We present a preliminary evaluation of the tool in Section V Sections VI and VII discuss the limitations and related work, respectively, and we conclude the paper in Section VIII.

## II. BACKGROUND

### A. Tensors and Einsum

Tensors are multi-dimensional arrays, and their computations can be concisely expressed using einsum notation [23]. The einsum notation generalizes tensor operations by implying summation over shared indices between tensors, eliminating the need for explicit summation symbols.

$A(j) = B(i,j) * C(i)$ is a simple kernel with tensor contraction expressed in einsum notation. As formalized by the summation $A_j = \sum_i B_{ij} C_i$, this operation defines the output tensor $A_j$ (or $A(j)$) by accumulating the product of $B_{ij}$ and $C_i$ along the shared index $i$. It is important to note that while einsum specification is declarative—defining the data dependencies rather than the execution flow—it implies a *reduction* over the $i$ dimension. In other words, this operation performs a reduction over the $i$ dimension while accumulating contributions from $B_{ij}$ weighted by $C_i$. For reference, $A(i,j) = B(i,k) * C(k,j)$ presents the standard General Matrix–Matrix Multiplication (GeMM) using the declarative einsum notation, while $A_{ij} = \sum_k B_{ik} C_{kj}$ shows the corresponding algebraic definition expressed as summation.

### B. Tensor Compilers and Loop Lowering

Tensor compilers, such as TVM [1], XLA [4], and MLIR [24], serve as bridges between high-level mathematical notation and efficient machine code. To insulate scientists from low-level implementation details, these frameworks expose declarative DSLs. While einsum notation serves as a powerful generic abstraction for defining arbitrary contractions, modern compilers also support direct declarative definitions for ubiquitous tensor operations, such as matrix multiplications and 2D convolutions.

The fundamental task of these compilers is lowering—the automated translation of these declarative specifications into optimized, imperative loop nests. Algorithm 1 presents a pseudo-code representation of the lowering output for the kernel $A(j) = B(i,j) * C(i)$. Even for dense tensors, this translation is non-trivial: a concise mathematical expression expands into a complex sequence of nested loops, explicit memory addressing, and boundary checks. Consequently, the cyclomatic complexity of the generated code scales rapidly with the dimensionality of the tensors and the depth of the operation chain, creating significant potential for faults during lowering.

```c
int main() {
  int j_A = 0;
  for (int i_C=C_col_ptr[0]; i_C < C_col_ptr[1]; i_C++) {
    int i = C_row_ind[iC];
    for (int j = 0; j < B2_dimension; j++) {
      // B2_dimension->Dimension of the B's column axis
      A_values[j_A] = 0.0;
      int j_B = i * B2_dimension + j;
      A_values[j_A] += B_values[j_B] * C_values[i_C];
      j_A++;
    }
  }
}
```

Listing 1: TACO Generated Program (Buggy)

```c
int main() {
  for (int i_C=C_col_ptr[0]; i_C < C_col_ptr[1]; i_C++) {
    int i = C_row_ind[i_C];
    int j_A = 0;
    for (int j = 0; j < B2_dimension; j++) {
    // B2_dimension->Dimension of the B's column axis
      int j_B = i * B2_dimension + j;
      A_values[j_A] += B_values[j_B] * C_values[i_C];
      j_A++;
    }
  }
}
```

Listing 2: Manually Corrected Program

Fig. 1: A critical miscompilation detected by TENSURE for the kernel $A(j) = B(i,j) \cdot C(j)$. Listing 1 (Left) shows the original TACO-generated code containing an initialization error in the sparse iteration loop. Listing 2 (Right) shows the manually corrected implementation, highlighting the specific control flow logic required for correct execution.

### C. Compressed Data Structure

Unlike dense tensors, where almost all entries are nonzero, most real-world large-scale tensors are highly sparse. For example, the Amazon Reviews tensor in particular, contains 1.5 x $10^{19}$ components corresponding to 107 exabytes of data (assuming 8 bytes are used per component), but only $1.7 \times 10^9$ of the components (13 gigabytes) are non-zero [6], [25]–[27]. Sparse tensors address this imbalance by representing data in compressed formats such as Coordinate (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC) [28], or more specialized hybrid layouts. These formats store only nonzero values, along with their index metadata, allowing compilers to skip redundant iterations and avoid unnecessary multiplications and additions by zero. This is critical for achieving efficient computation on large-scale tensor workloads.

### D. Sparse Tensor Compilers

The Tensor Algebra Compiler (TACO) [6] pioneered the automated generation of sparse tensor kernels and serves as a primary subject of our evaluation. Following this precedent, major infrastructure frameworks including MLIR [9], TVM [8], XLA [29], and PyTorch [11], [30] have introduced sparse extensions. However, despite this broad interest, support for sparse operations in these systems remains largely experimental. The inherent complexity of sparse compilation has hindered the development of fully robust, production-ready implementations, resulting in a landscape of beta-level features that lack correctness guarantees. This pervasive fragility underscores the critical need for a dedicated testing framework to validate these evolving STCs.

### E. Automated Testing Techniques

Compiler validation typically relies on the synergy between robust test oracles and structured input generation. Differential and metamorphic testing has established itself as the gold standard for validating general-purpose and dense tensor compilers [14], [15], [17], [19], [21], [22]. Fundamentally, this approach is black-box and relies on the invariant that semantically equivalent inputs—or the same input executed on different compiler implementations—must yield identical execution results. This simplicity allows it to detect subtle functional correctness bugs without requiring a formal specification of the compiler's internal logic.

To drive these differential and metamorphic testing campaigns, compilers require highly structured inputs that satisfy strict syntactic rules. Grammar-based Fuzzing addresses this by generating syntactically valid test cases through adherence to a formal language specification, typically defined using standard Backus-Naur Form (BNF) or tool-specific Extended BNF (EBNF) dialects such as ANTLR. Unlike unstructured mutational fuzzers that apply random bit-flips to seed inputs, grammar-based tools, such as Grammarinator [31], LangFuzz [32], and Nautilus [33] construct inputs by performing random walks over the grammar's derivation tree. This technique is particularly effective for testing language processors, as it ensures that the generated inputs successfully pass the parser and exercise the deeper semantic analysis and lowering phases of the compiler.

## III. MOTIVATION

```c
int main() {
  for (int iC = C1_pos[0]; iC < C1_pos[1]; iC++) {
    int i = C1_crd[iC];
    for (int j = 0; j < B2_dimension; j++) {
      int jB = i * B2_dimension + j;
      A_vals[j] = A_vals[j] + B_vals[jB] * C_vals[iC];
    }
  }
}
```

Listing 3: Correct TACO-Generated Program for the Dense Case. Implements the kernel $A(j) = B(i,j) \cdot C(i)$ where $A$ is a dense vector, $B$ is a CSR matrix and $C$ is a sparse vector.

**Algorithm 1:** Dense Tensor Computation
$A(j) = B(i,j) \cdot C(j)$

**Input :** Dense tensor $B \in \mathbb{R}^{m \times n}$, vector $C \in \mathbb{R}^n$
**Output:** Vector $A \in \mathbb{R}^n$ such that $A_j = B_{ij} \cdot C_j$

```
1 for j ← 0 to n − 1 do
2     A[j] ← 0
3     for i ← 0 to m − 1 do
4         A[j] ← A[j] + B[i][j] · C[j]
5 return A
```

**Algorithm 2:** Sparse Tensor Computation (CSR)
$A(j) = B(i,j) \cdot C(j)$

**Input :** Sparse tensor $B$ in CSR format with arrays:
- $B$.val: nonzero values,
- $B$.col_idx: column indices,
- $B$.row_ptr: row pointers;

Vector $C \in \mathbb{R}^n$
**Output:** Vector $A \in \mathbb{R}^n$ such that $A_j = \sum_i B_{ij} \cdot C_j$

```
1 Initialize A[j] ← 0 for all j = 0, 1 . . . n − 1
2 for i ← 0 to m − 1 do
3     for k ← B.row_ptr[i] to B.row_ptr[i + 1] do
4         j ← B.col_idx[k]
5         A[j] ← A[j] + B.val[k] · C[j]
6 return A
```

$$\underbrace{\begin{bmatrix} 9 \\ 16 \\ 0 \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 0 & 4 & 0 \\ 2 & 8 & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{\mathbf{B}} \times \underbrace{\begin{bmatrix} 0 \\ 2 \\ 5 \end{bmatrix}}_{\mathbf{C}} = \begin{bmatrix} 0 \cdot 0 + 2 \cdot 2 + 1 \cdot 5 \\ 4 \cdot 0 + 8 \cdot 2 + 0 \cdot 5 \\ 0 \cdot 0 + 0 \cdot 2 + 0 \cdot 5 \end{bmatrix} \quad (1)$$

To illustrate the challenges of sparse tensor compilation, consider the kernel: $A(j) = B(i,j) \cdot C(j)$. Here, each column of the tensor $B$ is scaled and reduced by the elements of vector $C$ to compute the elements of $A$. Algorithm 1 depicts the standard lowering for dense tensors, where elements are accessed via direct addressing. In contrast, Algorithm 2 demonstrates the corresponding pseudocode when $B$ is stored in CSR format. The transition to sparse storage forces the compiler to replace affine loop nests with complex data-dependent irregular iterations that traverse compressed index arrays. Regardless of the underlying format, the computation is expected to produce the deterministic results shown in Equation 1.

To demonstrate the impact of output storage formats on code generation, Listing 1 and Listing 3 present the code generated by TACO for this kernel using different storage formats for the output tensor $A$. While the dense output variant executes correctly, the sparse output variant produces erroneous results. The defect, exposed by TENSURE, stems from a miscompilation in the loop initialization. As identified in Listing 1 (Lines 2 and 7), the compiler fails to reset the coordinate tracker $j\_A$ within the loop correctly. The corrected implementation, shown in Listing 2, moves the initialization inside the loop body. This example underscores the inherent fragility of STCs: even elementary kernels can trigger errors that evade manual inspection.

Addressing these failures requires automated metamorphic testing. In the domain of dense tensor compilers, prior work has relied on *Computation Graph-based Fuzzing* [21], [22]. This methodology constructs extensive sequences of tensor operations and mutates the graph topology and individual operations to generate semantic equivalents. However, this approach is ill-suited for STCs for three reasons. First, the overhead of graph management incurs prohibitive latency, often limiting throughput to approximately four mutant executions per second [22], thereby restricting test coverage. Second, graph-level mutations produce bloated failure cases that require computationally expensive minimization to isolate the root cause. Finally, these tools are constrained to a fixed library of high-level operators, limiting their ability to stress the arbitrary iteration patterns defined by general einsum notation.

Moreover, a wider class of STCs treats tensor operations as isolated lowering targets. The compilation process resembles a macro expansion where each einsum expression is translated into a standalone loop nest, oblivious to the broader dataflow context. This design characteristic renders graph-based generation ineffective for metamorphic testing. Because modifications to one operation in a sequence do not influence the lowering strategy of its neighbors. Therefore, the search space for bugs is effectively partitioned by operation. Hence, a fuzzer that generates long chains of independent operations merely retests the same isolated expansion mechanisms repeatedly, validating the redundancy of graph-level mutation in this domain.

A potential alternative is to employ generic grammar-based fuzzers, such as Grammarinator [31], to synthesize einsum expressions directly. However, the validity of the einsum notation is governed by *context-sensitive* constraints. Standard grammar-based fuzzers, which typically operate on Context-Free Grammars, lack the semantic awareness to enforce these cross-reference constraints. Consequently, they produce a high volume of invalid kernels that are rejected by the compiler frontend and fail to reach the critical loop-lowering passes.

These limitations motivate the need for a specialized testing framework. Effective validation of STCs requires a system capable of (1) generating valid einsum expressions, (2) systematically mutating the expressions to generate semantically equivalent programs, and (3) leveraging metamorphic relations to detect semantic divergence without a reference compiler.

## IV. DESIGN & IMPLEMENTATION

Unlike dense tensor compilers, which construct a global computation graph to manage lowering and optimization across a sequence of operations, STCs take a different approach. Rather than maintaining a graph-like structure for the

**Algorithm 3:** Random Einsum Expression Generation

**Input :** $N$: number of input tensors,
$R_{\max}$: maximum tensor rank,
$I$: pool of index labels

**Output:** Random einsum expression
$A = B_1 * B_2 * \cdots * B_N$

**Step:** 1: Assign indices to input tensors

**for** $k \leftarrow 1$ **to** $N$ **do**
  $r_k \leftarrow \text{UniformRandom}(1, R_{\max})$
  $B_k$.indices $\leftarrow$ randomly select $r_k$ distinct indices from $\mathcal{I}$
  Update usage count of selected indices

**Step:** 2: Determine output tensor indices

$\mathcal{O} \leftarrow$ random subset of indices with non-zero usage

**Step:** 3: Ensure valid reduction indices

**for** *each index* $i \in \mathcal{I} \setminus \mathcal{O}$ *with usage count* $= 1$ **do**
  Add $i$ to a different input tensor to ensure it occurs at least twice

**Step:** 4: Assign tensor shapes

**for** *each index* $i \in \mathcal{I}$ **do**
  assign random dimension $d_i$

**for** *each tensor* $B_k$ **do**
  **for** *each* $j$ *in* $B_k$.*indices* **do**
    $B_k.\text{shape}[j] \leftarrow d_{B_k.\text{indices}[j]}$

**Step:** 5: Construct einsum expression

$A(\mathcal{O}) = \prod_{k=1}^{N} B_k(B_k.\text{indices})$

**return** $A$ and $\{B_1, \ldots, B_N\}$

entire operation sequence, STCs focus on constructing an *iteration graph* for a single einsum expression in isolation. Hence, the complexity of STCs lies in synthesizing the loop nests for individual operations, rather than performing graph-level rewrites on a multi-node sequence. In this context, chaining multiple sequences of einsum operations yields diminishing returns for testing, since the complexity lies in the synthesis of iterations rather than in the graph topology. Therefore, our fuzzer diverges from graph-based generation; instead, it synthesizes a single einsum notation to represent complex tensor operations, as detailed in Algorithm 3.

### A. Random Kernel Generation

While einsum equations appear syntactically simple, their validity relies on context-sensitive constraints that exceed the expressive power of CFGs. A fundamental validity rule is that the set of output indices $\mathcal{O}$ must be a subset of the union of all input indices $\bigcup \mathcal{I}_{\text{in}}$ (*i.e.,* $\mathcal{O} \subseteq \bigcup \mathcal{I}_{\text{in}}$). Therefore, it implies that the validity of the output symbols is dependent on the specific symbols consumed earlier in the input sequence. This dependency violates the *Pumping Lemma* for Context-Free Languages [34], classifying valid einsum expressions as a Context-Sensitive Language.

To synthesize a structurally valid einsum expression, we view the generation process as a constraint satisfaction problem involving *index connectivity* and *dimensional consistency*. First, we populate the index sets for all input tensors $(B_1, B_2, \ldots, B_N)$ by sampling from a global index pool $\mathcal{I}$, strictly enforcing a maximum rank $R_{\max}$ per tensor. We then partition the used indices into two sets: the output indices $\mathcal{O}$ (preserved dimensions) and the contraction indices $S = \mathcal{I} \setminus \mathcal{O}$. A critical validity constraint in einsum semantics is that a contraction index typically bridges dimensions across tensors. If an index $s \in S$ appears in fewer than two input tensors, the contraction is ill-defined. To resolve this, we enforce connectivity: we iterate through $S$ and inject any under-represented contraction indices into an additional randomly selected input tensor. Once the symbolic structure is validated, we map the abstract indices to concrete runtime dimensions. By assigning a distinct integer size to each unique index in $I$ and propagating these sizes to the corresponding axes of $(B_1, B_2, \ldots, B_N)$, we guarantee that the generated tensors possess compatible shapes for the specified contraction.

### B. Mutation Operators

However, constraining the search space to individual einsum operations precludes using traditional graph-level rewrite rules as mutation operators. We address this by introducing two invariance-based mutation strategies: *Algebraic Commutativity* and *Storage Format Heterogeneity*.

The algebraic commutativity is grounded in tensor algebra: assuming the tensor elements belong to a commutative ring (*e.g.,* the field of real numbers $\mathbb{R}$), the multiplication operation is commutative (*i.e.,* $\forall a, b \in \mathbb{R},\ a \cdot b = b \cdot a$). This property extends to einsum operations, where the order of operands does not alter the semantic result. For instance, the expression $A(i,j) = B(i,k) * C(k,j)$ is semantically equivalent to $A(i,j) = C(k,j) * B(i,k)$. By permuting operands, we force the compiler to generate different iteration schedules for the same mathematical operation.

The second operator, *Storage Format Heterogeneity*, targets the complexity of sparse iteration schemes for different compressed storage formats. In sparse tensor compilation, the mathematical definition of a kernel is orthogonal to the physical layout of its data. A tensor $B(i,j)$ contains the same nonzero values in the exact coordinates whether stored in COO, CSR, CSC, or any other compressed storage format. However, the choice of format significantly alters the code-generation path, as shown in Figure 1. We leverage this decoupling by randomly assigning distinct storage formats to each input and output tensor in the generated einsum expression. This forces the compiler to synthesize unique iteration graphs and loop nests for every format combination. Because the semantics of the computation remain invariant, any divergence in the output across these format permutations indicates a compilation fault in handling specific access patterns or compressed data formats. By composing these two mutation operators—permuting operand order and varying storage formats—we create a rich space of semantically equivalent test programs from a single einsum expression.

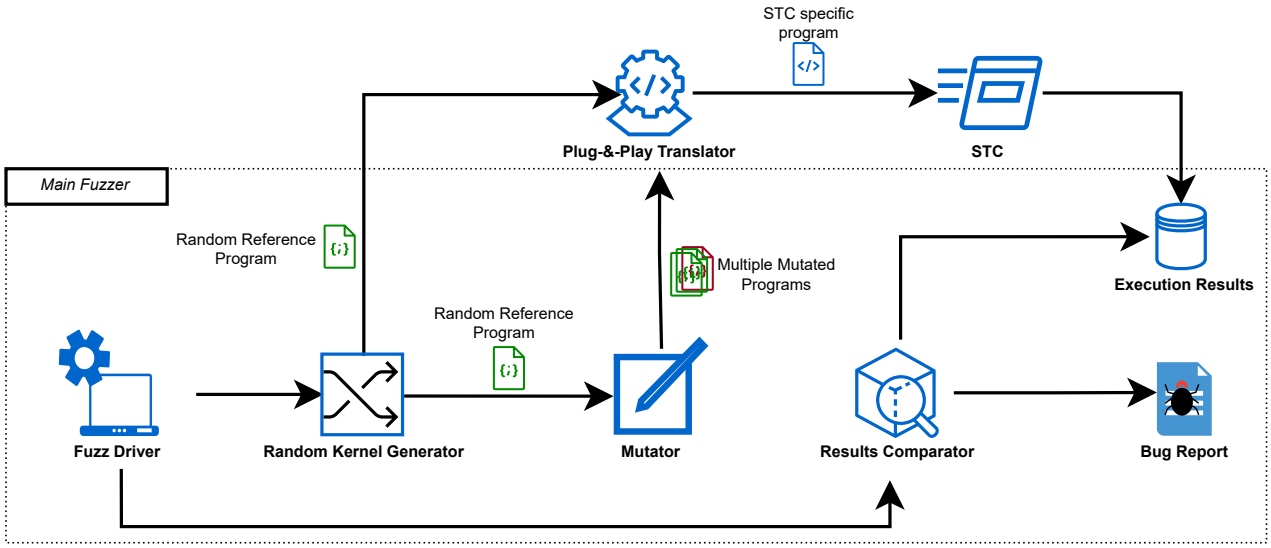


Fig. 2: The main fuzzing loop. The system generates a random tensor program and a few mutated variants. Both versions are compiled and executed, and the outputs are compared to identify mismatches indicating compiler bugs.

### C. Language-Agnostic Architecture

While our mutation operators generate a rich space of abstract test programs, executing them requires navigating the syntactic fragmentation of the STC landscape. Since most STCs operate as macro expanders within high-level host languages (*e.g.,* C++ for TACO, Julia for Finch), a direct-generation approach is not inherently portable. To ensure broad compatibility, we designed a generic JSON abstraction layer that captures all semantic details of the synthesized kernel, including randomized storage-format configurations. This design shifts the integration burden from the fuzzer to a thin abstraction layer: compiler developers can integrate their tools simply by implementing a lightweight translator that parses this JSON schema and emits the corresponding domain-specific sparse tensor program.

Figure 2 illustrates the decoupled execution workflow of our fuzzer. The core fuzzing engine is isolated from the target STC and interacts solely through the abstraction layer. At runtime, the fuzzer first dispatches a randomly selected reference program to the target STC via the translator. Upon successful execution, it generates semantically equivalent mutants—applying the commutativity and storage format heterogeneity operators—and executes them against the same backend. The fuzzer then acts as a metamorphic oracle, comparing the tensor outputs of the mutants against the reference result. Any discrepancy flags a correctness violation in the STC's iteration schemes. This plugin-based architecture ensures the system remains extensible to future research and industrial compilers.

### D. Program Execution & Bug Detection

Executing a tensor kernel requires not only the code but also valid input data. Complementing its program generation, TENSURE automatically synthesizes small input tensors that strictly adhere to the *dimensional consistency* of the einsum specification. Once the reference program successfully completes execution, its output is recorded as the ground truth. The mutated program is then executed, and its results are compared against this baseline. If a discrepancy is detected, TENSURE aggregates the input tensors, both program variants, and their respective outputs into a detailed bug report.

## V. EVALUATION

To empirically quantify the advantage of our generation strategy, we established a baseline using *Grammarinator* [31], a state-of-the-art grammar-based fuzzer configured with a standard context-free grammar for einsum notation. We conducted a large-scale comparative study, synthesizing a corpus of one million ($10^6$) kernels using both the baseline and our custom generator.

The results reveal a stark disparity in the efficiency of the generation. While our constraint-based algorithm guarantees a 100% validity rate by construction, the grammar-based baseline produced valid kernels in only $\sim$3.3% of attempts. The vast majority of grammar-generated inputs were rejected due to dimensional inconsistencies. This result confirms that generic grammar fuzzers are fundamentally inefficient for tensor compiler testing, as they waste over $\sim$96% of the fuzzing budget on syntactically valid but semantically malformed inputs.

We integrated our constraint-based einsum generator into the TENSURE framework to serve as the core input synthesis engine. To evaluate its effectiveness, we conducted a sustained six-hour fuzzing campaign targeting two state-of-the-art sparse tensor compilers: TACO and Finch, and the results can be found in Table I.

TABLE I: Bugs found by TENSURE over a 6-hour fuzzing campaign.

| **STCs** | #Iterations | STC-NA | C-Bugs | WC-Bugs |
|---|---|---|---|---|
| TACO | 267k | 236k | 14.4k | 5,758 |
| Finch | 1,619 | 7 | 57 | 0 |

**Note:** STC-NA: Not Acceptable Input for STC; C-Bugs: Number of Crash Bugs; WC-Bugs: Number of Wrong-Code Bugs (Silent Errors). '#Iterations' denotes total fuzzing iterations.

During the fuzzing lifecycle, TENSURE categorizes execution anomalies into three distinct failure scenarios that vary in diagnostic fidelity. The first scenario involves the failure of the initial reference kernel; since many STCs support only a strict subset of the einsum standard, these rejections frequently stem from unsupported features rather than genuine defects. The second scenario occurs when the reference kernel executes successfully, but a semantically equivalent mutant triggers a compilation error or runtime crash. While this often indicates a valid crash bug, it may also stem from gaps in the compiler's support for specific iteration schemes. The third and most critical scenario arises when both the reference and mutant programs execute successfully but yield divergent outputs. This constitutes a *functional correctness violation*, or silent miscompilation. Unlike crashes, which are often caught by runtime assertions, these divergences indicate that the compiler has generated incorrect sparse iteration schemes for a valid mathematical operation.

### A. TACO Evaluation Results

Among the valid einsum operations accepted by the compiler's frontend, TENSURE exposed defects in approximately ∼65.2% of cases. Crucially, ∼18.6% of these failures were classified as critical miscompilations (wrong code bugs), where the compiler silently generated incorrect code.

We attribute this high volume of defects to two primary factors. First, the reported figures reflect total failure counts; determining unique root causes would require exhaustive manual inspection, which was infeasible given the scale of failures. Second, TACO was designed primarily as a foundational research prototype to demonstrate sparse compilation concepts, rather than as a production-hardened system. As the project is no longer actively maintained or accepting bug reports, we focused our analysis on aggregate failure rates to demonstrate the fuzzer's efficacy, rather than performing granular deduplication for reporting purposes. Due to the black-box nature of TENSURE, reliable deduplication is inherently difficult, as a single underlying compiler defect can be triggered by multiple, syntactically distinct einsum expressions.

### B. Finch Evaluation Results

While our campaign against TACO evaluated over 267,000 kernels and uncovered thousands of defects, the evaluation on Finch was constrained by significant compilation latency, completing only 1,619 iterations in the same six-hour window. Despite this reduced throughput, TENSURE successfully identified 57 crash bugs in the Finch compiler. No silent miscompilations were detected within this limited sample size, though the presence of crash-inducing inputs confirms the fuzzer's ability to stress the Julia-based infrastructure.

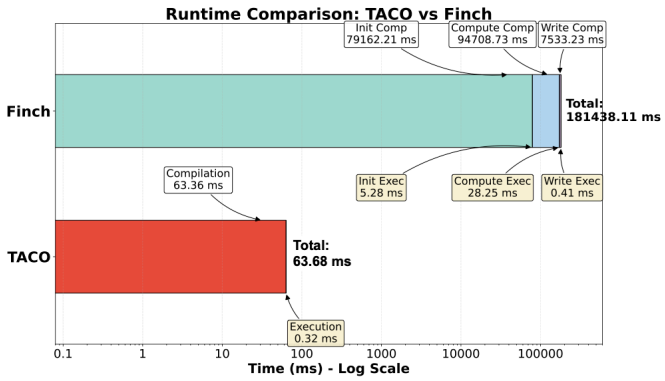


Fig. 3: Runtime Comparison: TACO vs Finch. The time axis uses a log scale to capture the disparity in compilation times.

Two factors explain this disparity. First and most significantly, the compilation time for Finch kernels is the primary bottleneck. As shown in Figure 3, TACO processed a single kernel in approximately 64 ms, whereas Finch required over 181 s per kernel. The complicated data layout of sparse tensors adds significant overhead to the *Initialization* and *Computation* compilation stages, a cost that is incurred repeatedly for every iteration. Second, the current evaluation was restricted to a preliminary six-hour window; we reserve more extensive, longitudinal fuzzing campaigns for future work.

One way to improve throughput is to reduce the granularity of the test variations. By isolating changes to specific expressions, we could allow the Julia compiler to only recompile the mutated parts instead of the whole program. However, this requires structural changes and specialization of the fuzzer.

## VI. DISCUSSION AND FUTURE WORK

A persistent challenge in validating numerical software is the handling of floating-point determinism. While our mutation operator exploits the commutative property of einsum operations, this high-level reordering often compels the compiler to alter the low-level reduction schedule. Since IEEE-754 floating-point arithmetic is not associative, these scheduling changes introduce minor numerical deviations. Currently, TENSURE employs a relaxed $\epsilon$-based comparator to tolerate this noise; however, this introduces a trade-off in which loose tolerances may mask subtle correctness bugs. In the future, we will mitigate this by introducing integer-only test modes to validate synthesized control flow strictly. This approach eliminates numerical noise, enabling the unambiguous distinction between valid algorithmic reordering and genuine miscompilation.

Beyond addressing precision challenges, a promising approach for enhancing the fuzzer is the systematic exploita-

tion of tensor transposition. Currently, our strategy relies on operand commutativity and storage format heterogeneity. However, by leveraging the algebraic invariance of tensor contractions under index permutation, we can introduce a new class of Transpose Mutation Operators. These operators would transform the input tensors by permuting their dimensions (effectively computing $A^T$) while adjusting the contraction indices to preserve semantic equivalence. This targeted approach would rigorously stress-test the compiler's ability to handle permuted access patterns, exposing defects in the index map generation logic that simple operand reordering cannot reach.

Moreover, we do not employ advanced deduplication strategies for redundant bug reports (Section V-A), nor do we implement compiler-specific optimizations to accelerate compilation times for targets like Finch (Section V-B). Addressing these challenges would require inspecting internal compiler states or tailoring the generator to specific backends, modifications that would fundamentally alter TENSURE from a general-purpose black-box fuzzer into a specialized grey- or white-box tool.

In future work, we plan to expand our evaluation by benchmarking TENSURE against a broader range of state-of-the-art baselines [33], [35], [36] to further validate its efficacy. Finally, we aim to demonstrate the extensibility of our framework by integrating emerging platforms such as the MLIR Sparse Dialect and PyTorch Sparse. While integrating PyTorch Sparse is straightforward due to its native einsum support, targeting MLIR Sparse Dialect presents a significant semantic gap, as the dialect does not directly consume high-level einsum declarations. Consequently, this integration will require enhancing our abstraction layer to translate our JSON-based intermediate representation into the MLIR Sparse dialect, thereby enabling a robust comparative analysis of diverse sparse compiler infrastructures.

## VII. RELATED WORK

Automated compiler validation is a well-established area of research. The seminal work of Csmith [14] demonstrated the efficacy of random differential testing, revealing hundreds of bugs in GCC and LLVM by generating valid C programs from scratch. Building on this foundation, subsequent frameworks such as Orion [17] and YarpGen [15] introduced sophisticated mutation-based strategies. These tools create semantically equivalent test cases by injecting dead code or simplifying arithmetic expressions.

The proliferation of deep learning compilers has catalyzed the development of specialized testing frameworks. NNSmith [21] pioneered the constraint-based generation of valid computation graphs, excelling at verifying diverse graph topologies and operator combinations. Building on this foundation, PolyJuice [22] employed equality saturation to synthesize semantically equivalent graph variants, specifically targeting defects in graph-level optimizers. At the same time, HiraGen [37] focused on stressing high-level optimization passes through systematic graph mutations.

More recently, Large Language Models (LLMs) have been adapted for input generation. TitanFuzz [38] utilizes LLMs in a black-box manner to synthesize operator sequences without internal compiler knowledge. In contrast, WhiteFox [39] introduces a white-box approach that leverages LLMs to analyze the compiler's source code and optimization passes to generate targeted test inputs. However, all these frameworks primarily target dense computation graphs constructed from standard operators, leaving the specific challenges of sparse tensor compilation and lowering phases largely unaddressed.

The landscape of sparse compilation is rapidly evolving, driven by the need to decouple algorithm specification from data representation. TACO [6] pioneered the concept of format-agnostic compilation, introducing a lattice-based theory to synthesize code for arbitrary sparse tensor formats. This foundational work has since been adopted by modern frameworks, including MLIR's Sparse Tensor Dialect [9], Finch [7], and TVM's SparseTIR [8], as well as industrial extensions to XLA [29] and PyTorch [11], [30]. As these systems transition from research prototypes to critical production infrastructure, the need for robust, specialized testing frameworks becomes increasingly acute.

Grammar-based fuzzing is a standard technique for testing language processors. Tools such as Grammarinator [31], LangFuzz [32], and Nautilus [33] generate inputs by traversing formal grammars, proving highly effective for syntactic testing across a wide range of languages. However, generic grammar-based fuzzers, which typically operate on Context-Free Grammars, lack the intrinsic mechanism to enforce these semantic constraints required by the sparse tensor programs in einsum notation.

## VIII. CONCLUSION

In this paper, we present TENSURE, the first extensible black-box fuzzing framework specifically architected for the automated testing of STCs. Unlike prior frameworks constrained by fixed operator libraries [21], [22], TENSURE leverages einsum notation as a fully general input abstraction to synthesize arbitrary tensor contractions. This approach bypasses the limitations of standard operator graphs, effectively stressing the compiler's synthesis algorithms for unconventional operations and exposing corner cases in critical lowering phases.

By combining a constraint-based generation algorithm that guarantees 100% validity with a metamorphic oracle leveraging algebraic commutativity and storage format heterogeneity for semantic-preserving mutation, TENSURE overcomes the limitations of traditional grammar fuzzers to rigorously validate sparse iteration schemes without a trusted reference compiler. We demonstrated the utility of TENSURE by targeting two distinct STCs, TACO and Finch. Our evaluation of TACO revealed significant robustness issues, with the fuzzer exposing crashes or miscompilations over $\sim$60% of generated test cases. These findings, alongside the successful adaptation to the Finch ecosystem, underscore the fragility of the current sparse compilation landscape and highlight the necessity of specialized tools like TENSURE to ensure the reliability of next-generation infrastructure.

## REFERENCES


[1] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," *arXiv*, Feb. 2018.

[2] "NVIDIA TensorRT," Nov. 2025, [Online; accessed 17. Nov. 2025]. [Online]. Available: https://developer.nvidia.com/tensorrt

[3] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[4] "XLA," Dec. 2024, [Online; accessed 17. Nov. 2025]. [Online]. Available: https://openxla.org/xla

[5] L. B. Kish, "End of moore's law: thermal (noise) death of integration in micro and nano electronics," *Physics Letters A*, vol. 305, no. 3-4, pp. 144–149, 2002.

[6] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[7] W. Ahrens, T. F. Collin, R. Patel, K. Deeds, C. Hong, and S. Amarasinghe, "Finch: Sparse and structured tensor programming with control flow," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 1042–1072, 2025.

[8] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze, "Sparsetir: Composable abstractions for sparse compilation in deep learning," ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 660–678. [Online]. Available: https://doi.org/10.1145/3582016.3582047

[9] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler Support for Sparse Tensor Computations in MLIR," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–25, Sep. 2022.

[10] A. J. C. Bik and H. A. G. Wijshoff, "Compilation techniques for sparse matrix computations," in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 416–424. [Online]. Available: https://doi.org/10.1145/165939.166023

[11] "Reference API - PyTorch Sparse," Nov. 2025, [Online; accessed 20. Nov. 2025]. [Online]. Available: https://docs.pytorch.org/docs/stable/sparse.html

[12] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 99–108. [Online]. Available: https://doi.org/10.1145/2751205.2751244

[13] "A postmortem of three recent issues," Sep. 2025, [Online; accessed 18. Nov. 2025]. [Online]. Available: https://www.anthropic.com/engineering/a-postmortem-of-three-recent-issues

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[15] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428264

[16] D. Xiao, Z. LIU, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3508035

[17] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, p. 216–226, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594334

[18] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133917

[19] "CLSmith," Nov. 2025, [Online; accessed 20. Nov. 2025]. [Online]. Available: https://github.com/ChrisLidbury/CLSmith

[20] A. Einstein, "The foundation of the general theory of relativity," 1916.

[21] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[22] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang, "Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689757

[23] G. Ricci and T. Levi-Civita, "Méthodes de calcul différentiel absolu et leurs applications," *Mathematische Annalen*, vol. 54, no. 1–2, p. 125–201, 1901. [Online]. Available: http://eudml.org/doc/157997

[24] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[25] R. M. Bell and Y. Koren, "Lessons from the netflix prize challenge," *Acm Sigkdd Explorations Newsletter*, vol. 9, no. 2, pp. 75–79, 2007.

[26] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM workshop on Online social networks*, 2009, pp. 37–42.

[27] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, 2013, pp. 165–172.

[28] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.

[29] "Working with sparse tensors," Oct. 2024, [Online; accessed 20. Nov. 2025]. [Online]. Available: https://www.tensorflow.org/guide/sparse_tensor

[30] B. Yan, A. J. Root, T. Gale, D. Broman, and F. Kjolstad, "Scorch: A Library for Sparse Deep Learning," *arXiv*, May 2024.

[31] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 45–48. [Online]. Available: https://doi.org/10.1145/3278186.3278193

[32] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 38.

[33] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, vol. 19, 2019, p. 337.

[34] H.-J. Kreowski, "A pumping lemma for context-free graph languages," in *Graph-Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 270–283.

[35] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[36] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[37] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 248–260. [Online]. Available: https://doi.org/10.1145/3597926.3598053

[38] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[39] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language

models," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689736