LLM-Assisted Input-Requirement-Aware Differential Testing of Array Programming Frameworks

ZHICHAO ZHOU, ShanghaiTech University, China
JINGZHU HE*, ShanghaiTech University, China

Array programming (AP) frameworks (e.g., NumPy and Octave) are widely adopted in scientific computing. Critical defects can jeopardize the entire ecosystem. The stability of API designs enables differential testing on various implementations (e.g., two versions). However, two primary obstacles remain. First, current test generation cannot effectively generate valid inputs, as the APIs (e.g., matrix multiplication) have type constraints and semantic requirements. Second, unit testing approaches test APIs independently, but they share a core N-dimensional array structure (ndarray) as inputs. Modifying one API may alter the ndarray's properties, breaking the correctness of others. We propose a differential testing tool for array programming, called ArrayDiff. We first collect semantic requirements from NumPy's APIs and leverage LLMs to transfer NumPy's requirements to other frameworks. Then, we propose an input-requirement-aware API call generator (IRA-ACG). Based on IRA-ACG, ArrayDiff employs search algorithms to evolve tests while ensuring valid inputs. ArrayDiff can generate valid and complex API call sequences to detect potential differences. We evaluate ArrayDiff and its ablation versions on five AP pairs. They detect 47 valid-input differences and 39 invalid ones, with 23 confirmed as bugs or document issues. IRA-ACG boosts the detection of valid-input differences, which constitute most confirmed bugs. Comparing ArrayDiff with TitanFuzz (LLM-based fuzzer) and Ghostwriter (unit tester) confirms the benefits of IRA-ACG and sequence-level testing.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Search-based software engineering*.

Additional Key Words and Phrases: array, random algorithm, genetic algorithm, differential testing

ACM Reference Format:
Zhichao Zhou and Jingzhu He. 2026. LLM-Assisted Input-Requirement-Aware Differential Testing of Array Programming Frameworks. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE155 (July 2026), 24 pages. https://doi.org/10.1145/3808162

1 Introduction

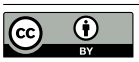
```
1  import numpy as np
2  v1 = np.ones(16, dtype="int64")
3  v2 = np.reshape(v1, (4, 4))
4  print(np.trace(v2)) # output: 4
```

Fig. 1. A NumPy Example.

Array programming (AP) offers a powerful and expressive syntax for manipulating data in multi-dimensional arrays [37]. Frameworks like NumPy [62], Matlab [56], and Octave [64] leverage this paradigm to provide high-level abstractions for scientific computing. They are essential to various domains [37] such as deep learning [50, 72] and astronomy [77]. Their key data structure is the N-dimensional array (a.k.a., *ndarray*). An ndarray is a container of a specific shape that contains data of a particular scalar type (dtype). They provide diverse APIs for the ndarray, such as creation, manipulation, and

*Jingzhu He is the corresponding author.

Authors' Contact Information: Zhichao Zhou, ShanghaiTech University, Shanghai, China, zhouzhch@shanghaitech.edu.cn; Jingzhu He, ShanghaiTech University, Shanghai, China, hejzh1@shanghaitech.edu.cn.



ACM 2994-970X/2026/7-ARTFSE155
https://doi.org/10.1145/3808162

mathematical operations. Fig. 1 provides an example with NumPy. It creates a 1-D ndarray whose dtype is integer, reshapes it to a matrix, and computes the diagonal sum. Due to their fundamental role in numerical computation, even a single defect can have a critical impact on the scientific computing ecosystem. A particular API difference between two implementations (for example, two versions) can be a severe bug [18, 76]. It inspires us to perform differential testing [24, 76], i.e., comparing outputs of two implementations under identical tests to detect potential defects.

Typically, the differences come from three major parts. First, continuous updates of these frameworks may introduce differences (e.g., bugs and API changes). Second, a framework's same version runs on different CPU architectures (e.g., AMD and ARM), which could behave differently due to not fully masking architectural discrepancies. Third, several API-compatible libraries are proposed to exploit specific data-parallel devices like GPUs. For example, CuPy [65] (compatible with NumPy) offers improved performance via NVIDIA GPUs. A critical factor for migrating developers is that these libraries' functionalities should behave as closely as possible to the original framework. Meanwhile, developers need to know the inherent differences between them (e.g., different reactions to undefined behaviors) to ensure that their programs behave correctly after migration.

```
1  import cupy as cp
2  # out triggers an unexpected exception
3  cp.nanargmin(a, out=b)
4  # CuPy's manual test:
5  data = random_array()
6  # without out
7  a = np.nanargmin(np.array(data))
8  b = cp.nanargmin(cp.array(data))
9  assert_array_equal(a, b)
```

Fig. 2. CuPy-8782: An exception detected by ArrayDiff in `nanargmin` with the ineffectual manual test.

To be compatible with the origin, API-compatible libraries' maintainers have adopted differential testing in their manual unit tests [10]. However, the array framework provides numerous APIs to deliver diverse functionalities, each of which may contain several optional parameters to provide fine-grained customization. Manually testing such complex logic is tedious and a heavy burden, calling for automated testing to bridge the gap. Fig. 2 provides an example: a CuPy previously unknown exception detected by our tool, ArrayDiff. It calls `nanargmin` to get `a`'s minimum value's index with an optional parameter `out=b` to set the result in `b` instead of returning it. However, `nanargmin` incorrectly swaps `out` with another optional parameter `dtype` for calling an inner function, and a crash happens due to a type error. CuPy's manual test (Lines 4-9) misses this bug because it only compares `nanargmin` with default parameters (`out` and `dtype` as `None`), failing to exhaustively explore program behaviors.

1.1 Challenges and Our Approach

We aim to automatically detect differences between AP framework implementations. These differences include implementation bugs and inherent differences (e.g., numerical errors).

Current testing techniques for AP frameworks face two primary challenges. First, they struggle to generate valid inputs that trigger comprehensive program behaviors. Array APIs impose strict semantic requirements beyond basic language grammar and type constraints. For example, the `matmul` operation necessitates $(m \times n)$ and $(n \times p)$ dimension compatibility. While grammar-based fuzzers [30, 83] generate inputs with a grammar-based specification and automated test generation tools [25, 51, 69] ensure type correctness, they lack the domain-specific knowledge required to satisfy these semantic constraints, leading to invalid tests. To address this challenge, TitanFuzz [16], which tests deep learning libraries with many array operations, employs LLMs to generate inputs; however, its evaluation shows that the validity rate is only about 40%, which is confirmed by our experiment. The second challenge is that many tools create tests for each API independently (i.e., unit testing) [17, 45, 51, 53, 89]. However, AP frameworks' primary characteristic is that most APIs use ndarrays as input/output. Modifying one API may modify the ndarray's properties, breaking other API's correctness. Hence, they could miss many bugs caused by chained API sequences [16].

```python
import numpy as np
# (1) diagflat's expected behavior:
a = np.array(1.) # a: 1.
print(np.diagflat(a).shape) # output: (1,1)
# (2) ArrayDiff's generated test:
a = np.var(np.array([1,3])) # a: 1.
print(np.diagflat(a).shape) # output: ()
```

(a) Bug reproduction.

```python
# (1) Pynguin's generated test for diagflat:
none_type_0 = None
diagflat(none_type_0)
# (2) Ghostwriter's generated test for diagflat:
@given (v=arrays(dtype=dtypes(), shape=shapes()),
k=integers()):
    diagflat(v, k)
```

(b) Tests of Pynguin and Ghostwriter.

Fig. 3. NumPy-27783 detected by ArrayDiff: Violate `diagflat` contract that the output is 2-D.

Input-requirement-aware API call generation: To address the first challenge, we propose an LLM-assisted procedure to gather input semantics beyond type constraints for an AP framework's APIs. A semantic requirement defines an input condition on the shapes of ndarrays and the values of non-ndarray parameters. Then, we propose an input-requirement-aware API call generator (IRA-ACG) to generate API calls with inputs satisfying the collected requirements for a test. Its general process is (1) pick an array API to call; (2) attempt to select an existing ndarray from the test that satisfies the requirements of the ndarray-type parameter, and generate a valid concrete value for the scalar-type parameter; (3) and infer the output ndarray's shape and save it as the input candidate for future call generation. The shape inference is based on the API semantics and does not require dynamic execution. Hence, IRA-ACG can be executed multiple times continuously to generate complex API call sequences without runtime information.
API call sequence generation: To overcome the second challenge, we employ a search algorithm (e.g., feedback-directed random algorithm [69] and genetic algorithm [70]) to generate tests for difference detection. It generates tests by repeatedly invoking IRA-ACG to ensure input validity. Our tests consist of an API call sequence: The output ndarrays of an API call are used as inputs for subsequent calls, thereby overcoming the limitation that unit testing only tests a single API.

1.2 Motivating Example

Fig. 3 presents a formerly unknown NumPy bug detected when applying ArrayDiff to the NumPy-CuPy pair. Fig. 3a shows that NumPy claims that `diagflat` returns a 2-D array. Line 3 behaves correctly with the constant input "1.". However, an ArrayDiff's test passes the `var`'s output "1." to `diagflat`. Its return violates the contract that `diagflat`'s return is 2-D. This bug can cause programs that depend on this contract to crash or run abnormally. The root cause is that `diagflat` calls a function `__array_wrap__`. It behaves normally on the constant input "1." but mishandles an internal property (`PY_TYPE`) of `var`'s output. We use two test generators to generate tests for `diagflat`, and present them in Fig. 3b: Pynguin [51] is the state-of-the-art search-based test generator, and Ghostwriter [53] (a property-based test generator [31]) ranked first in NumPy code coverage in a recent competition [23] while Pynguin ranked third. Pynguin's test calls `diagflat` with a `None` value (Lines 2-3). The input is invalid, not to mention triggering the bug. Ghostwriter takes NumPy's type constraints into account and uses `@given` annotation to describe them, so that the test is parameterized and can be executed with a random input engine (Lines 5-7). However, this bug is undetected since the input is not created via APIs like `var`.

ArrayDiff generates a valid (non-empty) ndarray input for `var` based on IRA-ACG and creates the call sequence from Lines 6-7 by passing `var`'s output to `diagflat`, thus detecting this bug.

1.3 Contributions

Our contributions are summarized as follows:

•We collect input semantic requirements for 216 NumPy APIs and propose an LLM-assisted procedure to transfer them to other AP frameworks, enabling ArrayDiff's general applicability. This procedure successfully transfers NumPy's requirements to 249 Octave APIs using five open-weighted models.
• We propose the input-requirement-aware API call generator (IRA-ACG) to generate API calls with inputs that satisfy these semantic requirements.
•We employ search algorithms (e.g., feedback-directed random [69] and genetic [70] algorithms) to iteratively generate test sequences via the IRA-ACG for difference detection.
•We evaluated ArrayDiff (variants with/without semantic knowledge) and the baselines including TitanFuzz [16] and Ghostwriter [53] over 5 runs (10h budget) on five AP pairs across NumPy, CuPy, and Octave. Ghostwriter's NumPy-only support limits its use to NumPy pairs.

For all pairs, ArrayDiff variants with semantic knowledge detect 46 (average 29 in the first hour) valid-input differences and 20 invalid ones. The counterpart numbers are 43 (9 in the first hour) and 38, confirming that IRA-ACG enhances it in effectiveness and efficiency of detecting valid-input differences, which accounts for 19 out of 21 confirmed bugs. TitanFuzz detects 26 valid (3 missed by ArrayDiff) and 40 invalid differences (15 missed). Its performance mirrors non-semantic-knowledge ArrayDiff owing to their comparable input validity (47% vs. 49%), whereas semantic ArrayDiff reaches 84%. Ghostwriter's single detection confirms the value of API sequences. Valid differences mainly stem from bugs, API changes, and numerical errors.

The paper is structured as follows. Section 2 reviews related work. Section 3 introduces ArrayDiff. Section 4 presents the evaluation. Section 5 discusses the experimental setting and validity threats. Section 6 concludes the paper. Section 7 provides our artifact.

2 Related Work

We introduce related studies on (1) Array programming, (2) Differential testing, and (3) Feedback-directed random software testing and Search-based software testing.

Array programming. Array programming (AP) provides a concise and expressive syntax for operating multidimensional arrays [37]. The famous frameworks include NumPy [62], Matlab [56], and Octave [64] (Matlab's open-source alternative). They underpin nearly all computing fields [37], such as deep learning [42, 50, 72], quantum computing [41, 43], and data science [9, 55, 74, 75, 77, 82, 90]. Existing studies on array programming mainly focus on performance improvement [6, 7, 29, 46, 65, 96, 98] and language design [4, 5, 73]. Our work focuses on detecting functionality differences between array programming framework implementations (e.g., two versions). The closest study is AutoMR [95]. AutoMR automatically infers mathematical metamorphic relations (MRs, e.g., $\texttt{sin(x)=sin(x+2}\pi\texttt{)}$) to test 29 NumPy's numerical functions. Each MR is related to a single API, and AutoMR uses them to test APIs individually, ignoring the bugs caused by API sequences. In contrast, ArrayDiff generates API sequences for differential testing.

Differential testing. Our work applies differential testing, which addresses the challenge of test assertion generation [27, 33, 57]. General automated testing techniques often lack domain knowledge of specific software, making it difficult to verify whether test results meet expectations. Differential testing runs the same tests on two software versions and compares their results as the test assertion. It has been widely used in software testing. Many studies focus on differentially testing the deep learning (DL) libraries, which, like AP frameworks, are also built around multi-dimensional arrays (tensors [1]). They can be categorized into API-level testers [17, 45, 68, 89, 93] and sequence-level testers [16, 35, 47, 48, 85, 88] like ArrayDiff. Most of these sequence-level testers [35, 47, 48, 85, 88] synthesize DL models to test DL libraries. However, the DL models are the API sequences with limited patterns [16], missing diverse sequences that cause bugs. ArrayDiff can generate arbitrary API sequences without such structural constraints. TitanFuzz [16] also generates

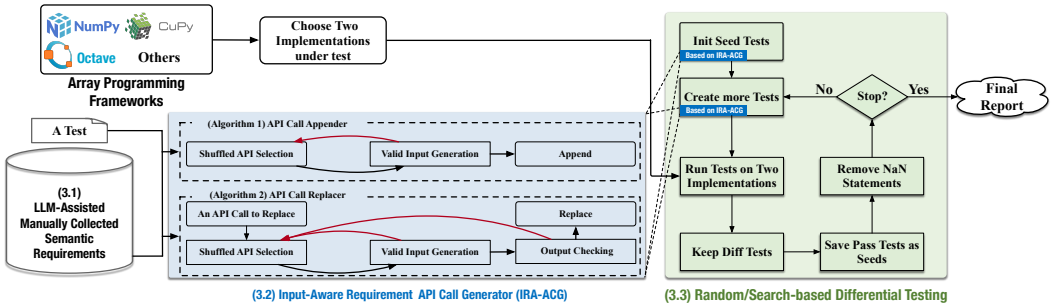


Fig. 4. Overview of ArrayDiff.

arbitrary API sequences via leveraging LLMs to insert API calls. However, LLMs may generate invalid inputs. Our evaluation shows that only about 49% of the generated API sequences are valid. In contrast, ArrayDiff considers input requirements when inserting API calls. Furthermore, ArrayDiff leverages LLMs to transfer manually collected API requirements from NumPy to other AP frameworks, enabling its general applicability. ArrayDiff achieves a valid test rate of roughly 80%, both for NumPy and Octave. ArrayDiff detects nearly twice as many valid-input differences (the main source of confirmed bugs) as TitanFuzz (see Section 4.2). Furthermore, LLM-based testing has widely used in various areas, such as GUI [49], language processors [22, 87], and embedded devices [66]. Refer to surveys [84] for further studies.

Feedback-directed random and search-based software testing. Feedback-directed random software testing (RST) [69] and Search-based software testing (SBST) [80] are two widely used techniques in generating tests. RST randomly inserts statements into tests and checks whether they pass or fail (trigger an exception). A test is used for further insertion if it passes. Otherwise, it is saved as a bug trigger. SBST formulates the test goal (e.g., code coverage) as fitness functions to guide genetic algorithms in finding solutions. A typical SBST procedure involves a generator generating random solutions (e.g., tests). Then, SBST selects some of them based on feedback from fitness functions as the next generation. Next, it mutates them, creating more solutions. The routine stops when the optimum is reached or the time budget is exhausted. Existing research on RST and SBST mainly focuses on applications [3, 19, 36, 39, 40, 51, 54, 86], search algorithms [26, 32, 70, 71, 94] and test understandability [12, 14, 79]. Our work uses RST and SBST to differentially test NumPy implementations. Our contribution is the proposal of an input-requirement-aware call generator that enables the algorithms to generate valid inputs.

Grammar-based Fuzzing. Grammar-based fuzzers [21, 30, 44, 83] leverage language grammars (e.g., context-free grammars) to produce structured inputs such as XML and Python. ArrayDiff also generates grammatically valid programs. Besides, ArrayDiff goes beyond the language grammar. It emphasizes the semantic input requirements. For example, `dot([1,2], [1])` satisfies Octave grammar, but `dot` raises an error "sizes of X and Y must match". Thus, a grammar formalism alone cannot replace our LLM-assisted requirement-collection procedure.

3 ArrayDiff

In this section, we present ArrayDiff. Fig. 4 shows its overview. ArrayDiff accepts two array programming framework implementations, such as NumPy or Octave across different versions. The general process of ArrayDiff is running a search algorithm (e.g., a random or genetic algorithm) to conduct differential testing. To effectively detect differences, the search algorithm must produce valid inputs (e.g., the $(m \times n)(n \times p)$ compatibility for `matmul`'s two operands) to trigger diverse

program behavior. To achieve this, we first manually collect input requirements for NumPy APIs. Next, we leverage LLMs to transfer NumPy's requirements to other array programming frameworks by identifying equivalent APIs. The requirements on ndarrays all relate to their shape (Section 3.1). Then, we propose an input-requirement-aware API call generator to produce inputs satisfying the requirements, a.k.a., IRA-ACG. It provides two operators for the search algorithm: appending and replacing API calls. The appending operator performs three steps: (1) selects a candidate from shuffled APIs; (2) attempts to synthesize inputs satisfying the collected requirements; and (3) appends a new call if successful or chooses the next API. The replacing operator is similar to the former one, but has an extra check before replacing an existing call with a new one: The output ndarray of the old call may be used as input in other calls. Hence, the new and old outputs should be the same in data type and shape to prevent breaking the input requirements (Section 3.2). With IRA-ACG, ArrayDiff employs a search algorithm: ❶ initialize tests through the appending operator, ❷ create more tests (through appending and replacing operators), ❸ run tests and keep difference-exposing tests, ❹ save tests passing on two implementations for further test creation, and ❺ remove pass tests' statements that yield NaNs (i.e., invalid values) (Section 3.3).

3.1 LLM-Assisted Input Requirement Collecting Procedure

Table 1. Input semantic requirements of APIs under test.

Category	Requirement	Representing APIs
Single NDArray	Non-empty (contains ≥ 1 element)	`min`, `max`, `average`
	Particular shapes	`diag` (1 or 2-D) `vander`, `convolve` (1-D)
Inter NdArrays	All/partial dimensions match	`stack`, `concatenate` `matmul`, `dot`

	Broadcast compatibility [37]:
	arrays have compatible shapes that can be
	auto-expanded to the same shape
		`add`, `equal`, `where`
		`left_shift`
		`logical_and`

Non-NDArray Parameters	Specific value ranges, e.g., Parameter order (ndarray's memory layout): "C", "F", "A", "K" Parameter trim: "b" (from back), "f" (from front), "fb" (both)	`reshape`, `ravel` `trim_zeros`
Inter NDArrays and Non-NDArrays	Parameter axis's value range is [0, ndarray.dimension)	`min`, `std`
	Parameter shape matches element count of ndarray	`reshape`

```
1 from numpy import *
2 v1 = arange(0,4)
3 v2 = reshape(v1, (4, 1))
4 v3 = matmul(v1, v2)
```

Fig. 5. An example showing input requirements.

Generating tests with valid inputs to reach core program states is key to difference detection. We divide the valid input requirements into types and semantics. We leverage Fig. 5 as an example. The type constraints are: (1) `arange` accepts numerical parameters; (2) `reshape` requires an ndarray as its first argument, followed by a tuple specifying a new shape; and (3) `matmul` requires two ndarrays and their dtypes support the multiplication operation. The semantic requirements are: (1) `reshape`'s new shape must preserve element count ($4 = 4 \times 1$); and (2) `matmul` requires the columns of the first operand match the rows of the second (`(1,4)` and `(4,1)`). It is feasible to automatically infer type constraints since the documents of the AP frameworks equip each API with structured comments to describe types [63]. However, the semantic requirements are described in natural language and are hard to infer automatically. To address this challenge, TitanFuzz [16] leverages LLMs to generate inputs, while its evaluation shows that about 60% of the generated tests are invalid. Deng et al. [15] show that even state-of-the-art LLMs (e.g., GPT-4-Turbo) struggle with these implicit constraints. However, the APIs of AP frameworks are stable. For example, from Aug 2024 to Jan 2025, only one API (`unstack`) was added into NumPy. Therefore, rather than relying solely on LLMs, we adopt an LLM-assisted

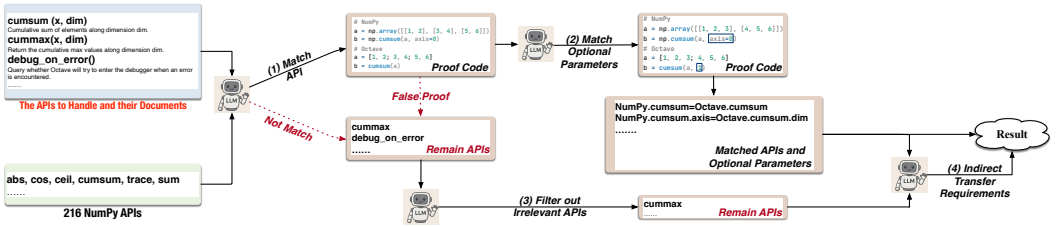


Fig. 6. Transferring semantic requirements from NumPy to other AP frameworks.

process: We manually collect NumPy's requirements and transfer them to other frameworks via LLMs. The input generator (Section 3.2) employs these requirements to filter out the invalid inputs.

To conduct manual analysis on NumPy, we first determine the APIs under test: From 400+ APIs collected by developers [20], we exclude nondeterministic APIs (e.g., random operations) and APIs whose inputs and outputs are not scalar or ndarrays. Finally, the 216 APIs are selected. For each API, we manually inspect the documentation and validate the presence of the requirements by designing and running tests. A requirement is determined if a program can be constructed to trigger an exception when the input violates the requirement. For example, in Fig. 5, the program triggers a *mismatch* exception when replacing `matmul(v1, v2)` with `matmul(v2, v2)`. After the manual study, we collected seven types of semantic requirements for the APIs, shown in Table 1. We categorize them based on the parameters involved: Single NDArray, Inter NdArrays, Non-NDArray Parameters, and Inter NDArrays and Non-NDArray Parameters. The requirements concerning ndarrays exclusively relate to their shape attributes. A shape is a tuple that specifies the size of each dimension. For example, the `min` API returns the minimum value, thus requiring that ndarrays are not empty, i.e., the shape's all elements must be positive. The requirements concerning non-ndarray parameters relate to the values. For example, `trim_zeros` removes leading and trailing zeros from ndarrays. The `trim` parameter controls the truncation direction and has three valid options: "b" (from back), "f" (from front), "fb" (both). Each requirement is formulated as an input filter. For example, Table 1's Non-empty is formulated as `lambda a: prod(a.shape)>0`. These filters are used in the input generator (Section 3.2) to get the valid inputs.

For other AP frameworks, we automatically transfer NumPy's requirements to them, mainly based on LLMs identifying equivalent APIs. Fig. 6 shows the process using Octave as an example: (1) For each API in the target AP framework (called APIT), we provide its documentation and the names and brief descriptions of 216 NumPy APIs to the LLM, and prompt it to identify the equivalent API and generate proof code. Fig. 6 exemplifies the proof code with `cumsum` of NumPy and Octave. If the LLM does not return the equivalent API (APIE) or the proof code fails (i.e., the ndarray values between NumPy and Octave are different), we mark the APIT as the remaining APIs. (2) Otherwise, LLM's correct proof enables us to transfer the mandatory parameters' requirements from APIE to APIT. Then, for APIT's each optional parameter, we ask the LLM to return which APIE's parameter is equivalent and the proof code. Fig. 6 shows that the LLMs prove that for `cumsum`, NumPy's `axis` (0-based indexing) and Octave's `dim` (1-based) are equivalent. As a result, we transfer the requirement of `cumsum` of NumPy related to the ndarray and `axis` parameters to Octave, i.e, `axis` in $[0, len(\text{shape}))$ is transferred to `dim` in $[1, len(\text{shape})]$. (3) For each remaining API (e.g., `cummax` and `debug_on_error` in Fig. 6), we ask the LLM to determine whether it is irrelevant to array programming (e.g., debugging and logging functions), and we filter it (e.g., `debug_on_error`) out if the response is yes. (4) For each relevant API (e.g., `cummax`), we identify those Octave APIs that share identical input and output parameter names and are matched with NumPy APIs (e.g.,

Algorithm 1: Appending an API call to a test

```
1  Algorithm append(test)
2      APIs ← shuffle(ArrayAPIs)
3      foreach api ∈ APIs do
4          ok, inputs ← genInput(test, api)
5          if !ok then
6              continue
7          end
8          ndarray ← api.getReturnNDArray(inputs)
               // unique variable name, dtype, shape
9          test.ndarrays.append(ndarray)
10         test.appendStatement(ndarray, api, inputs)
11         return
12     end
```

Algorithm 2: Replacing a test statement's API call

```
1  Algorithm replace(test, stmt)
2      APIs ← shuffle(ArrayAPIs)
3      foreach api ∈ APIs do
4          ok, inputs ← genInput(test.preStmts(stmt), api)
5          if !ok then
6              continue
7          end
8          ndarray ← api.getReturnNDArray(inputs)
9          if !sameDTypeAndShape(ndarray, stmt.lhs) then
10             continue
11         end
12         stmt.replaceAPICall(api, inputs)
13         return
14     end
```

cumsum). Then, we pass them to LLM for choosing one with the same input requirements as the target API. Therefore, we indirectly extract cummax's requirements. Our experiment (Section 4.1) evaluates five models, and they transfer requirements for 249 Octave APIs.

3.2 Input-Requirement-Aware API Call Generator (IRA-ACG)

We propose our input-requirement-aware API call generator (IRA-ACG) to guarantee that the inputs satisfy the collected requirements. It provides two operations for the test evolution of search algorithms to conduct differential testing (Section 3.3): appending an API call to a test and replacing an existing one. Their general workflow is: (1) pick one from shuffled APIs; (2) synthesize inputs satisfying the requirements; and (3) append or replace a call if successful, otherwise go back to (1). Our evaluation (Section 4.2) shows it significantly improves ArrayDiff's valid input generation.

Algorithm 3: Generating inputs for an API call

```
1  Algorithm genInput(test, api)
2      inputs ← [ ]
3      foreach p ∈ api.parameters do
4          if p.optional ∧ random(0, 1) < 0.5 then
5              continue
6          end
7          if p.type = NDArray then
8              chosen ← dtypeFilter(api, inputs, test.ndarrays)
9              chosen ← semanticFilter(api, inputs, chosen)
10         else
11             chosen ← semanticPickValues(api, inputs)
12         end
13         if |chosen| = 0 then
14             return False, None
15         end
16         inputs.append(randomPick(chosen))
17     end
18     return True, inputs
```

Algorithm 1 shows the appending procedure. It loops to select a candidate from shuffled APIs (Line 3) and applies genInput to it (Line 4): genInput (Algorithm 3) iteratively generates a concrete value for each parameter. Lines 4-6 probabilistically skip optional parameters. For ndarray parameters (Lines 8-9), ndarrays are first chosen by dtype compatibility by the target API's requirements and determined inputs, followed by semantic constraints checking through our LLM-assisted collected rules. For non-ndarray parameters, which include scalar types (integer, float, boolean, and char) and Python data structures (string, list, and tuple) containing scalar elements, Line 11 chooses the candidates from semantically-specified ranges when available; otherwise, the type-determined global range is used. As a result, the filtered candidates satisfy the type and semantic requirements. We leverage choosing two ndarrays (a1 and a2) for matmul as an example. The only requirement for $a1$'s candidate inputs is that the dtype supports multiplication. After a1 is resolved, dtypeFilter filters ndarrays whose dtype does not support multiplication with a1, and semanticFilter filters ones whose shape is not compatible with a1. The procedure aborts and returns false if no candidate satisfies the requirements (Lines 13-15), and append goes to the next API. After receiving inputs, append's Line 8 forms a new ndarray variable with a unique name, and

Algorithm 4: Random testing

```
1  Algorithm RA(popSize, initMaxLength)
2      currentPop ← InitTests(popSize, initMaxLength) // ❶
3      passTests ← [ ]
4      repeat
5          CreatMore(currentPop, passTests, popSize) // ❷
6          runResult ← RunAndSaveDiff(currentPop) // ❸
7          newPassTests ← runResult.passTests // ❹
8          RemoveNaNStmts(newPassTests, runResult) // ❺
9          passTests ← passTests +newPassTests
10         currentPop ← [ ]
11     until time budget is out
1  Procedure CreatMore(currentPop, passTests, popSize)
2      while |currentPop| < popSize do
3          passTest ← copy(random(passTests))
4          Append(passTest)
5          currentPop ← currentPop + [passTest]
6      end
```

Algorithm 5: Search-based testing

```
1  Algorithm GA(popSize, initMaxLength)
2      currentPop ← InitTests(popSize, initMaxLength) // ❶
3      repeat
4          newPop ← [ ]
5          CreateMore(newPop, currentPop) // ❷
6          runResult ← RunAndSaveDiff(newPop) // ❸
7          currentPop ← SelectTests(runResult, popSize) // ❹
8          RemoveNaNStmts(currentPop, runResult) // ❺
9      until time budget is out
10 Procedure CreateMore(newPop, currentPop)
11     foreach t1, t2 ∈ currentPop do
12         n1, n2 ← Crossover(t1, t2)
13         newPop.add( Mutate(t1), Mutate(t2), Mutate(n1),
             Mutate(n2) )
14     end
15     return newPop
16 Procedure SelectTests(runResult, popSize)
17     fitness ← ComputeFitness(runResult)
18     selected ← Select(fitness, popSize)
19     return selected
```

Algorithm 6: Auxiliary functions

```
1  Procedure InitTests(popSize, initMaxLength)
2      tests ← [ ]
3      for i ∈ [0, popSize) do
4          test ← Test()
5          i, length ← 0, random(0, initMaxLength)
6          for j ∈ [0, length) do
7              append(test) // invoke Algorithm 1
8          end
9          tests.append(test)
10     end
11     return tests
1  Procedure Append(test)
2      for i ∈ [0, random(0, |test|)] do
3          append(test) // invoke Algorithm 1
4      end
1  Procedure Minimize(test, stmt)
2      return getRecursiveDependencies(stmt) + [stmt]
1  Procedure RemoveNaNStmts(tests, runResult)
2      foreach test ∈ tests do
3          stmts ← runResult.getNaNStmts(test)
4          test.RemoveWithRecursiveDependent(stmts)
5      end
6      return test
1  Procedure Modify(test)
2      foreach statement ∈ test do
3          if random(0, 1) < 1/|test| then
4              replace(test, statement) // invoke Algorithm 2
5          end
6      end
1  Procedure Remove(test)
       // remove each statement with probability 1/|test|
1  Procedure Mutate(test)
2      operators ← [Remove, Modify, Append]
3      foreach operator ∈ operators do
4          if random(0, 1) < 1/3 then
5              operator(test)
6          end
7      end
8      return test
```

its dtype and shape. Note that (1) the dtype and shape are inferred based on the API's semantics and do not require dynamic execution. For example, `matmul` on two ndarrays with (1,2) and (2,1) shapes always generates one with (1,1) shape; and (2) some APIs may produce shape-undeterministic outputs. For example, `trim_zeros` trims the input' zeros as the output. Hence, the output's shape depends on the input's element values. For these APIs, we mark the output ndarrays as *ShapeUnknown*, and they are always filtered by `semanticFilter`. Lines 9-10 add the novel ndarray into the test's ndarrays for being an input candidate in the future, and append the complete API call statement with resolved inputs and the returned ndarray into the test. This procedure can append a statement to an empty test. The creation APIs (e.g., `array` and `arange`) have no ndarrays as inputs and no semantic requirements, so inputs can always be generated.

Algorithm 2 shows the replacement. It tries to replace a statement's API call with a new one, and the old and new calls return two ndarrays with the same dtype and shape. Like Algorithm 1, it depends on `genInput` (Line 4). The difference is that we use `preStmts` to select statements before the chosen one. Therefore, the ndarrays defined after it will not be input candidates. Then, Lines 8-11 get the new ndarray and continue if it differs from the old one in dtype and shape. Otherwise, Line 12 replaces *stmt*'s old call with the new one.

3.3 Random/Search-based Differential Testing

We present a differential testing approach that employs a search algorithm to iteratively invoke IRA-ACG (Section 3.2) to evolve tests (API sequences) for difference detection. Unlike prior work that generates API sequences with patterns (e.g., deep learning models) [35, 47, 48, 85, 88], our algorithm can produce arbitrary sequences. Moreover, the inputs generated by IRA-ACG satisfy the semantic requirements collected in Section 3.1. Our evaluation (Section 4.2) shows that about 80% of tests are valid, while the counterpart number of TitanFuzz [16] (an LLM-InputGen approach to create arbitrary sequences) in its experiment is nearly 40%. ArrayDiff is theoretically compatible with any search algorithm. We choose two mainstream algorithms in software testing: a feedback-directed random algorithm (RA) [69] and a genetic algorithm (GA) [70].

3.3.1 Feedback-directed Random Algorithm. We refer to previous studies [52, 69] to implement our random algorithm (RA). The overall procedure is shown in Algorithm 4. Each step is as follows:
❶ **Initialize Tests:** A group of tests is generated by Algorithm 6's `InitTests` (Line 1): loop to create a random-length test by iteratively invoking IRA-ACG's appending operation until tests reach the population size. These tests are later used to generate more tests. Then, we initialize *passTests* as empty to store tests that pass on both AP implementations (Line 2).
❷ **Create More Tests:** RA uses `CreatMore` to create tests until *currentPop*'s size reaches *popSize* (Line 5). `CreatMore` randomly chooses a *passTest*, applies Algorithm 6's `Append` into it, and adds it to *currentPop*. `Append` first selects a random value β from 0 to l. Then, it repeats β times to apply IRA-ACG's appending operator to the test.
❸ **Run Tests and Keep Diff Tests:** We run the generated tests on two AP implementations (Line 6). For a test's two executions, we check whether they exit abnormally (i.e., trigger an exception and crash). If both exit abnormally, we regard this test as invalid and remove it in evolution since evolving on it will likely keep it invalid, thus making it useless in difference detection. If only one exits abnormally, we regard the test as detecting a difference. Otherwise, we compare the runtime values of all ndarrays. If they differ, the test is also a detector. Note that float-value ndarrays from two executions may have slight errors. To tolerate them, we adopt NumPy's testing practice [61] and use the following formula to compare two float values:

$$|a - b| < \max(1, |b|) \times 1^{-7}, \tag{1}$$

where max ensures that the tolerance keeps 1^{-7} when $|b| < 1$. Finally, we keep a detector in the final report and remove it in evolution to reduce redundant detection. Meanwhile, we apply Algorithm 6's `Minimize` to the detector and the difference-inducing statement to get the minimized version and report it too. `Minimize` relies on the assumption that the observed difference originates from the difference-inducing statement or its dependencies. This assumption does not hold if the AP framework enters an incorrect internal state due to other statements. However, this case does not arise in our evaluation. For the tests that pass on two implementations, we record their runtime information into *runResult*, including the ndarray values and coverage.
❹ **Save Pass Tests as Seeds:** We obtain all tests passing on two implementations from *runResult* (Line 7). It will be added to *passTests* (Line 9) for further test creation.
❺ **Remove NaN Statements:** IRA-ACG ensures that input ndarrays meet type and shape requirements, but their values may still be invalid. For example, it may pass negative values to `sqrt`, yielding NaN (not-a-number) values to denote invalid inputs. These NaN values propagate when using them as inputs [48], obstructing effective comparison of AP frameworks' core behaviors. Hence, Line 8 removes statements returning NaN values from *passTests*. Deleting a statement causes the statements using its defined variable to be invalid. Hence, we recursively delete them.

After the last step, we update *passTests* and clear *currentPop*. Then, the RA repeats the routine if the budget remains. Otherwise, it conveys detected differences in the final report.

3.3.2 Genetic Algorithm. We refer to previous studies [26, 70] to implement our genetic algorithm (GA). The procedure (Algorithm 5) is similar to the RA. The main differences are as follows:

❷ **Create More Tests:** Different from the RA that creates tests only via appending statements to the *passTests*, the GA employs two GA operators to generate tests (Algorithm 5's `CreateMore`):

Crossover chooses two parents and exchanges parts of their copies to produce two children. The general procedure is: Copy two tests (t_1 and t_2) and select a random value α in $(0, 1)$. Then, Swap the last $\alpha|t_1|$ statements ($s1$) of t_1 and the last $(1-\alpha)|t_2|$ statements ($s2$) of t_2. However, these statements may use the ndarrays defined by previous statements, and swapping without these dependencies causes the new tests to be illegal. Hence, we add the dependencies to $s1/s2$ too. Finally, we remove s_1 from $t1$ and add $s2$, and vice versa.

Mutate (shown in Algorithm 6's `Mutate`) applies `Remove`, `Modify`, and `Append` into a test with probability 1/3. Hence, only one sub-operator is applied on average. Assuming the test's length is l, `Remove` deletes each statement with probability $1/l$. We recursively delete the statements using the variable defined in the deleted ones; `Modify` applies IRA-ACG's replacing operator (Algorithm 2) into each statement with probability $1/l$; `Append` is already described in the RA. It applies IRA-ACG's appending operator (Algorithm 1) to the test.

❹ **Save Pass Tests as Seeds:** Different from the RA, which saves all tests that pass on two implementations for further test creation, the GA requires fitness functions to judge tests for keeping elites from them in further evolution. A fitness function denotes whether a code element (e.g., a statement or method) is covered. We maintain a binary fitness vector for each passing test, where elements correspond to statements, with 1 indicating covered and 0 indicating uncovered. Specifically, our NumPy/CuPy fitness functions correspond to statement coverage [78], where each fitness value reflects the coverage of a specific statement. For Octave, we instead adopt method coverage [78], in which each fitness value corresponds to a method, due to the lack of suitable statement coverage tools. After computing the fitness vectors for all tests, we select a subset of tests that outperform others in fitness scores (i.e., the elites) for further evolution via MOSA [70]. MOSA and its variant [70, 71], based on Pareto dominance [13], are the state-of-the-art multiple-objective sorting algorithms in search-based test generation [8, 23]. Pareto dominance defines a dominance relation for two fitness vectors of length n: v_1 dominates ($\succ$) v_2 if:

$$(\forall i \in \{1, ..., n\} : v_1[i] \geq v_2[i]) \wedge (\exists i \in \{1, ..., n\} : v_1[i] > v_2[i]). \tag{2}$$

MOSA sorts vectors into m groups called Pareto fronts [13] that satisfy (1) no dominance relation in a Pareto front, i.e.,

$$\forall i \in \{1, ..., m\} : \forall j, k \in \{1, ..., |F_i|\} : \ (F_i[j] \not\succ F_i[k]) \wedge (F_i[k] \not\succ F_i[j]), \tag{3}$$

(2) and any vector in a latter front is dominated by one in a previous front, i.e.,

$$\forall i, j \in \{1, ..., m\} : i < j \implies (\forall k \in \{1, ..., |F_j|\}, \exists m \in \{1, ..., |F_i|\} : F_i[m] \succ F_j[k]). \tag{4}$$

Then, MOSA keeps tests in evolution by their vectors from F_1 to F_m until the kept tests reach the maximum. These tests form a novel population. Then, the GA repeats until the budget is exhausted.

4 Evaluation

Our evaluation concentrates on ArrayDiff's detection performance and answers the following research questions:

RQ1: (Effectiveness of LLM-assisted requirement transfer approach) How effectively does the LLM-assisted approach transfer NumPy API requirements to other AP frameworks (Section 4.1)?

Table 2. Requirement transfer results of five models.

Model	Directly Matched APIs	Matched Optional Parameters	Non-Array APIs	Indirectly Matched APIs	Whole Matched APIs
	#correctness (Acc.)	#correctness (Acc.)	#correctness (Acc.)	#correctness (Acc.)	#correctness (Acc. | Recall)
DeepSeek-R1 (671B MoE [34])	126 (0.992)	47 (0.979)	855 (0.993)	32 (0.800)	158 (0.946 | 0.635)
DeepSeek-V3 (671B MoE)	111 (1.000)	38 (1.000)	778 (0.997)	75 (0.670)	186 (0.834 | 0.747)
Llama4-Maverick (400B MoE)	107 (0.982)	9 (1.000)	762 (0.996)	64 (0.621)	171 (0.807 | 0.687)
Qwen3-32B (Dense)	128 (0.992)	47 (1.000)	920 (0.980)	53 (0.779)	181 (0.919 | 0.727)
Qwen3-14B (Dense)	106 (0.981)	27 (1.000)	864 (0.969)	36 (0.750)	142 (0.910 | 0.570)

RQ2: (Effectiveness of ArrayDiff) How effectively doesArrayDiff detect AP implementation differences (Section 4.2)?
RQ3: (Root Causes of Differences) What are the root causes of differences (Section 4.3)?

4.1 RQ1: Effectiveness of LLM-assisted Requirement Transfer Approach

We propose an LLM-assisted approach for transferring manually collected input requirements from NumPy to other AP frameworks (Section 3.1). This RQ investigates its effectiveness.
•**Subjects:** We select Octave [64] as the target AP framework, extracting 1,546 APIs along with their functional and parameter descriptions from its structured documentation.
•**LLMs:** We evaluate five popular open-weighted LLMs (details shown in Table 2). The parameter number of these models ranges from 14B to 671B.
•**Evaluation Metrics:** We judge the results by running proof code and checking results for direct API and optional parameter matching. For non-array APIs, we judge the results by reviewing the documents. For indirect transfer requirements, we write tests to verify whether the target API shares the same input requirements as the API selected by LLMs. Then, we count two metrics: the number of correct cases and the accuracy (#correctness/#total) of (1) target APIs directly matched to NumPy APIs, (2) matched optional parameters, (3) non-array APIs, and (4) indirectly matched APIs. We count all models' whole matched APIs (direct/indirect) and compute each's total correct APIs, accuracy, and recall (#correctness/#whole). Previous items' recalls are not computed, since an API may be directly matched by a model and indirectly by another.
•**Results:** Table 2 presents the requirement transfer results for five models. Among these, Qwen3-32B achieves the highest matched API correctness (128) and non-array APIs (920), Qwen3-32B and DeepSeek-R1 attain the highest matched parameter correctness (47), and DeepSeek-V3 reaches the highest indirect match correctness (75). In terms of accuracy, DeepSeek-R1 leads in indirect match accuracy and non-array APIs, while Qwen3-32B dominates direct API and parameter accuracy. We summarize these results, and the whole matched API count is 249, and the non-array API count is 979. We manually examine the 316 remaining APIs that are neither non-array nor matched, mostly involving special types (e.g., first-class function) or advanced APIs (e.g., differential equation solvers) that are not supported by our collected 216 NumPy APIs. These advanced APIs are applications of AP rather than its core (creations, manipulations, and basic math computations) and are thus beyond the scope of this paper. Testing them requires manual integration into ArrayDiff.
•**Analysis:** Even the smallest model, Qwen3-14B, retrieves over half of the matched APIs, indicating that our approach is effective across model scales. Although DeepSeek-V3 shows lower performance in matching APIs between NumPy and Octave, it excels at indirect matching. For instance, while failing to directly match `var` (failing to generate proof code), it correctly directly matches `mean` and

Table 3. AP framework implementation pairs under test.

A	B	Description
CuPy-13.3.0 (latest) on NVIDIA GPU	NumPy-2.2.2 (latest) on AMD64	NumPy and NumPy-compatible library
CuPy-13.3.0 (latest) on NVIDIA GPU	NumPy-2.1.0 on AMD64 (latest at CuPy-13.3.0 release)	NumPy and NumPy-compatible library
NumPy-2.2.2 (latest) on AMD64	NumPy-1.26.4 on AMD64	Two NumPy versions
NumPy-2.2.2 (latest) on AMD64	NumPy-2.2.2 on ARM64	NumPy under two architectures
Octave-10.2.0 (latest) on AMD64	Octave-9.4.0 on AMD64	Two Octave versions

infers that Octave's `var` and `mean` share the same input requirement (non-empty ndarrays). This demonstrates that our indirect matching strategy enhances LLMs' requirement transfer capability.

Answer to RQ1: Our approach successfully transfers requirements for 249 Octave APIs using five models. Qwen3-32B achieves the highest direct API matches (128) and optional parameter matches (47), while DeepSeek-V3 leads in indirect matches (75).

4.2 RQ2: Effectiveness of ArrayDiff

This RQ aims to investigate ArrayDiff in detecting AP implementation differences.

•**Subjects:** Table 3 lists five AP pairs, where **A** is the latest and **B** is its compatible library, prior major release, or a different architecture. Note that the first pair corresponds to the latest versions of CuPy/NumPy. However, when CuPy-13.3.0 was released in August 2024, NumPy-2.1.0 was the latest. The API changes introduced in NumPy-2.2.2 had not yet been integrated into CuPy. Therefore, we additionally include CuPy-13.3.0 vs. NumPy-2.1.0 as a more suitable pair for differential testing. Section 5.1 justifies excluding NumPy/CuPy vs. Octave.

•**Evaluated Approaches:** (1) ArrayDiff and the ablated variants: We integrate a random algorithm (RA) and a genetic algorithm (GA) with ArrayDiff (Section 3.3). These two algorithms utilize our IRA-ACG (Section 3.2) to generate tests with inputs that satisfy the input requirements (Table 1) and dynamically remove the statements that yield NaN values. To evaluate ArrayDiff and each component's contribution, we propose six ArrayDiff variants:

RA+Semantic+NaN (RSN): The RA with semantic knowledge and the NAN statement remover.
RA+Semantic (RS): The RA with semantic knowledge and without the NAN statement remover.
RA: The RA without semantic knowledge and the NAN statement remover. It only guarantees to generate inputs that satisfy the type constraints.
GA+Semantic+NaN (GSN): The GA with semantic knowledge and the NAN statement remover.
GA+Semantic (GS): The GA with semantic knowledge and without the NAN statement remover.
GA: The GA without semantic knowledge and the NAN statement remover.

(2) Baselines: To validate IRA-ACG and API sequence generation, we use TitanFuzz (T) [16] and Python unit test generator Ghostwriter (GW) [53] as baselines. Unlike our approach, which extracts requirements via LLMs for IRA-ACG (Section 3.2), TitanFuzz generates sequences directly using LLMs. We extend TitanFuzz's support beyond PyTorch and TensorFlow to include NumPy, CuPy, and Octave. This requires minimal effort as TitanFuzz leverages LLMs for code generation, requiring only minor modifications to the prompts and execution environment. Unlike ArrayDiff, Ghostwriter generates unit tests at the API level. It achieved the highest NumPy coverage in [23] by considering NumPy-specific types and semantic constraints (e.g., broadcasting [37] in Table 1). Ghostwriter is excluded from NumPy–CuPy and Octave evaluations. For CuPy, migrating its NumPy-generated tests requires significant manual effort (e.g., converting ndarrays). For Octave, Ghostwriter lacks support, and to our knowledge, no unit test generator exists.

Table 4. Result of effectiveness (Differences), efficiency (AUC), validity, and difference-trigger test (Δ Test) length. Values are reported as Median (IQR); *Italics* indicate the best median.

(a) CuPy 13.3.0 versus NumPy 2.2.2

Method	Diff. (Valid | Invalid)	AUC (Valid | Invalid)	Δ Len (Origin | Mini.)	Validity
RSN	32 (32-32) | 15 (14-15)	82 (78-82) | 35 (33-37)	135 (134-136) | 8 (8-8)	86% (85%-86%)
RS	*33 (33-33)* | 15 (14-15)	*83 (82-85)* | 35 (34-36)	131 (129-132) | 8 (8-8)	85% (85%-85%)
RA	28 (27-28) | *30 (29-30)*	64 (62-64) | *70 (68-70)*	*19 (19-19)* | 6 (6-6)	33% (33%-34%)
GSN	27 (27-28) | 13 (12-14)	71 (70-72) | 33 (30-33)	430 (369-532) | 6 (6-6)	*90% (89%-91%)*
GS	28 (27-28) | 13 (12-13)	73 (73-75) | 33 (30-33)	336 (312-393) | 6 (6-6)	*90% (88%-90%)*
GA	18 (18-19) | 21 (20-22)	43 (43-46) | 53 (46-55)	62 (45-65) | *5 (5-5)*	63% (62%-63%)
T	14 (14-15) | 29 (27-29)	- | -	34 (33-37) | -	43% (42%-43%)

(b) CuPy 13.3.0 versus NumPy 2.1.0

Method	Diff. (Valid | Invalid)	AUC (Valid | Invalid)	Δ Len (Origin | Mini.)	Validity
RSN	*31 (30-32)* | 13 (12-13)	81 (79-84) | 33 (31-34)	136 (135-136) | 8 (8-8)	86% (85%-86%)
RS	*31 (30-31)* | 14 (13-14)	*82 (78-82)* | 36 (33-36)	128 (126-129) | 8 (8-8)	85% (85%-85%)
RA	26 (24-26) | *28 (27-28)*	64 (61-65) | *69 (66-70)*	*19 (18-19)* | 6 (6-6)	33% (32%-34%)
GSN	27 (27-27) | 12 (11-12)	71 (70-72) | 30 (28-31)	275 (219-323) | 6 (6-6)	*95% (93%-95%)*
GS	27 (25-27) | 12 (11-12)	71 (67-71) | 30 (29-31)	231 (182-240) | 6 (6-6)	90% (89%-93%)
GA	17 (16-18) | 21 (20-21)	42 (42-43) | 50 (49-53)	23 (23-58) | *5 (5-5)*	63% (63%-63%)
T	14 (14-15) | 26 (25-27)	- | -	34 (32-36) | -	42% (39%-43%)

(c) NumPy-2.2.2 versus NumPy-1.26.4

Method	Diff. (Valid | Invalid)	AUC (Valid | Invalid)	Δ Len (Origin | Mini.)	Validity
RSN	*7 (7-7)* | 0 (0-0)	68 (68-69) | 0 (0-0)	220 (218-222) | 9 (8-9)	85% (85%-85%)
RS	*7 (7-7)* | 0 (0-0)	69 (68-69) | 0 (0-0)	207 (207-216) | 8 (8-8)	83% (83%-83%)
RA	6 (6-7) | 4 (4-4)	57 (55-64) | *64 (64-64)*	*19 (19-19)* | *6 (6-6)*	34% (33%-34%)
GSN	*7 (6-7)* | 0 (0-0)	68 (59-68) | 0 (0-0)	376 (321-666) | 7 (7-8)	*89% (87%-90%)*
GS	*7 (7-7)* | 0 (0-0)	*69 (68-69)* | 0 (0-0)	355 (255-405) | 7 (7-7)	87% (86%-88%)
GA	3 (3-4) | 3 (2-3)	29 (27-36) | 48 (32-49)	28 (24-110) | *6 (5-6)*	61% (61%-61%)
T	6 (5-7) | *10 (10-11)*	- | -	29 (27-35) | -	62% (62%-62%)
GW	1 (1-1) | 0 (0-0)	- | -	- | -	-

(d) NumPy on AMD64 versus NumPy on ARM64

Method	Diff. (Valid | Invalid)	AUC (Valid | Invalid)	Δ Len (Origin | Mini.)	Validity
RSN	*2 (2-2)* | *2 (1-2)*	63 (63-64) | 89 (49-94)	235 (234-271) | 9 (8-9)	83% (83%-84%)
RS	*2 (2-3)* | *2 (1-2)*	*66 (66-96)* | 85 (49-95)	210 (207-219) | 9 (9-10)	82% (82%-82%)
RA	*2 (2-2)* | 1 (1-1)	63 (54-65) | 49 (49-49)	*18 (17-18)* | *5 (5-5)*	35% (34%-35%)
GSN	*2 (1-2)* | 1 (1-2)	65 (33-65) | 47 (47-90)	249 (211-255) | 7 (7-8)	83% (82%-83%)
GS	*2 (2-2)* | *2 (2-2)*	65 (65-66) | *96 (90-97)*	437 (361-440) | 9 (8-11)	*84% (82%-84%)*
GA	1 (0-1) | 1 (1-1)	32 (0-32) | 49 (49-49)	32 (20-148) | *5 (5-6)*	60% (60%-61%)
T	1 (1-1) | *2 (2-3)*	- | -	21 (13-22) | -	53% (51%-53%)
GW	0 (0-0) | 0 (0-0)	- | -	- | -	-

(e) Octave-10.2.0 versus Octave-9.4.0

Method	Diff. (Valid | Invalid)	AUC (Valid | Invalid)	Δ Len (Origin | Mini.)	Validity
RSN	*4 (4-4)* | *1 (0-1)*	97 (96-97) | 42 (0-46)	162 (162-163) | 5 (5-5)	78% (78%-78%)
RS	*4 (4-4)* | *1 (1-1)*	*98 (97-98)* | 40 (40-48)	165 (164-168) | 5 (5-5)	77% (77%-77%)
RA	*4 (3-4)* | *1 (1-1)*	96 (74-97) | *44 (41-47)*	26 (25-26) | 5 (5-6)	34% (33%-34%)
GSN	3 (3-3) | 0 (0-0)	73 (73-74) | 0 (0-0)	282 (214-335) | *4 (4-4)*	*80% (79%-80%)*
GS	3 (2-3) | 0 (0-0)	73 (49-74) | 0 (0-0)	254 (212-415) | *4 (4-4)*	79% (78%-80%)
GA	2 (2-2) | 0 (0-1)	49 (48-49) | 0 (0-49)	84 (74-663) | 5 (5-5)	56% (56%-57%)
T	0 (0-0) | 0 (0-0)	- | -	*0 (0-0)* | -	45% (42%-45%)

Table 5. Pairwise comparison across all pairs under test; Each cell (a, b) denotes the number of pairs where method a (row) significantly outperforms method b (column) ($A_{ab} > 0.5, p < 0.05$).

(a) Valid Differences

	RSN	RS	RA	GSN	GS	GA	T	GW
RSN	–	0	2	3	3	5	3	2
RS	0	–	2	2	3	4	4	2
RA	0	0	–	0	0	5	3	2
GSN	0	0	0	–	0	3	3	2
GS	0	0	0	0	–	4	3	2
GA	0	0	0	0	0	–	2	0
T	0	0	0	0	0	0	–	2
GW	0	0	0	0	0	0	0	–

(b) Invalid Differences

	RSN	RS	RA	GSN	GS	GA	T	GW
RSN	–	0	0	0	0	0	0	1
RS	0	–	0	0	2	0	0	1
RA	3	3	–	4	4	2	1	2
GSN	0	0	0	–	0	0	0	1
GS	0	0	0	0	–	0	0	1
GA	3	3	0	3	3	–	0	2
T	3	3	2	3	3	4	–	2
GW	0	0	0	0	0	0	0	–

(c) Valid AUC

	RSN	RS	RA	GSN	GS	GA
RSN	–	0	2	3	3	5
RS	0	–	4	2	3	5
RA	0	0	–	0	0	5
GSN	0	0	2	–	0	4
GS	0	0	2	1	–	4
GA	0	0	0	0	0	–

(d) Invalid AUC

	RSN	RS	RA	GSN	GS	GA
RSN	–	0	0	2	1	0
RS	1	–	0	1	1	1
RA	3	3	–	4	4	2
GSN	0	0	0	–	0	0
GS	0	0	0	0	–	1
GA	3	3	0	3	3	–

•**Configuration:** For ArrayDiff, the parameters' values of RA and GA are 200 for the population size and 50 for the initial tests' maximum length (see Section 3.3). We follow TitanFuzz's default configuration [16], using InCoder 1.3B [58] for fuzzing with a temperature of 1.0. As the original test seed generator, Codex (code-davinci-002) [67], is deprecated and unavailable, we substitute it with the more advanced GPT-5.1-Codex (2025), which is unlikely to degrade TitanFuzz's effectiveness.
•**Methodology:** For each pair, we run ArrayDiff five times with a 10-hour search budget, since we observe that difference detection converges before reaching the budget. Ghostwriter accepts a NumPy API and deterministically creates tests for it. The tests consist of an API call, an annotation to describe input validity for random engines to generate diverse valid inputs (see Fig. 3b), and assertions. We first use Ghostwriter to generate tests for 216 NumPy APIs supported by ArrayDiff. Then, we repeat five times to use Ghostwriter's default input engine to execute each API's tests on two implementations until a test passes on one implementation and fails on another (i.e., a difference), or the search budget is out. The budget is 10 (hours)/216 $*$ 2 $\approx$ 400 seconds. We double the time to ensure a fairer comparison, because when Ghostwriter's test terminates upon detecting an issue, much of the budget that could be assigned to other tests would be wasted. While TitanFuzz generates sequences like ArrayDiff, it processes APIs independently by generating and fuzzing seeds for each separately. We thus allocate 400 seconds per API for TitanFuzz, matching Ghostwriter. This budget stems from $(10\,\mathrm{h}/216) \times 2 \approx 400$ s (where 216 is the NumPy API count), ensuring 400 seconds is also sufficient for Octave's 249 APIs.
•**Evaluation Metrics:** Each difference of the final reports includes a test and the detection time point relative to starting the execution. We remove duplicates, record the unique differences, and the time first detected. Then, we compute a tool's *effectiveness*, *efficiency* [71], *Validity*, and *Difference-trigger test length*. *Effectiveness* denotes the total of unique differences. We regard exception-related differences as equivalent if they are triggered at the same code line with identical messages. For data-related issues, we identify the triggering API, review the runtime information, and examine the source code to determine duplicates. If duplication cannot be confirmed, we treat the case as unique. *Efficiency* refers to the unique count at various time points. Formally, we regard whole differences as a set D. We construct $P = \{P_1, ..., P_n\}$ for each execution, where $P_i = (c, t)$ is a tuple of the percentage of the detection number to $|D|$ (c) and the time point (t). Then, we quantify efficiency via the Area-Under-Curve (AUC) [71]:

$$AUC(P) = \frac{\sum_{i=1}^{|P|-1}(P_i.c + P_{i+1}.c) \times (P_{i+1}.t - P_i.t)}{2 \times TotalSeachBudget} \times 100. \tag{5}$$

AUC's range is $[0, 100]$. Higher AUC represents better efficiency. Specifically, we compute effectiveness and efficiency in two groups: those with valid inputs and those with invalid inputs. The former are potential bugs that matter to maintainers. The latter are trivial for maintainers but important for

```
1 import numpy_or_cupy as xp
2 a = xp.array([1])
3 a[[0,0]] = [2,3]
4 print(a)
5 # numpy: [3] cupy: vary
```

Fig. 7. Reference difference between NumPy and CuPy.

Table 6. Summary of invalid inputs.

Parameter Type	Reason	Count
ndarray	Invalid shape	28
	Invalid value	12
	Invalid dtype	3
	Parameter miss	1
non-ndarray	Invalid value	9
	Parameter miss	1

migrating developers (e.g., from NumPy to CuPy) to ensure their programs behave normally. After the effectiveness and efficiency are calculated, we use Mann-Whitney U Test and Vargha-Delaney $\hat{A}_{ab}$ [2, 81] to check whether an approach a statistically outperforms another b ($\hat{A}_{ab} > 0.5$ and the significant value p is smaller than 0.05). *Validity* denotes the proportion of valid tests, i.e., those that pass on both AP implementations or are difference-trigger tests with manually confirmed valid inputs. We also report the *difference-trigger test length*, measured both before and after the Minimize procedure (Algorithm 6). Efficiency is not reported for TitanFuzz and Ghostwriter because they utilize independent per-API budgets, unlike ArrayDiff's fixed total budget. Furthermore, since Ghostwriter generates fixed parameterized tests, metrics tailored for sequence generators (i.e., validity and test length) are inapplicable to its evaluation.

•**Effectiveness:** Table 4 reports the effectiveness of the evaluated approaches, using the median and IQR (interquartile range, from the 1st to 3rd quartile) of detected differences. Pairwise Mann-Whitney U test results across all approaches are detailed in Table 5 (a, b). Three key findings emerge: (1) semantic-aware ArrayDiff variants significantly outperform their non-semantic counterparts on valid inputs. Specifically, RA+Semantic+NaN (RSN) and RA+Semantic (RS) are the top performers, with median valid differences of 76 and 77, respectively. Table 5 (a) confirms they significantly surpass other approaches on two to five AP pairs; (2) TitanFuzz (T) and ArrayDiff RA lead in detecting differences on invalid inputs, achieving median totals of 67 and 64, respectively. Table 5 (b) confirms they significantly dominate other approaches on two to four test pairs; (3) ArrayDiff's RA variants outperform the GA variants. ArrayDiff variants collectively identify 47 valid and 39 invalid unique differences. These counts are below the median summary due to substantial redundancy (97%) between the first two CuPy/NumPy pairs. TitanFuzz detects 26 valid differences, of which 3 are not detected by ArrayDiff, and 40 invalid differences, of which 15 are not detected by ArrayDiff. Ghostwriter only detects one difference between two NumPy versions. It is a previously reported crash bug (NumPy-18445) related to a particular ndarray dtype `timedelat64` for date time computation. Two valid differences between two NumPy versions detected by TitanFuzz but not by ArrayDiff are related to `string` and `subarray` dtypes, which are documented API changes [59]. ArrayDiff does not detect them since it currently only involves `int`, `float`, and `bool`. Besides, TitanFuzz reveals a mechanism-level difference between CuPy and NumPy using valid inputs, which ArrayDiff fails to detect: CuPy's `__setitem__` exhibits different behavior from NumPy when integer arrays reference the same memory location multiple times [11]. Fig. 7 provides an example. Line 3 assigns the 0th element of `a` twice. NumPy stores the value from the last assignment, resulting in `a` being "`[3]`". Such behavior is undefined in CuPy; therefore, `a`'s value may vary. ArrayDiff misses this case because it only generates API calls rather than assignment statements, whereas TitanFuzz's LLM can insert arbitrary statements. ArrayDiff and TitanFuzz detect 54 differences caused by invalid inputs, among which 15 are detected only by TitanFuzz and 14 only by ArrayDiff. After manual review, we categorize them in Table 6. Invalid shape occupies over half, confirming the importance of our semantic requirements. TitanFuzz detects

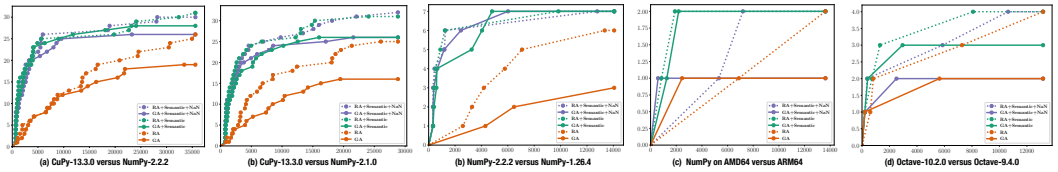


Fig. 8. Differences with valid inputs over time.

two cases with missing required parameters because its LLM can insert arbitrary code. We review the documents and the behavior of the latest version to check whether each difference is addressed. Eleven are solved. Two (CuPy-8807 and CuPy-9110) are confirmed as a NumPy-compatible bug and an illegal-memory-access bug. We list all of them in our artifact.

•**Efficiency:** Table 4 and Table 5 (c, d) present the efficiency results. Fig. 8 illustrates the detection of valid-input differences over time. To minimize random effects, we only report differences identified in at least three out of five runs, using their mean detection time. Under valid inputs, the variants with semantic knowledge outperform the other variants for all pairs. Meanwhile, the semantic-aware variants identify nearly 85% of all valid-input differences included in Fig. 8 within the first hour. For AUC under invalid inputs, RA is the best approach except for NumPy on AMD64/ARM64, where this pair exhibits only two differences, providing insufficient evidence.

•**Validity:** Table 4 shows that, on average of five pairs, the median proportions of valid tests are 84% (RA+Semantic+NaN), 82% (RA+Semantic), 34% (RA), 87% (GA+Semantic+NaN), 86% (GA +Semantic), 61% (GA), and 49% (TitanFuzz). IRA-ACG improves RA by 48%, and the NAN statement remover improves by 2%. IRA-ACG improves GA by 25%, and the remover improves by 1%. GA variants outperform RA variants in validity, particularly in the GA–RA comparison (61% vs. 34%). The main reason is that GA's Mutate (see Algorithm 6) applies `Remove`, `Modify`, and `Append` into a test with probability 1/3. `Modify` and `Append` may introduce invalidity but `Remove` does not. Meanwhile, the probability of at least one of `Modify` and `Append` applied in an iteration is $1-(1-1/3)\times(1-1/3) = 5/9$, i.e., the probability of introducing invalidity is 5/9. RA always applies `Append` to test in an iteration, i.e., RA theoretically generates $1/(5/9) \approx 1.8$ times more invalid tests than GA, which is consistent with our experimental data ($(1-0.34)/(1-0.61) \approx 1.69$). TitanFuzz's validity is nearly half of ArrayDiff's variants with semantic knowledge, showing that its LLM (InCoder 1.3B model [58]) for fuzzing has limited capability in generating valid NumPy/CuPy and Octave programs, likely due to its relatively small model size. Although more advanced LLMs (e.g., GPT-5 and Gemini) could improve validity, the large number of invocations during fuzzing would consume a substantial number of tokens and incur high financial costs.

•**Difference-trigger test length:** Table 4 shows that, averaged over all pairs, the median original test length of ArrayDiff's variants with semantic knowledge is 248, whereas the mean length without semantic knowledge is 33. This gap arises because, without semantic knowledge, ArrayDiff cannot generate long and valid tests, which in turn impairs its ability to detect differences. TitanFuzz's average validity and test length are 0.49 and 24, respectively. These values are close to ArrayDiff without semantic knowledge, supporting our argument. The average of all ArrayDiff variants' median minimized test lengths is 6, confirming the effectiveness of our Minimize procedure and its significant improvement in test readability.

•**Analysis: Usefulness of IRA-ACG.** We propose the IRA-ACG (Section 3.2) to generate API calls with inputs satisfying LLM-assisted collected semantic requirements (Table 1). The experimental results of the five pairs confirm the usefulness of IRA-ACG. Note that IRA-ACG still generates invalid inputs. One reason is that the IRA-ACG's semantic requirements only involve the ndarrays'

shape and data type, not their concrete values. Hence, invalid inputs (e.g., zero for `reciprocal`) could be generated. Second, the implementations vary slightly in validity. We use NumPy-2 as the norm. Hence, others may receive invalid inputs.

Superiority of RA. The RA variants outperform their GA counterparts in detecting differences. The reason is that a coverage criterion, such as statement or method coverage, is insufficient for detecting differences [28, 97]. GA thus favors tests covering more code, but ignores other key factors like specific inputs triggering exceptions. In contrast, RA randomly selects tests, improving test diversity. Fig. 8 shows GA converging earlier than RA, supporting this observation.

TitanFuzz. TitanFuzz detects fewer valid-input differences than semantic-aware ArrayDiff, but more with invalid inputs than ArrayDiff's all variants. For valid inputs, the primary reason is that the InCoder 1.3B model (the LLM used by TitanFuzz for fuzzing) has limited capacity to generate valid programs mainly due to its relatively small parameter size. Using larger models such as GPT-5 may improve TitanFuzz's performance; however, unlike the InCoder 1.3B model, they are difficult to deploy locally in practice and would incur substantial costs from model providers due to massive token consumption during fuzzing, confirming ArrayDiff's resource-efficiency advantage. For invalid inputs, unlike ArrayDiff, which always generates API calls with required parameters (with valid/invalid values), TitanFuzz's LLM produces any statements (e.g., API calls with missing or incorrectly typed parameters). Hence, it reveals more invalid-input differences.

```python
import numpy as np
a1 = np.flipud(np.array([1,2])) # a1: [2, 1]
a2 = np.empty((), dtype="int64")
np.min(a1, out=a2)
assert a2.item() == 1 # 2!=1
```

Fig. 9. NumPy-27820: `min`'s incorrect output.

Ghostwriter. Two reasons cause Ghostwriter's poor performance. First, it misses program behavior, such as not taking the various APIs' outputs as inputs and only using the default values for optional parameters. Second, it produces inadequate assertions: It only checks shape and dtype instead of values. The bug shown in Fig. 9 detected by ArrayDiff exemplifies Ghostwriter's limitations: `min` outputs 2 for $[2, 1]$. Triggering it requires that (1) creating $[2, 1]$ via `flipud` instead of direct creation, (2) using `out` optional parameter to store the result in `a2`, (3) and asserting the `a2`'s value. Ghostwriter satisfies none of these conditions.

Two NumPy-CuPy pairs. We evaluate CuPy-13.3.0 against both NumPy-2.1.0 (CuPy-13.3.0's officially supported version) and 2.2.2 (to match the version used in our other pairs). Findings are 97% identical (67/69 differences), as NumPy's evolution from 2.1.0 to 2.2.2 primarily involved non-breaking bug fixes. A unique finding in 2.1.0 identified a bug (Fig. 9) fixed in 2.2.2 after our report. One invalid-input difference detected only in 2.2.2 also exists in 2.1.0, but was missed. The reason may be due to the algorithm's randomness.

Answer to RQ2: ArrayDiff's RA variants with semantic knowledge detect more valid differences and achieve higher efficiency than other variants and baselines. The NaN statement remover improves test validity but yields no significant improvement in difference detection. Our Minimize procedure produces significantly shorter difference-triggering tests.

4.3 RQ3: Root Causes of Differences

This RQ investigates the root causes of the 50 differences with valid inputs. We exclude the only one difference detected by Ghostwriter since it has been reported and still unfixed (NumPy-18445), and those with invalid inputs, as their root causes originate from input invalidity rather than actual implementation inconsistencies. Specifically, ArrayDiff identifies 47 differences, with 24 being unique to it, whereas TitanFuzz detects 26 cases, including only three exclusive findings.

Table 7. Summary of reviewed differences.

Library	VerifiedBug (Novel | Fixed)	API Changes	False Positive	All
CuPy	15 (14 | 3)	0	8	23
NumPy	4 (4 | 3)	5	1	10
Octave	0	3	1	4
All	19 (18 | 6)	8	10	37

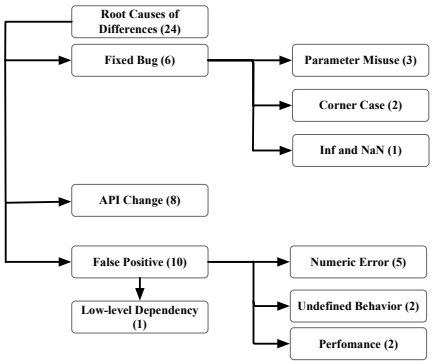


Fig. 10. Root causes of differences.

•**Methodology:** We first verify each difference in the latest version, the documentation, and the report's existence in GitHub. If it is not solved, we report it to GitHub. We mark it with one of the following tags: *VerifiedBug*, *APIChange*, *FalsePositive*, *PendingReview*. Finally, we analyze its root cause if it is a fixed bug, an api change, or a false positive. We require the bug to be fixed to determine its cause.

•**Root Causes:** Table 7 presents the summary of 37 reviewed differences. 19 cases are bugs: one was previously reported, and 6 have been fixed. Hence, we analyze the root causes of 24 differences, including 6 fixed bugs, 8 API changes, and 10 false positives. Fig. 10 presents them:

Fixed Bugs (6). The root cause of the three fixed bugs (CuPy-8782, CuPy-8784, and CuPy-8806) is the misuse of the optional parameters. For example, CuPy-8782 describes that the `nanargmin` API mismatches two parameters when using them as a call's inputs, triggering a crash. Two bugs (NumPy-27783 and NumPy-27820) are due to corner cases, resulting in incorrect results. For example, NumPy-27783 shows that an auxiliary function `__array_wrap__` behaves incorrectly on ndarrays with a particular `PY_TYPE` (a property) value. Third, special floating-point values (Inf and NaN) often trigger differences. Their non-standard handling requirements significantly complicate implementation. NumPy-27902 shows that `dot` varies in NaNs for different-shaped ndarrays. Seven unfixed bugs are also related to Inf and NaN.

API Changes (8). Six differences are due to API changes. NumPy-27969 and NumPy-27971 report two cases where NumPy-2 and CuPy/NumPy-1 vary in the data type of the API's output. They also vary in the output's shape of `linalg.solve`. The evaluation also detects five changes listed in release note of NumPy-2 and Octave-10 (2 by TitanFuzz). For example, Octave's `var` now raises an error for non-float inputs to prevent an overflow bug.

False Positives (10). Numeric errors yield four benign NumPy-CuPy differences and one in Octave. CuPy-8755 reports that they vary in an integer computation. The reason is that even if users declare an integer ndarray, CuPy treats its elements as floating-point type in the GPU computation. Another issue related to the GPU is the float16 precision difference between GPU and CPU [92] (posted in CuPy-9086 posted by others). CuPy-8736 reports that the results of applying `prod` on numbers are different. The reason is the difference in computation order. Different computation orders cause different intermediate errors, thereby affecting numerical accuracy [38]. The last one is that ndarrays may have a numeric error that we allow (see Formula 1), but using them as the inputs triggers a difference. For example, we may get a 0 and 1^{-8} from NumPy and CuPy. The difference is smaller than our threshold (1^{-7}), but calling `sign` returns a 0 (zero) and 1 (positive), leading to an unignorable error. Such a difference exists in Octave, too. Two differences are due to undefined behaviors: NumPy's `reshape` and `empty` do not guarantee the output ndarray's memory layout. CuPy-8779 highlights a performance-driven divergence: while `ravel` affects array contiguity, CuPy does not guarantee NumPy-identical properties to minimize execution overhead. Fig. 7 presents another change of CuPy for performance, detected by TitanFuzz, when integer arrays reference the

same memory location multiple times [11]. NumPy-28376 shows that `signbit` for NaN differs on AMD64 and ARM64 because their low-level dependencies represent the NaN sign bit inconsistently.
•**Analysis:** Octave yields significantly fewer differences. The main reasons are its simplicity and maturity. Unlike NumPy/CuPy, Octave avoids view-related bugs (e.g., the view created by `flipud` in Fig. 9) by returning fresh arrays rather than views [60]. Moreover, with a stable codebase since version 4 (2015) [91], Octave offers greater consistency than rapidly evolving NumPy and CuPy.

Answer to RQ3: The primary root causes of the detected differences with valid inputs include program bugs, API changes, and numerical errors caused by platform differences.

5 Discussion

5.1 Choice of AP Pairs under Test

Section 4.2 chooses five AP pairs for differential testing. NumPy/CuPy versus Octave is not chosen. NumPy view mechanism makes many shared APIs by NumPy and Octave unsuitable for sequence-level testing [60]. Fig. 11 illustrates this: updating `a` also modifies its view `b` (`flipud(a)`), whereas Octave's `b` uses new memory and remains unchanged. Widespread use of NumPy views leads to an excess of such false positives, masking real differences. Restricting tests to non-view APIs is counterproductive, as view-related bugs constitute the majority of our bug findings.

```
1 import numpy as np # Octave
2 a = np.array([[1.], [2.]]) # a = [1.; 2.]
3 b = np.flipud(a) # b = flipud(a)
4 np.exp2(a, out=a) # a(:) = 2 .^ a
5 # np b: [[2.], [4.]] octave b: [[2.], [1.]]
```

Fig. 11. NumPy's view mechanism.

5.2 Threats to Validity

Randomness. The algorithm randomness is a threat. To enhance the result's stability, we run ArrayDiff and other baselines for 10 hours, five times.
Limitations. ArrayDiff may miss certain crash-related bugs, as it marks a test invalid if both implementations crash. It also misses differences due to the threshold for float values.
Parameters. RA and GA have two parameters (see Algorithm 4 and 5) to control the population size and the initial test length. Smaller values reduce the test diversity. Larger values spend more time running tests, reducing the iteration rounds. Therefore, both large and small numbers may weaken the performance. We plan to study them in the future.
Experimental subjects. We pick five pairs as the subjects. RQ2's results of all pairs (Section 4.2) show that IRA-ACG stably enhances ArrayDiff in generating valid inputs.

6 Conclusion

We propose ArrayDiff, a differential tester for AP frameworks. It contains an LLM-assisted requirement collector and a requirement-aware API call generator (IRA-ACG). ArrayDiff integrates search algorithms to evolve API call sequences based on the IRA-ACG for difference detection. Experimental results show that IRA-ACG improves ArrayDiff in detecting valid-input differences.

7 Data Availability

We present ArrayDiff's prototype, experimental data, and detection details at Zenodo.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and feedback. This work was supported by ShanghaiTech.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *Proceedings of OSDI*. 265–283. doi:10.48550/arXiv.1605.08695

[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486

[3] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31. doi:10.1145/3391533

[4] Jakub Bachurski and Alan Mycroft. 2024. Points for Free: Embedding Pointful Array Programming in Python. In *Proceedings of ARRAY*. 1–12. doi:10.1145/3652586.3663312

[5] Jakub Bachurski, Alan Mycroft, and Dominic Orchard. 2025. Structuring Arrays with Algebraic Shapes. In *Proceedings of ARRAY*. 1–16. doi:10.1145/3736112.3736141

[6] Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and distributed array computing. In *Proceedings of SC*. 1–23. doi:10.1145/3295500.3356175

[7] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2018. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2018). doi:10.1109/TPDS.2018.2872064

[8] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235. doi:10.1016/j.infsof.2018.08.010

[9] Peter JA Cock, Tiago Antao, Jeffrey T Chang, Brad A Chapman, et al. 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* 25, 11 (2009), 1422–1423. doi:10.1093/bioinformatics/btp163

[10] CuPy. 2025. CuPy sort tests. https://github.com/cupy/cupy/blob/v13.3.0/tests/cupy_tests/sorting_tests/test_sort.py. [Online; accessed 1-July-2025].

[11] CuPy Developers. 2026. CuPy User Guide: Differences from NumPy. https://docs.cupy.dev/en/stable/user_guide/difference.html. Accessed: 2026-01-30; Official CuPy documentation.

[12] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?. In *Proceedings of ISSTA*. 57–67. doi:10.1145/3092703.3092727

[13] Kalyanmoy Deb. 2014. *Multi-objective optimization*. 403–449.

[14] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. In *Proceedings of ICSE*. doi:10.1109/ICSE55347.2025.00032

[15] Yinlin Deng, Chunqiu Steven Xia, Zhezhen Cao, Meiziniu Li, and Lingming Zhang. 2024. Can LLMs implicitly learn numeric parameter constraints in data science APIs? *Advances in Neural Information Processing Systems* 37 (2024), 54205–54238. doi:10.52202/079017-1718

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of ISSTA*. 423–435. doi:10.1145/3597926.3598067

[17] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of ESEC/FSE*. 44–56. doi:10.1145/3540250.3549085

[18] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of ASE*. IEEE, 509–519. doi:10.1109/ASE.2017.8115662

[19] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of ICSE*. 481–492. doi:10.1145/3377811.3380402

[20] dpnp. 2025. NumPy APIs. https://intelpython.github.io/dpnp/reference/comparison.html. [Online; accessed 1-July-2025].

[21] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *International Symposium on Search Based Software Engineering*. Springer, 105–120. doi:10.1007/978-3-030-59762-7_8

[22] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of ISSTA*. 1656–1668. doi:10.1145/3650212.3680389

[23] Nicolas Erni, Mohammed Al-Ameen, Christian Birchler, Pouria Derakhshanfar, Stephan Lukasczyk, and Sebastiano Panichella. 2024. SBFT tool competition 2024-python test case generation track. In *Proceedings of SBFT*. 37–40. doi:10.1145/3643659.3643930

[24] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *Proceedings of ESEC-FSE companion*. 549–552. doi:10.1145/1295014.1295038

[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of ESEC/FSE*. 416–419. doi:10.1145/2025113.2025179

[26] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. doi:10.1109/TSE.2012.14

[27] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of ISSTA*. 147–158. doi:10.1145/1831708.1831728

[28] Gregory Gay. 2017. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*. Springer, 65–82. doi:10.1007/978-3-319-66299-2_5

[29] M N Gevorkyan, A V Demidova, A V Korolkova, and D S Kulyabov. 2019. Statistically significant performance testing of Julia scientific programming language. *Journal of Physics: Conference Series* 1205, 1 (2019), 012017. doi:10.1088/1742-6596/1205/1/012017

[30] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of PLDI*. 206–215. doi:10.1145/1375581.1375607

[31] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of ICSE*. 1–13. doi:10.1145/3597503.3639581

[32] Giovanni Grano, Christoph Laaber, Annibale Panichella, and Sebastiano Panichella. 2019. Testing with fewer resources: An adaptive approach to performance-aware test case generation. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2332–2347. doi:10.1109/TSE.2019.2946773

[33] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *Proceedings of ICSE-SEIP*. IEEE, 71–80. doi:10.1109/ICSE-SEIP.2019.00016

[34] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* (2025). doi:10.1038/s41586-025-09422-z

[35] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *Proceedings of ASE*. 486–498. doi:10.1145/3324884.3416571

[36] Fitash Ul Haq, Donghwan Shin, Lionel C Briand, Thomas Stifter, and Jun Wang. 2021. Automatic test suite generation for key-points detection DNNs using many-objective search (experience paper). In *Proceedings of ISSTA*. 91–102. doi:10.1145/3460319.3464802

[37] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. doi:10.1038/s41586-020-2649-2

[38] Nicholas J Higham. 2002. *Accuracy and stability of numerical algorithms*. SIAM. doi:10.1137/1.9780898718027.bm

[39] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proceedings of SCAM*. 249 – 258. doi:10.1109/SCAM.2008.36

[40] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[41] J Robert Johansson, Paul D Nation, and Franco Nori. 2012. QuTiP: An open-source Python framework for the dynamics of open quantum systems. *Computer physics communications* 183, 8 (2012), 1760–1772. doi:10.1016/j.cpc.2012.02.021

[42] Phil Kim. 2017. Matlab deep learning. *With machine learning, neural networks and artificial intelligence* 130, 21 (2017), 151. doi:10.1007/978-1-4842-2845-6

[43] Sebastian Krämer, David Plankensteiner, Laurin Ostermann, and Helmut Ritsch. 2018. QuantumOptics. jl: A Julia framework for simulating open quantum systems. *Computer Physics Communications* 227 (2018), 109–116. doi:10.1016/j.cpc.2018.02.004

[44] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2021. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2021), 14–14. doi:10.1145/3364452.3364455

[45] Meiziniu Li, Dongze Li, Jianmeng Liu, Jialun Cao, Yongqiang Tian, and Shing-Chi Cheung. 2026. Enhancing Differential Testing With LLMs For Testing Deep Learning Libraries. *ACM Transactions on Software Engineering and Methodology* (2026). doi:10.1145/3735637

[46] Xinyi Li, Mark Baranowski, Harvey Dam, and Ganesh Gopalakrishnan. 2025. Array Programming on GPUs: Challenges and Opportunities. In *Proceedings of ARRAY*. 41–52. doi:10.1145/3736112.3736144

[47] Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. 2023. Generation-based differential fuzzing for deep learning libraries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–28. doi:10.1145/3628159

[48] 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 ASPLOS*. 530–543. doi:10.1145/3575693.3575707

[49] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of ICSE*. 1–13. doi:10.1145/3597503.3639180

[50] Carlo Lucibello and Aurora Rossi. 2025. GraphNeuralNetworks. jl: Deep Learning on Graphs with Julia. *Journal of Machine Learning Research* 26, 80 (2025), 1–6. doi:10.48550/arXiv.2412.06354

[51] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of ICSE-Companion*. 168–172. doi:10.1145/3510454.3516829

[52] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (2023), 36. doi:10.1007/s10664-022-10248-w

[53] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/joss.01891

[54] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of ISSTA*. 94–105. doi:10.1145/2931037.2931054

[55] Wendy L Martinez, Angel R Martinez, and Jeffrey Solka. 2017. *Exploratory data analysis with MATLAB*. doi:10.1201/b10434

[56] MathWorks. 2025. Matlab official website. https://www.mathworks.com/products/matlab.html. [Online; accessed 1-July-2025].

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

[58] Meta AI and Hugging Face. 2022. facebook/incoder-1B: InCoder 1B parameter code model. https://huggingface.co/facebook/incoder-1B. Accessed: 2026-01-30; Model card on Hugging Face Model Hub.

[59] NumPy. 2024. NumPy 2.0.0 Release Notes. https://numpy.org/doc/stable/release/2.0.0-notes.html. Accessed: 2026-01-30; Official release notes on NumPy documentation site.

[60] NumPy. 2025. Key differences between NumPy and Matlab/Octave. https://tinyurl.com/ynasry65. [Online; accessed 1-July-2025].

[61] NumPy. 2025. NumPy float testing. https://numpy.org/doc/stable/reference/generated/numpy.testing.assert_allclose.html. [Online; accessed 1-July-2025].

[62] NumPy. 2025. NumPy official website. https://numpy.org. [Online; accessed 1-July-2025].

[63] NumPy. 2025. NumPy sort document. https://numpy.org/doc/stable/reference/generated/numpy.sort.html. [Online; accessed 1-July-2025].

[64] Octave. 2025. Octave official website. https://octave.org. [Online; accessed 1-July-2025].

[65] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of earningSys NIPS*.

[66] Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, Houman Homayoun, et al. 2024. Fuzzing {BusyBox}: Leveraging {LLM} and Crash Reuse for Embedded Bug Unearthing. In *Proceedings of USENIX Security*. 883–900. doi:10.48550/arXiv.2403.03897

[67] OpenAI. 2026. OpenAI Codex Developer Documentation. https://developers.openai.com/codex. Accessed: 2026-01-30; Official developer documentation for the OpenAI Codex coding agent.

[68] Shiwen Ou, Yuwei Li, Lu Yu, Chengkun Wei, Tingke Wen, Qiangpu Chen, Yu Chen, Haizhi Tang, and Zulie Pan. 2025. MirrorFuzz: Leveraging LLM and Shared Bugs for Deep Learning Framework APIs Fuzzing. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3619966

[69] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *OOPSLA-Companion*. 815–816. doi:10.1145/1297846.1297902

[70] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *Proceedings of ICST*. 1–10. doi:10.1109/ICST.2015.7102604

[71] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44 (2018), 122–158. doi:10.1109/TSE.2017.2663435

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019). doi:10.48550/arXiv.1912.01703

[73] Adam Paszke, Daniel D Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. doi:10.1145/3473593

[74] Jonathan Peirce, Jeremy R Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, Erik Kastman, and Jonas Kristoffer Lindeløv. 2019. PsychoPy2: Experiments in behavior made easy. *Behavior research methods* 51 (2019), 195–203. doi:10.3758/s13428-018-01193-y

[75] Jeffrey M Perkel et al. 2019. Julia: come for the syntax, stay for the speed. *Nature* 572, 7767 (2019), 141–142. doi:10.1038/d41586-019-02310-3

[76] Alexander Prochnow and Jinqiu Yang. 2022. DiffWatch: watch out for the evolving differential testing in deep learning libraries. In *Proceedings ICSE-Companion*. 46–50. doi:10.1145/3510454.3516835

[77] Thomas P Robitaille, Erik J Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis, Adam Ginsburg, Adrian M Price-Whelan, Wolfgang E Kerzendorf, et al. 2013. Astropy: A community Python package for astronomy. *Astronomy & Astrophysics* 558 (2013), A33. doi:10.1051/0004-6361/201322068

[78] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Search-Based Software Engineering*, Márcio Barros and Yvan Labiche (Eds.). 93–108. doi:10.1007/978-3-319-22183-0_7

[79] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of ASE*. 287–298. doi:10.1145/3324884.3416622

[80] Paolo Tonella. 2004. Evolutionary Testing of Classes. In *Proceedings of ISSTA*. 119–128. doi:10.1145/1007512.1007528

[81] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. doi:10.3102/10769986025002101

[82] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. doi:10.1038/s41592-019-0686-2

[83] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of ICSE*. IEEE, 724–735. doi:10.1109/ICSE.2019.00081

[84] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[85] Jiannan Wang, Hung Viet Pham, Qi Li, Lin Tan, Yu Guo, Adnan Aziz, and Erik Meijer. 2024. D 3: Differential Testing of Distributed Deep Learning with Model Generation. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3461657

[86] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. 2021. Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?. In *Proceedings of ICSE*. 1548–1560. doi:10.1109/ICSE43902.2021.00138

[87] Taiyan Wang, Ruipeng Wang, Yu Chen, Lu Yu, Zulie Pan, Min Zhang, Huimin Ma, and Jinghua Zheng. 2024. Enhancing Black-box Compiler Option Fuzzing with LLM through Command Feedback. In *Proceedings of ISSRE*. IEEE, 319–330. doi:10.1109/ISSRE62328.2024.00039

[88] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of ESEC/FSE*. 788–799. doi:10.1145/3368089.3409761

[89] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of ICSE*. 995–1007. doi:10.1145/3510003.3510041

[90] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of SciPy*. 56 – 61. doi:10.25080/Majora-92bf1922-00a

[91] Wikipedia contributors. 2026. GNU Octave. https://en.wikipedia.org/wiki/GNU_Octave. Accessed: 2026-02-19.

[92] Nicholas Wilt. 2013. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.

[93] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. In *Proceedings of ICSE*. IEEE, 1174–1186. doi:10.1109/ICSE48619.2023.00105

[94] Kohsuke Yatoh, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2015. Feedback-controlled random test generation. In *Proceedings of ISSTA*. 316–326. doi:10.1145/2771783.2771805

[95] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *Proceedings of ICSME*. 235–245. doi:10.1109/ICSME.2019.00035

[96] Tong Zhou, Jun Shirako, and Vivek Sarkar. 2024. APPy: Annotated Parallelism for Python on GPUs. In *Proceedings of CC*. 113–125. doi:10.1145/3640537.3641575

[97] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. 2024. Coverage goal selector for combining multiple criteria in search-based unit test generation. *IEEE Transactions on Software Engineering* 50, 4 (2024), 854–883. doi:10.1109/TSE.2024.3366613

[98] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: A benchmarking suite for high-performance NumPy. In *Proceedings of ICS*. 63–74. doi:10.1145/3447818.3460360

Received 2026-02-20; accepted 2026-03-24