AI for Program Analysis

Vasu Vikram

Recap: Program Analysis

Goal: Automatically reason about program behavior to find bugs, prove properties, optimize code, etc.

Static Analysis: Analyze code without executing

- Successful techniques in practice: type checkers, null checkers (NulAway/NilAway at Uber), linters

Dynamic Analysis: Analyze executing programs

- Successful techniques in practice: fuzz testing, race detection (Tsan), profilers

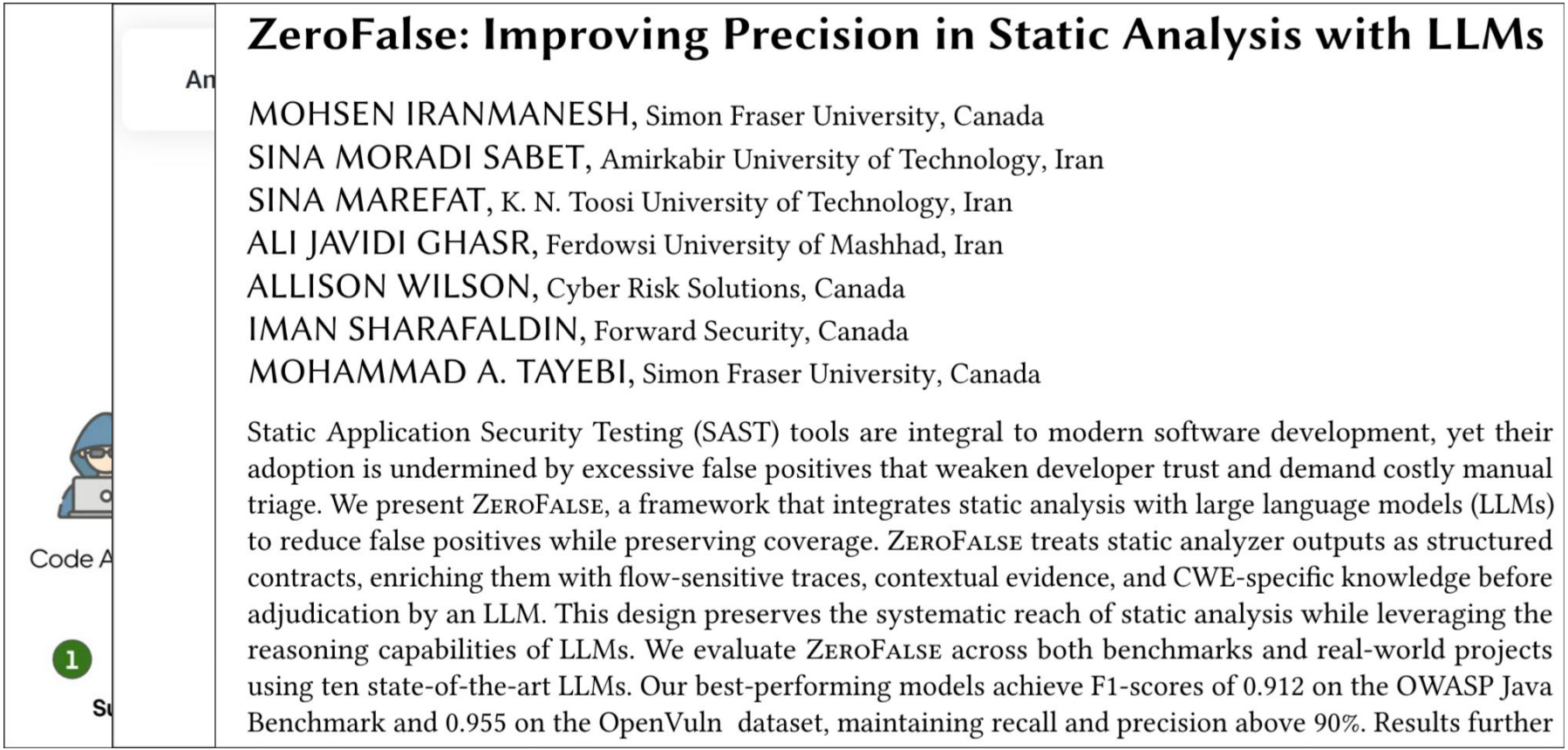
Limitations of Static Analysis

Over-approximation and imprecision leads to **false positives** and **usability** concerns!

It’s hard to model different language features precisely, like aliasing, heap, pointers, dynamic dispatch, reflection, I/O, concurrency, etc.

How can LLMs help with some of these issues?

How can LLMs augment static analysis?



ZeroFalse: Improving Precision in Static Analysis with LLMs

MOHSEN IRANMANESH, Simon Fraser University, Canada
SINA MORADI SABET, Amirkabir University of Technology, Iran
SINA MAREFAT, K. N. Toosi University of Technology, Iran
ALI JAVIDI GHASR, Ferdowsi University of Mashhad, Iran
ALLISON WILSON, Cyber Risk Solutions, Canada
IMAN SHARAFALDIN, Forward Security, Canada
MOHAMMAD A. TAYEBI, Simon Fraser University, Canada

Static Application Security Testing (SAST) tools are integral to modern software development, yet their adoption is undermined by excessive false positives that weaken developer trust and demand costly manual triage. We present ZeroFalse, a framework that integrates static analysis with large language models (LLMs) to reduce false positives while preserving coverage. ZeroFalse treats static analyzer outputs as structured contracts, enriching them with flow-sensitive traces, contextual evidence, and CWE-specific knowledge before adjudication by an LLM. This design preserves the systematic reach of static analysis while leveraging the reasoning capabilities of LLMs. We evaluate ZeroFalse across both benchmarks and real-world projects using ten state-of-the-art LLMs. Our best-performing models achieve F1-scores of 0.912 on the OWASP Java Benchmark and 0.955 on the OpenVuln dataset, maintaining recall and precision above 90%. Results further

How can LLMs augment static analysis?

Can LLMs be used to *build* analyzers themselves?

KNighter: Transforming Static Analysis with LLM-Synthesized Checkers

Chenyuan Yang
University of Illinois Urbana-Champaign
USA
cy54@illinois.edu

Zijie Zhao
University of Illinois Urbana-Champaign
USA
zijie4@illinois.edu

Zichen Xie
Zhejiang University
China
xiezichen@zju.edu.cn

Haoyu Li
Shanghai Jiao Tong University
China
learjet@sjtu.edu.cn

Lingming Zhang
University of Illinois Urbana-Champaign
USA
lingming@illinois.edu

How can LLMs augment static analysis?

Key Ideas:

1. Mine Git
2. Synthesi

s.

atterns.

```
void checkLocation(...) const {
  ...
  // Look up the region in the PossibleNullPtrMap.
  const bool *Checked = State->get<PossibleNullPtrMap>(MR);
  // If the region is recorded as unchecked, warn.
  if (Checked && *Checked == false)
    reportUncheckedDereference(MR, S, C);
}
void checkBind(...) const {
  ...
  // For pointer assignments, update the aliasing map.
  State = State->set<PtrAliasMap>(LHSReg, RHSReg);
  State = State->set<PtrAliasMap>(RHSReg, LHSReg);
 }
```

(c) A checker synthesized by KNighter for the patch in Fig. 2a.

Yang, Chenyuan, et al. "KNighter: Transforming static analysis with LLM-synthesized checkers, 2025." URL https://arxiv.org/abs/2503.09002 (2025).

Considerations: AI-based static analysis in practice

Recall Shrey's lecture: additional dimensions to evaluate in practice

- **Retrospective Detection Rate:** detecting historic bugs that lead to past production incidents
- **Preventable Incident Count:** detecting bugs before they reached production
- **Developer Satisfaction Rate:** Median developer feedback very positive*
- **Comment Addressal Rate:** % of the posted comments resolved

How can LLMs augment dynamic analysis?

We've learned about:

- Dynamic symbolic execution
- Concolic testing
- Greybox fuzzing
- Mutation testing

In-class exercise: choose one of these techniques we've learned about.

1. Write a limitation of the technique.

2. Brainstorm a way LLMs can help with this limitation.

3. What are some tradeoffs / considerations when using LLMs in this way?

LLM-guided Concolic Testing

Agentic Concolic Execution

Zhengxiong Luo*, Huan Zhao*, Dylan Wolff*, Cristian Cadar†, Abhik Roychoudhury*
* *National University of Singapore*
† *Imperial College London*
{luozx, abhik}@nus.edu.sg, {zhaohuan, wolffd}@comp.nus.edu.sg, c.cadar@imperial.ac.uk

***Abstract*—Concolic execution is a practical test generation technique that explores execution paths by coupling concrete execution with symbolic reasoning. It runs programs on given inputs while capturing symbolic path representations, then mutates and solves these constraints to generate new test inputs for alternative paths. This approach has several fundamental challenges, such as (C1) the inherent complexity of symbolically modeling diverse programming language constructs and environmental interactions, and (C2) the scalability issues of constraint solvers when handling large, complex formulas.**

gather symbolic constraints representing the executed path, then negates selected constraints and solves for new inputs to explore alternative paths [2]. These approaches have achieved success in discovering vulnerabilities in real-world software [3], played key roles in competitions like DARPA Cyber Grand Challenges [4], [5], and proven effective in both FLOSS and commercial software [2], [6], [7].

Despite their success, these approaches still face two fundamental challenges when applied to modern software:

***C1**: Complex and Incomplete Implementations.* Modern

Luo, Zhengxiong, et al. "Agentic Concolic Execution." Proceedings of the IEEE Symposium on Security and Privacy (S&P). 2026.

LLM-assisted Fuzz Testing

STITCH: Synthesizing Test Inputs Through Constrained Harnessing

Harrison Green
Carnegie Mellon University
harrisog@cmu.edu

Fraser Brown
Carnegie Mellon University
fraserb@cmu.edu

Claire Le Goues
Carnegie Mellon University
clegoues@cmu.edu

Abstract

Fuzz harnessing, the process of codifying how a fuzzer interacts with its target, is typically a manual, burdensome task requiring expertise in both fuzzing and the target domain. Recent attempts to automate the process take a scattershot approach, generating hundreds or thousands of discrete fuzz harnesses, and selecting some of them to run; suffering from issues of scalability, overlapping functionality, and requiring significant human effort to correct and maintain. Furthermore, intricate API rules are not always captured in the harnesses, leading to false positives or missed coverage.

the target API-under-test. In essence, the harness defines the fuzzer's search space: *which possible arguments to which API inputs can be tested*.

In domains where potential attackers control both the sequence of API invocations and their arguments, such as web APIs [3], kernels [8], or JavaScript engines [10, 11], fuzz harnesses can be extremely flexible. They can issue arbitrary API sequences with arbitrary arguments, and *any* crashing test-case is interesting even when the API is used in semantically invalid ways.

By contrast, for C/C++ library APIs (the focus of this work),

LLM-synthesized Mutants for Mutation Testing

LLMorpheus: Mutation Testing Using Large Language Models

Frank Tip, *Senior Member, IEEE*, Jonathan Bell, and Max Schäfer

***Abstract*—In mutation testing, the quality of a test suite is evaluated by introducing faults into a program and determining whether the program's tests detect them. Most existing approaches for mutation testing involve the application of a fixed set of mutation operators, e.g., replacing a "+" with a "-", or removing a function's body. However, certain types of real-world bugs cannot easily be simulated by such approaches, limiting their effectiveness. This paper presents a technique for mutation testing where placeholders are introduced at designated locations in a program's source code and where a Large Language Model (LLM) is prompted to ask what they could be replaced with. The technique is implemented in *LLMorpheus*, a mutation testing tool for JavaScript, and evaluated on 13 subject packages, considering several variations on the prompting strategy, and using several LLMs. We find *LLMorpheus* to be capable of producing mutants that resemble existing bugs that cannot be produced by *StrykerJS*, a state-of-the-art mutation testing tool. Moreover, we report on**

have shown that, given two test suites for the same system under test, the one that detects more mutants (even using only these limited mutation operators) is likely to also detect more real faults [5], [6].

However, not *all* real faults are coupled to mutants due to the limited set of mutation operators. For example, a fault resulting from calling the wrong method on an object is unlikely to be coupled to a mutant, as state-of-the-art mutation tools do not implement a "change method call" operator. While a far wider range of mutation operators has been explored in the literature [7], [8], state-of-the-practice tools like Pitest [9], [10], Major [11], and Stryker [12] typically do not implement them because of the implementation effort required and, especially, the increased cost of mutation analysis. Each additional mu-

Tip, Frank, Jonathan Bell, and Max Schäfer. "Llmorpheus: Mutation testing using large language models." IEEE Transactions on Software Engineering (2025).

Considerations: AI-based Dynamic Analysis in practice

Now, we can automatically (and easily) do things that required substantial manual effort. **This substantially increases the bug-finding surface**!

Things to consider:

- Why are we able to find more bugs? Is it the program analysis technique? Are the LLMs just really capable?
- How would the next LLM impact the dynamic analysis system you've built?
- How do we properly report these bugs? See Google vs. ffmpeg battle

Program Analysis for AI?

LLMs can be used to generate/refactor/optimize code

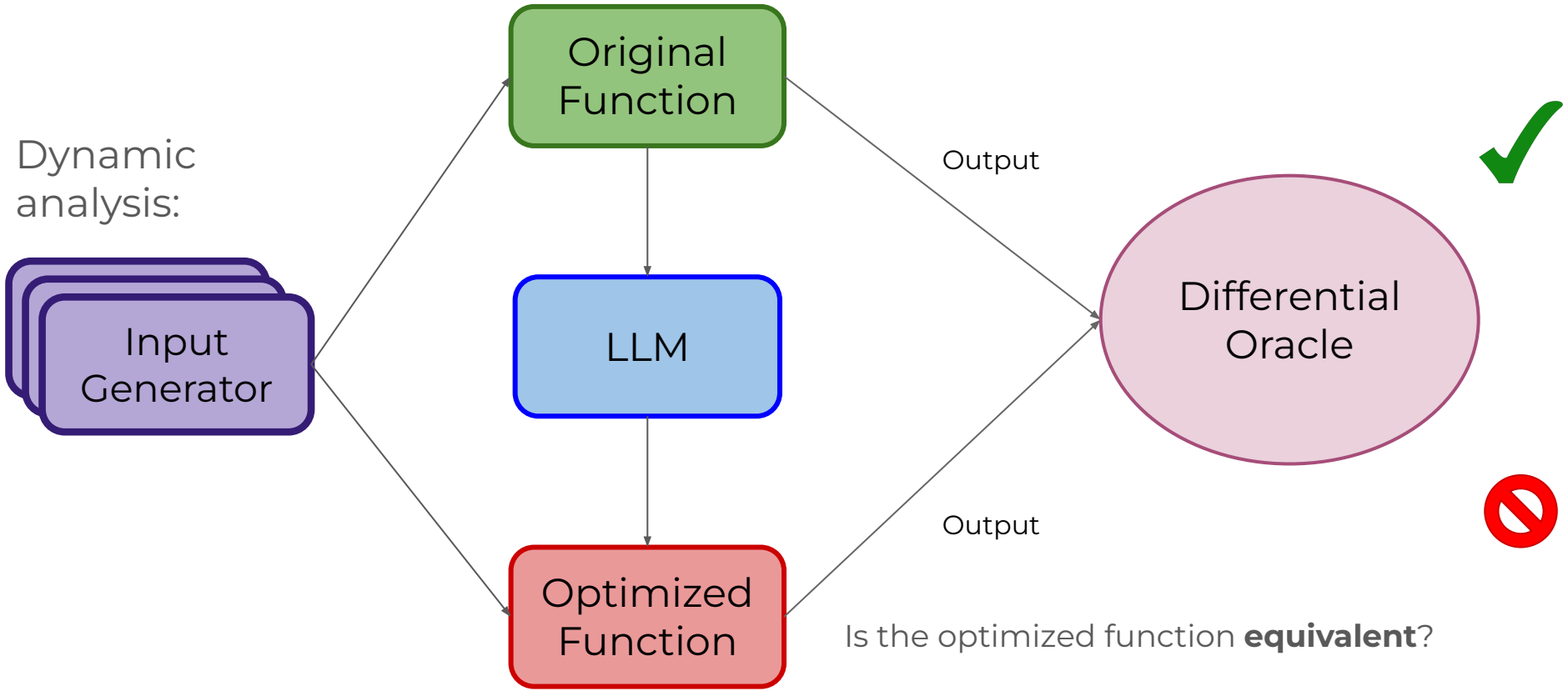
Question: how do we measure correctness of LLM-generated code?

Pre-training: Evaluate next-token accuracy (loss function) on static datasets

Post-training: Use verifiable rewards (e.g. compilations, static analysis, unit tests) during fine-tuning and reinforcement learning

Inference-time / agents: Execute generated code against test suites, run dynamic analysis, call MCP tools, etc.

Example: LLMs for Codebase Optimizations



Where else could program analysis fit in?

In-class exercise: write down how we could use a program analysis technique that learned in class to help with some aspect of LLM code generation.

(AI for)
Program Analysis for AI?

How do coding agents generally check correctness?

In almost all cases:
writing unit tests

What are some dangers here?

b. Write tests, commit; code, iterate, commit

This is an Anthropic-favorite workflow for changes that are easily verifiable with unit, integration, or end-to-end tests. Test-driven development (TDD) becomes even more powerful with agentic coding:

1. **Ask Claude to write tests based on expected input/output pairs.** Be explicit about the fact that you're doing test-driven development so that it avoids creating mock implementations, even for functionality that doesn't exist yet in the codebase.
2. **Tell Claude to run the tests and confirm they fail.** Explicitly telling it not to write any implementation code at this stage is often helpful.
3. **Ask Claude to commit the tests** when you're satisfied with them.
4. **Ask Claude to write code that passes the tests,** instructing it not to modify the tests. Tell Claude to keep going until all tests pass. It will usually take a few iterations for Claude to write code, run the tests, adjust the code, and run the tests again.
    1. At this stage, it can help to ask it to verify with independent subagents that the implementation isn't overfitting to the tests
5. **Ask Claude to commit the code** once you're satisfied with the changes.

https://www.anthropic.com/engineering/claude-code-best-practices

Can LLMs be used to generate property-based tests?

```
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

Input Generator

Can LLMs be used to generate property-based tests?

```python
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

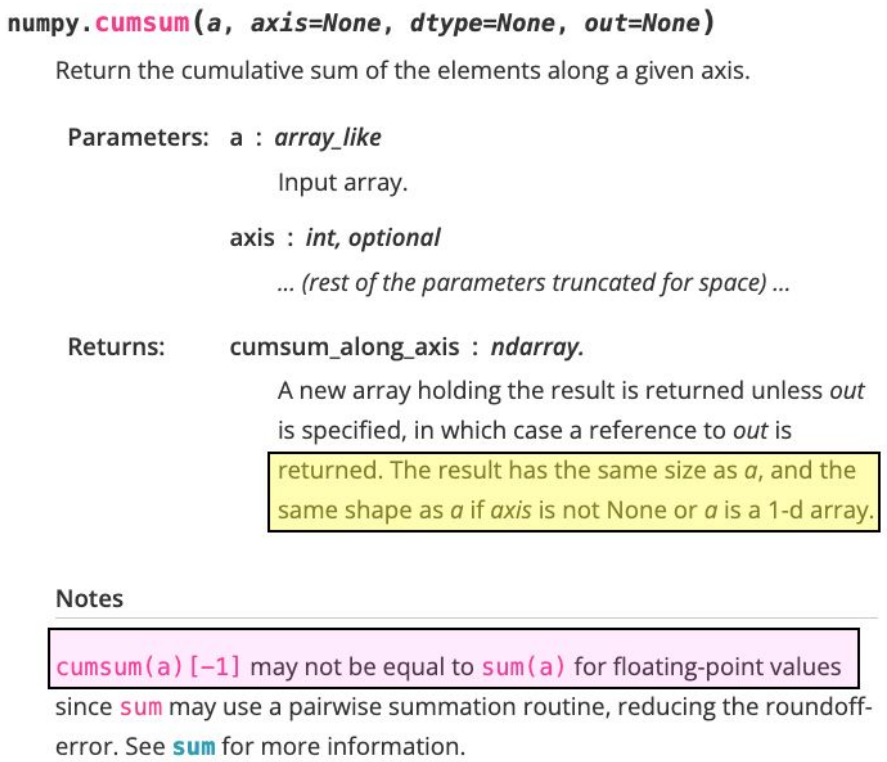
Property

Vikram, Vasudev, et al. "Can large language models write good property-based tests?." arXiv preprint arXiv:2307.04346 (2023).

Can LLMs be used to generate property-based tests?

Given a natural language specification (e.g. documentation):

can we translate these to **executable PBTs**?



numpy.cumsum(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along a given axis.

Parameters: **a** : *array_like*

Input array.

axis : *int, optional*

... (rest of the parameters truncated for space) ...

Returns: **cumsum_along_axis** : *ndarray.*

A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

Notes

`cumsum(a)[-1]` may not be equal to `sum(a)` for floating-point values since `sum` may use a pairwise summation routine, reducing the roundoff-error. See **sum** for more information.

Vikram, Vasudev, et al. "Can large language models write good property-based tests?." arXiv preprint arXiv:2307.04346 (2023).

Can LLMs be used to generate property-based tests?

```python
from hypothesis import given, strategies as st
import numpy as np

# Summary: Generate random input parameters for numpy.cumsum
@given(st.data())
def test_numpy_cumsum(data):
    # Generating a list with varying length and integer elements
    a = data.draw(st.lists(st.integers(min_value=-10, max_value=10), min_size=0, max_size=10))

    # Generate random axis
    axis = data.draw(st.one_of(st.none(), st.integers(min_value=0, max_value=a.ndim-1)))

    # Call numpy.cumsum with generated input
    cumsum_result = np.cumsum(a, axis=axis)

    # Check: output shape should be the same as input shape if conditions met
    if axis is not None or a.ndim == 1:
        assert cumsum_result.shape == a.shape

    # Check: output size should be the same as input size
    assert cumsum_result.size == a.size

    # Check: cumsum(a)[-1] should be approx equal to sum(a) for non floats
    if not np.issubdtype(a.dtype, np.floating):
        np.testing.assert_almost_equal(cumsum_result.flatten()[-1], np.sum(a))
```

Vikram, Vasudev, et al. "Can large language models write good property-based tests?." arXiv preprint arXiv:2307.04346 (2023).

Considerations: how do we know the PBT is good?

Validity: is the PBT able to run without errors?

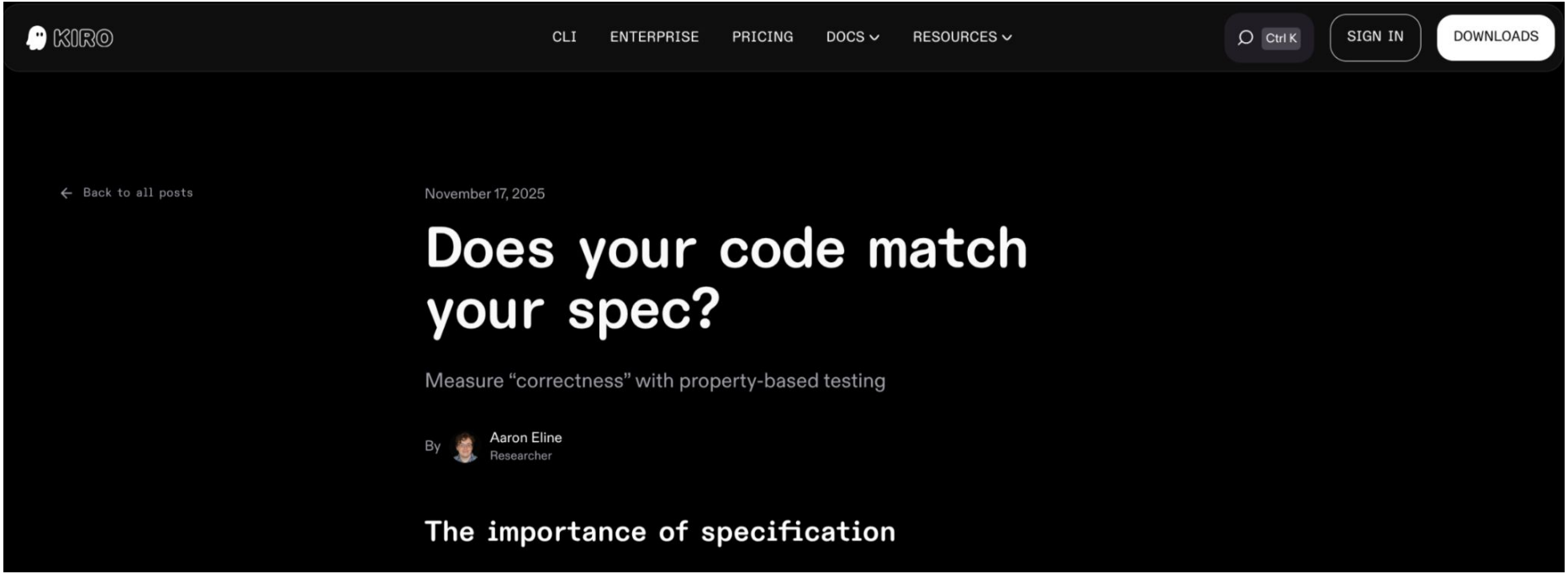
Soundness: are the assertions in the PBT sound?

Input Diversity: are the generated inputs testing different things?

PBT Strength: are the assertions actually checking the properties from the specification?

Vikram, Vasudev, et al. "Can large language models write good property-based tests?." arXiv preprint arXiv:2307.04346 (2023).

Kiro: agentic IDE that does this in practice!



Key Takeaways

1. AI can be used to overcome many traditional program analysis limitations.
   - Considerations: scaling to large codebases, latency, cost
2. Program analysis is used check correctness of LLM-generated code.
   - Considerations: which PA techniques are best suited for specific LLM programming tasks?
3. AI agents can go hand-in-hand with program analysis
   - Considerations: how do we best expose PA tools to agents? How do we measure success?