DyLin: A Dynamic Linter for Python

ANONYMOUS AUTHOR(S)

Python is a dynamic language with applications in many domains, and one of the most popular languages in recent years. To achieve higher code quality, developers have turned to "linters" that statically analyze the source code and warn about potential programming problems. However, the inherent limitations of static analysis and the dynamic nature of Python make it difficult or even impossible for static analysis to detect some problems. This paper presents DyLin, the first dynamic linter for Python. Similar to a traditional linter, the approach has an extensible set of checkers, which, unlike in traditional linters, search for specific programming anti-patterns by analyzing the program as it executes. A key contribution of this paper is a set of 15 Python-specific anti-patterns that are hard to find statically but amenable to dynamic analysis, along with corresponding checkers to detect them. Our evaluation applies DyLin to 37 popular open-source Python projects on GitHub and to a dataset of code submitted to Kaggle machine learning competitions, totaling over 683k lines of Python code. The approach reports a total of 68 problems, 48 of which are previously unknown true positives, i.e., a precision of 70.6%. The detected problems include bugs that cause incorrect values, such as `inf`, incorrect behavior, e.g., missing out on relevant events, unnecessary computations that slow down the program, and unintended data leakage from test data to the training phase of machine learning pipelines. These issues remained unnoticed in public repositories for more than 3.5 years, on average, despite the fact that the corresponding code has been exercised by the developer-written tests. A comparison with popular static linters and a type checker shows that DyLin complements these tools by detecting problems that are missed statically. Based on our reporting of 42 issues to the developers, 31 issues have so far been fixed.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software verification and validation**.

Additional Key Words and Phrases: Dynamic analysis, Linter, Bug

ACM Reference Format:
Anonymous Author(s). 2025. DyLin: A Dynamic Linter for Python. 1, 1 (April 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Python has established itself as a mainstream programming language, with substantial presence across a diverse array of application domains, and strong interest by the research community [10, 17, 23, 26, 32, 38, 41, 43, 49]. From web development to data analysis, scientific computing, and machine learning, Python's versatility has made it the language of choice for many developers and organizations. Its elegant and concise syntax, coupled with an extensive ecosystem of libraries and frameworks, has rendered it indispensable in the modern software landscape. One of the key reasons for the widespread adoption of Python is its dynamic nature, which developers celebrate for its flexibility and expressiveness.

Paradoxically, the dynamic nature of Python also renders it susceptible to subtle and difficult to notice programming mistakes. To detect such mistakes, Python developers currently rely on two main techniques: static, lint-style analyses and testing. Static linting tools, such as Pylint, Flake8, and Ruff, while formidable in their own right, are fundamentally rooted in static analysis.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
for event in self.received_events:
    if event['event_descriptor']['event_status'] == 'cancelled' or
        utils.determine_event_status(event['active_period']) == 'completed':
        logger.info(f"Removing event {event} because it is no longer relevant.")
        self.received_events.pop(self.received_events.index(event))
```

Fig. 1. Example from the OpenLEADR project, where removing items while iterating over a list results in unintended behavior for some inputs.

```
def power_method(matrix, epsilon):
    transposed_matrix = matrix.T
    sentences_count = len(matrix)
    p_vector = numpy.array([1.0 / sentences_count] * sentences_count)
    lambda_val = 1.0

    while lambda_val > epsilon:
        next_p = numpy.dot(transposed_matrix, p_vector)
+       next_p /= numpy.linalg.norm(next_p)
        lambda_val = numpy.linalg.norm(numpy.subtract(next_p, p_vector))
        p_vector = next_p

    return p_vector
```

Fig. 2. Buggy implementation of the power method algorithm in the automatic text summarizer project sumy, which can be fixed by adding the line highlighted in green.

They scan the source code for potential issues by applying a set of predefined rules. However, their effectiveness is inherently limited [21, 28], particularly given Python's dynamic type system and some of its highly dynamic features, such as metaclasses, decorators, and dynamic attribute assignment. Hence, linters can only approximate the program's behavior and often fail to uncover subtle runtime errors that manifest under specific conditions [18]. Testing, on the other hand, is a powerful technique to detect bugs, but it is inherently limited by the test cases that developers write. In particular, a test case can reveal only violations of assertions that the developer has thought of and written down, even when the buggy behavior is exercised by the test input.

Figure 1 and Figure 2 show two examples of mistakes missed by current techniques. The problem in Figure 1 shows a loop that iterates over a list, and removes an element from the list if a complex condition is true. Modifying a list during iteration is strongly discouraged as it easily leads to subtle mistakes. In the given example, modifying the list will cause missing events if two adjacent events in the list get removed. The popular Pylint tool indeed has a checker that tries to warn about code that modifies a list while iterating over it. Unfortunately, the checker misses the bug because Pylint only reports a warning if it is certain that the iterated-over object is a list. Interestingly, the existing test suite of the project exercises the buggy code, but hits a path where an item gets removed from the iterated-over list without exposing any checked-for misbehavior. The problem in Figure 2 is a buggy implementation of the power method algorithm, which computes the greatest eigenvalue of a given matrix. Due to an algorithmic mistake, the code returns `inf` values for some matrices. Because the bug is algorithmic, it is hard to detect it with a general-purpose linter. The bug is also missed by the existing test suite of the project, which exercises the buggy code, but does not check the return value against `inf`.

This paper presents DyLin, the first dynamic linter for Python. The approach fills a gap between static linting and testing by checking program executions, e.g., driven by existing tests, against properties that should likely hold, similar to those checked by traditional linters. The difference to static linters is that DyLin operates on program executions rather than static source code, avoiding any approximations imposed by Python's dynamic language features. The difference to testing is that DyLin does not rely on assertions written by developers, enabling the approach to find previously unknown, unintended behavior during the execution of existing tests.

A key contribution that enables DyLin is an extensible set of 15 checkers that target programming problems related to general Python language constructs, usages of standard library functions, custom functions, and – motivated by the importance of Python in machine learning – common mistakes in machine learning code. The checkers are designed to detect problems that are difficult to find statically, but amenable to dynamic analysis. While previous work has investigated dynamically detectable programming anti-patterns in other languages [15, 16], to the best of our knowledge, the set of anti-patterns and checkers described here are the first such set for Python. DyLin can be applied to any execution of the software under analysis, e.g., the executions of existing test suites, newly generated test suites, or production runs.

In addition to a new technique, we also present a novel metric, called *analysis coverage*, to assess how much code gets analyzed by a dynamic analysis. The metric complements classical test coverage metrics by measuring how many of all executed lines are analyzed by at least one of DyLin's checkers. Analysis coverage is useful for understanding the effectiveness of a dynamic analysis and for guiding the development of new checkers.

To evaluate DyLin, we run its checkers on 37 popular, open-source Python projects and on 123 Kaggle submissions. The approach reports 68 warnings, 48 of which are true positive, i.e., a precision of 70.6%. The detected problems include bugs that cause incorrect values, such as `inf`, incorrect behavior, e.g., missing out on relevant events, unnecessary computations that slow down the program, and unintended data leakage from test data to the training phase of machine learning pipelines. We report all 42 true positive issues in the open-source projects to the developers of those projects, and 31 of these issues have been fixed so far. These include bug fixes in popular and well-maintained repositories like keras, kafka-python, and sphinx. Although the source code lines of the issues found by DyLin are covered by tests, the issues were not previously discovered and have existed in the codebase for over 3.5 years, on average. Moreover, we compare DyLin with popular static linters and a type checker, and show that DyLin complements these tools by detecting problems that they are missing.

Measuring the runtime overhead that DyLin imposes, we find that it slows the execution by 6x, on average, which is acceptable for running the approach, at least occasionally, on a test suite or on selected production runs. As a proof-of-concept to validate this claim, we implement a GitHub action, i.e., an automated check that can be added to regression tests executed during continuous integration, and have successfully used it to find some issues in public GitHub repositories.

In summary, this paper contributes the following:

- The first dynamic linting technique for Python.
- A set of 15 Python programming anti-patterns and checkers to detect occurrences of them.
- A novel metric for assessing the amount of code analyzed by a dynamic analysis.
- Empirical evidence that the approach is effective, by identifying 48 previously undetected problems with 70.6% precision.

2 Approach

We start by giving an overview of the approach (Section 2.1) and then present 15 anti-patterns in Python and corresponding checkers to detect them (Sections 2.2 to 2.4). Next, Section 2.5 gives an example of how checkers in DyLin are implemented. Section 2.6 presents our novel analysis coverage metric. Finally, we describe several improvements we made to the underlying analysis framework [12] that improve the effectiveness and efficiency of our approach (Section 2.7).

2.1 Overview

Figure 3 gives an overview of DyLin. Because DyLin is a dynamic linter, the given input program must be executable, which can be achieved, e.g., by running the program's test suite. The core of the approach is a set of checkers, each of which applies a dynamic analysis to a given program to detect occurrences of a specific anti-pattern. The checkers are built on top of DynaPyt [12], which is a general-purpose dynamic analysis framework for Python. DynaPyt facilitates implementing dynamic analyses by providing hooks associated with runtime events, such as function calls, assignments, and comparisons. Each of the checkers in DyLin uses a subset of these hooks to track specific runtime events and to detect occurrences of a specific anti-pattern. Whenever a checker detects an occurrence of an anti-pattern, it reports a warning. In addition, the approach reports the analysis coverage achieved by the checkers, which is a novel metric to assess the amount of code analyzed by a dynamic analysis.

To detect likely misbehavior in a program's execution, DyLin requires a set of anti-patterns to look out for. Given the dynamic nature of the approach, these anti-patterns should be non-trivial to detect statically, but easier to detect when runtime information, such as the type of value or the value itself, is available. To the best of our knowledge, no such set of anti-patterns exists for Python. To address this problem, we gather anti-patterns from three kinds of sources: (i) We systematically analyze all checkers of the widely used static linter Pylint to identify cases that are hard to detect statically, i.e., where a dynamic analysis could complement the existing static checker. The problem mentioned in Figure 1 is an example of such a case. (ii) We search the academic literature on Python-specific bugs and include kinds of problems mentioned in these papers, such as data leakage in machine learning pipelines [46]. (iii) We search for online resources, such as StackOverflow posts mentioning a common mistake or collections of particularly hard to understand Python features. Table 1 gives an overview of the anti-patterns we consider in DyLin. The last column gives a reference to a paper, online resource, etc. that has inspired the anti-pattern.

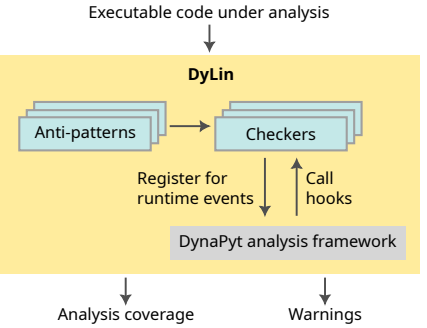

Fig. 3. Overview of DyLin.

For each anti-pattern in Table 1, DyLin has a corresponding dynamic checker. Creating a checker involves three design decisions. First, we must decide which runtime events to track, which is shown in the "Hooks used" column of the table. Second, we must decide what information to gather whenever one of these events occurs. Finally, we must decide when to report a warning about likely incorrect behavior. The following subsections describes our 15 checkers in more detail.

Table 1. Anti-patterns in Python code and DyLin checkers to detect their occurrences.

Anti-pattern	Checker	Hooks used	Reference
Python programming constructs:			
Comparing a value to a function	InvalidFunction-Comparison	`comparison`	PyLint (W0143)
Comparing nearby floats	RiskyFloatComparison	`equal`, `not_equal`	Wikipedia[a]
Type mismatch in list or set	WrongTypeAdded	`pre_call`, `add`, `add_assign`	PEP 484
Mutate list while iterating	ChangeListWhile-Iterating	`enter_for`, `exit_for`, `_break`	PyLint (W4701)
Search for item in a list	ItemInList	`_in`, `not_in`	Eghbali et al. [12]
Inconsistent comparison	InconsistentComparison	`equal`, `not_equal`	Properties of equality
Python standard library functions:			
Unnecessarily copying a list for sorting	InPlaceSort	`pre_call`, `read_identifier`	StackOverflow question[b]
Unexpected `any`/`all` result	AnyAllMisuse	`post_call`	WTFPython[c]
String strip misuse	StringStrip	`post_call`	PyLint (E1310)
Inefficient string concatenation	StringConcat	`add_assign`	Python wiki[d]
Type check with ==	InvalidTypeComparison	`equal`, `not_equal`	StackOverflow question[e]
Relying on non-deterministic ordered set	NondetermisticOrder	`post_call`, `function_exit`	StackOverflow question[f]
Machine learning:			
Data leakage between train and test data	DataLeakage	`post_call`, `function_exit`	Yang et al. [46]
Non-finite values	NonFiniteValues	`post_call`	StackOverflow question[g]
Gradient explosion	GradientExplosion	`pre_call`, `post_call`	StackOverflow answer[h]

a https://en.wikipedia.org/wiki/Floating-point_arithmetic#Accuracy_problems
b https://stackoverflow.com/q/22442378
c https://github.com/satwikkansal/wtfpython/blob/ccf5be/README.md#L1245
d https://wiki.python.org/moin/PythonSpeed/PerformanceTips#StringConcatenation
e https://stackoverflow.com/q/152580
f https://stackoverflow.com/q/9792664
g https://stackoverflow.com/q/61006754
h https://stackoverflow.com/a/69553633

2.2 Checking Python Programming Constructs

We present the checkers in DyLin in three categories, where the first is about anti-patterns related to programming constructs offered by the Python language. These anti-patterns may occur in any

Python program as they are related to the basic constructs of the language, such as function calls, floating point numbers, and lists.

2.2.1 *Comparing a Value to a Function (InvalidFunctionComparison).*

Anti-pattern. Because Python supports first-class functions, developers may accidentally refer to a function without calling it. This mistake is likely to remain unnoticed when it happens in a comparison, because comparing a function to some other value does not raise an exception in Python. The static Pylint tool also warns about comparing values to functions.[1] However, static analysis can only approximately determine whether a value is a function [5].

Checker. DyLin offers a checker that warns about likely incorrect comparisons caused by forgetting to call a function. To this end, the checker uses the `comparison` hook, which gets triggered whenever the program performs a comparison operation. On each comparison, the checker checks the types of both sides of the comparison operator and warns if one side, but not the other, is a function.

2.2.2 *Comparing Nearby Floats (RiskyFloatComparison).*

Anti-pattern. Comparing floating point numbers should be done differently from integers, as inaccuracies in storing values and performing arithmetic operations can cause unexpected results. In particular, when comparing two floating point values that are nearby but not exactly equal, a program may incorrectly conclude the values to be different, even though the difference is simply due to the inherent inaccuracies of floating point computations. Since static analyzers have only limited knowledge of types and values, warning about such risky floating point comparisons is difficult for them.

Checker. DyLin checks the type of the operands of each equality and inequality using the `equal` and `not_equal` hooks. If both values are floating point numbers, they have nearby values, and they are not exactly equal, our checker raises a warning. To determine whether two values are "nearby", we use the `math.isclose` function with a relative tolerance of $1e-8$.

2.2.3 *Type Mismatch in List or Set (WrongTypeAdded).*

Anti-pattern. As a dynamically typed language, Python allows adding values of different types to the same list or set. While type-heterogenous lists or sets may be intended in some programs, they may also be a sign of an unexpected program state. For example, consider a long list of elements that all have the same type T, but then a value of another type $T' \neq T$ gets added.

Checker. DyLin warns about likely unintended type-heterogeneous lists and sets. The checker tracks all additions of values to lists and sets, via the hooks `add`, `add_assign`, and `pre_call`. The first two hooks are used to track operations of the form `someList + ["a"]` and `someList += "a"`. The `pre_call` hook is used to track calls of the `append`, `add`, `extend`, and `insert` methods offered by Python's built-in list and set types. The checker reports a warning if and only if the list or set exceeds a configurable size (default: 100) and contains elements of only a single type, but then a value of another type is added into it.

2.2.4 *Mutate List While Iterating (ChangeListWhileIterating).*

[1] https://pylint.readthedocs.io/en/latest/user_guide/messages/warning/comparison-with-callable.html

Anti-pattern. Lists in Python allow for mutating a list while iterating over it, which is considered bad practice as the semantics are subtle and easily misunderstood. Figure 1 shows a real-world example of this problem. In this example, if two adjacent items in the list satisfy the condition starting in the second line, the first one gets removed from the list, but the second item is skipped in the iteration, which causes unexpected behavior.

Checker. DyLin warns about code that changes the length of a list while iterating over the same list. The checker uses the `enter_for` and `exit_for` hooks, which are triggered just before entering and just after leaving a for-loop. Using the two hooks, the checker tracks the length of the list and warns if the length is decreased during the iteration.

2.2.5 *Search for Item in a List (ItemInList).*

Anti-pattern. The running time to search for an item in a list is linear to the length of the list [12]. However, for sets this time is constant as sets are implemented using hash tables in Python. Therefore, it would be faster to search a set instead of a list if the size is large enough.

Checker. The checker uses `_in` and `not_in` hooks to detect when searching for an item in a collection is happening. If the collection is a list, the length is longer than a pre-defined threshold (default: 100), and the search happens multiple times so that the total length searched is larger than 5 times the default threshold, then it warns that using a set can improve performance. This checker is an improved version of the KeyInListAnalysis by Eghbali and Pradel [12].

2.2.6 *Inconsistent Comparison (InconsistentComparison).*

Anti-pattern. The equality operator should satisfy the following mathematical properties:

- Symmetry: Swapping the operands should not affect the result, e.g., `a == b` should be equal to `b == a`.
- Stability: The comparison operator should not have any side effects that cause consecutive comparisons to vary in result.
- Identity: Comparing any object to `None` should return `False`, except for `None` itself.
- Reflexivity: Comparing an object to itself should always return `True`.

Classes may override special Python functions, such as the equality operator, which can invalidate any of the above properties.

Checker. DyLin warns if at least one of the above properties gets violated. The checker uses the `equal` and `not_equal` hooks, which get called whenever a `==` or `!=` operation occurs, to check the four properties. For example, to check for symmetry, we swap the arguments and inspect the results, and to check for stability, the checker repeats the comparison three times.

2.3 Checking Usage of the Python Standard Library

The issues in this category are either caused by functions in the standard library of Python, or are issues that can be fixed by using functions from the standard library.

2.3.1 *Unnecessarily Copying a List for Sorting (InPlaceSort).*

Anti-pattern. When a list is sorted using the `sorted` function and the original unsorted list is not used later in the execution, then the sorting will be more space- and time-efficient if it is done in-place.

Checker. To detect code that unnecessarily copies a list for sorting, this checker uses the `pre_call` hook to keep track of calls to the `sorted` function. The checker stores the list given as an argument

```
all([]) # True
all([[]]) # False
all([[[]]]) # True
any([]) # False
any([[]]) # False
any([[[]]]) # True
```

Fig. 4. any and all can have unexpected output.

```
file_paths = ["foo.py", "fooy.py", "foo.py.py"]
result = [fp.strip(".py") for fp in file_paths]  # result: ["foo", "foo", "foo"]
```

Fig. 5. Incorrect usage of strip function.

to sorted, and then checks if the list is used afterwards using the read_identifier hook, which gets triggered whenever the program accesses an identifier, e.g., a variable. If the original list, which remains unsorted when calling sorted, is not used later in the execution, the analysis suggests using in-place sort to improve performance.

2.3.2 *Unexpected* any/all *Results (AnyAllMisuse).*

Anti-pattern. Python's any and all functions return values for special cases that can be unexpected for developers not familiar with truth values of nested lists.[2] Some of these special cases are shown in Figure 4.

Checker. DyLin offers a checker that tracks calls to any and all, using the post_call hook. If the flattened list in the argument to any or all is an empty list, but the function returns True, the checker raises a warning. Detecting these issues with static analysis is not practical, because the value during runtime is the only indicator of a potential problem.

2.3.3 *String Strip Misuse (StringStrip).*

Anti-pattern. The strip function for strings takes a string as its parameter, which specifies the set of characters to strip from both ends of the target string. However, according to Pylint's documentation,[3] it is a common error to assume that the function strips the exact string from both ends. For example, consider the code shown in Figure 5, which tries to remove the Python file extensions from a file path. Unfortunately, the implementation is incorrect, as the strip function does not remove the exact string.

Checker. DyLin has a checker that warns about two kinds of likely misuses of the strip function. First, the checker warns when the value of the argument to strip has repeated characters. Second, the checker raises a warning when the argument is a prefix or suffix of the target string, but strip removes more characters from the target string. Both checks are implemented using the post_call hook, which gets triggered once a function has returned. The static analyzer Pylint also has a checker for this kind of problem, but due to the limitations of static analysis, warns only if strip is called with a string literal with at least one repeated character. In contrast, our checker reasons about any argument to strip, even if it is not a literal.

2.3.4 *Inefficient String Concatenation (StringConcat).*

[2] https://github.com/satwikkansal/wtfpython/blob/ccf5be/README.md#L1245
[3] https://pylint.pycqa.org/en/latest/user_guide/messages/error/bad-str-strip-call.html#bad-str-strip-call-e1310

```
- if param.default == param.empty:
+ if param.default is param.empty:
    yield getattr(self, param.name)
```

Fig. 6. Bug in Textualize/rich, showing that types should not be compared with ==.

Anti-pattern. Concatenating strings using the += operators creates a new object because strings are immutable in Python. Although CPython checks whether strings can be concatenated in-place to improve efficiency, other interpreters do not. Moreover, developers should not rely on such optimizations in their programs.

Checker. To detect inefficient string concatenations, DyLin warns if a string concatenation is done using += for more than configurable number of times (default: 1000). In such cases, the code should use the more efficient `"".join(<parts>)`. Using the `add_assign` hooks, the checker counts how often += occurs on the same string, and raises a warning of the threshold gets exceeded.

2.3.5 *Type Check with == (InvalidTypeComparison).*

Anti-pattern. Comparing types using the == operator is discouraged because it does not consider inheritance. Developers should use the `isinstance` function instead. Even for cases where inheritance should be ignored, it is recommended to use the `is` operator.

Checker. DyLin warns whenever values of different types are compared using ==. The checker found a bug in the popular "rich" library, which is shown with its fix in Figure 6. The bug is explained in more detail in Section 3.3.

2.3.6 *Relying on a Non-Deterministically Ordered Set (NondetermisticOrder).*

Anti-pattern. Sets are unordered data structures in Python. However, casting them to a sequence-based type, e.g., a string or a list, results in an ordering of the items. This ordering might be different in different interpreters and executions, which can result in unintended behavior.

Checkers. DyLin warns whenever a set is used with a function that enforces an ordering. We realize this analysis using a simplified dynamic taint analysis, which propagates whether a value enforces an ordering with the `post_call` hook, which is invoked after every function call.

2.4 Checkers Related to Machine Learning

Our final category of checkers targets machine learning code, i.e., one of the most popular application domains of Python.

2.4.1 *Data Leakage between Train and Test Data (DataLeakage).*

Anti-pattern. The preprocessing phase in a machine learning pipeline can leak data from the test set to the training phase [46]. Yang et al. [46] propose a static analyzer to detect data leaks in machine learning pipelines.

Checker. Using a simplified dynamic taint analysis, DyLin checks for data leakage by checking calls to functions (with the `post_call` hook) that are prone to leaking the data during pre-processing. This analysis works by tainting multiple pre-defined transformations that are commonly used, and then tracks if they reach the prediction function.

2.4.2 *Non-Finite Values (NonFiniteValues).*

Anti-pattern. Non-finite values, such as `inf` and `-inf`, often are an unexpected result of an arithmetic operation or a function call. However, when such values are generated, Python does not raise any warning by default.

Checker. DyLin provides a checker that tracks function calls, using the `post_call` hook, and warns about calls that return a non-finite value when their inputs are all finite.

2.4.3 *Gradient Explosion (GradientExplosion).*

Anti-pattern. A common issue in deep neural networks is an explosion of the gradients. This problem hinders the training process and tracking the location of the issue in the source code is non-trivial.

Checker. In the `pre_call` and `post_call` hooks, DyLin checks to see if any of the parameters or the return value have gradients. For those objects that have gradients, if the absolute gradient value exceeds a threshold, the checker raises a warning.

2.5 Example of Checker Implementation

An important property of a successful bug detection framework is the option to easily add support for additional anti-patterns. By building on top of the DynaPyt dynamic analysis framework, the checkers in DyLin are relatively lightweight in terms of their implementation effort. Each of the checkers discussed above is implemented in tens of lines of non-comment, non-empty lines of Python code.

For example, Figure 7 shows the implementation of the *InPlaceSort* checker. The checker implements three hook functions: (i) `post_call`, which gets applied only to the `sorted` function thanks to the `@only` function decorator (described in Section 2.7) and tracks all lists that get sorted and are longer than `threshold = 1000`; (ii) `read_identifier`, which discards any list whose elements were sorted if the original list is used afterwards; and (iii) `end_execution`, which issues a warning at the end of a program's execution if a list given to `sorted` has not been used at all afterwards.

2.6 Analysis Coverage

When developers apply a quality assurance tool, it is useful to measure how much of the code it actually considers. For testing, this is measured using test coverage. We present a similar metric for dynamic analysis, which we call *analysis coverage*. For a program P and a set of checkers C, we define the absolute analysis coverage as the number of lines analyzed by at least one checker:

$$abs\_analysis\_coverage = |\{l \mid l \in P \wedge \exists c \in C.\, analyzes(c, l)\}|$$

where $analyzes(c, l)$ is true if checker c analyzes line l, and relative analysis coverage as the ratio of absolute analysis coverage to the number of executed lines:

$$rel\_analysis\_coverage = \frac{abs\_analysis\_coverage}{|\{l \mid l \in P \wedge executed(l)\}|}$$

where $executed(l)$ is true if line l is executed.

Analysis coverage differs from test coverage, because it not only assesses whether a line gets executed, but whether it is also analyzed by the dynamic analysis. While test coverage provides an insight for developer on how much of their code is tested for behavioral correctness, analysis coverage provide the insight into how much of the tested code is also checked for common programming mistakes. To measure analysis coverage, DyLin records each line that is executed and that triggers a call of a hook implemented by one of the checkers. The approach reports both the overall analysis coverage across all checkers and the analysis coverage for individual checkers.

```python
class InPlaceSortAnalysis(BaseDyLinAnalysis):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.analysis_name = "InPlaceSortAnalysis"
        self.stored_lists = {}
        self.threshold = 1000

    @only(patterns=["sorted"])
    def pre_call(self, dyn_ast: str, iid: int, function: Callable, pos_args, kw_args):
        if function is sorted:
            if hasattr(pos_args[0], "__len__") and len(pos_args[0]) > self.threshold:
                self.stored_lists[id(pos_args[0])] =
                {
                    "iid": iid,
                    "file_name": dyn_ast,
                    "len": len(pos_args[0]),
                }

    def read_identifier(self, dyn_ast: str, iid: int, val: Any) -> Any:
        if len(self.stored_lists) > 0 and type(val) is list:
            self.stored_lists.pop(id(val), None)
        return None

    def end_execution(self) -> None:
        for _, l in self.stored_lists.items():
            self.add_finding(
                l["iid"], l["file_name"], "A-09",
                f"unnecessary use of sorted(), len:{l['len']} in {l['file_name']}",
            )
        super().end_execution()
```

Fig. 7. The analysis code for the *InPlaceSort* checker.

2.7 Improvements of the Underlying Analysis Framework

To ensure the effectiveness and efficiency of DyLin, we augmented the existing DynaPyt framework in two ways.

Name-based selective instrumentation. To keep the overhead imposed by any instrumentation-based analysis manageable, it it crucial to instrument only those source code locations that are actually of interest to the analysis. DynaPyt already provides a form of selective instrumentation, where it selects code locations based on AST node types. However, several of the DyLin checkers need to track only particular program elements, e.g., calls to only a specific built-in function. In the original DynaPyt, such checkers would need to implement one of the `pre_call` or `post_call` hooks, and then check inside the hook for their desired function.

Instead, we augment DynaPyt with a novel filtering technique that allows for selecting program elements to instrument based on their name. The `@only` function decorators in Figure 7 shows an example, where the `pre_call` hook will intercept only calls to functions named “sorted”. With the new selective instrumentation technique, checkers can specify the names of program elements, operands, and parameters as a regular expression. The augmented DynaPyt considers this selection during the instrumentation, by only instrumenting code locations that match the filters, and also when dispatching runtime events to the hooks.

Multi-analysis execution. The original DynaPyt implementation supported only a single analysis to run for each execution. As DyLin consists of multiple checkers, running the program under analysis once for each checker wastes a lot of time. Instead, we modify DynaPyt to run multiple checkers during one execution. The augmented DynaPyt framework takes a list of analyses as its input, and then instruments the code based on all of the analyses. During the execution, whenever a runtime hook is executed, the augmented framework checks and runs the relevant analysis hooks from all analyses.

Analysis coverage. Since our novel metric, analysis coverage, can be applied to all dynamic analyses, we integrated the implementation to DynaPyt. If the option to track analysis coverage is turned on, any DynaPyt-based analysis gets a detailed coverage report. For each covered line, the report shows which analyses and for how many times have covered it.

3 Evaluation

Our evaluation addresses the following research questions:

RQ1 How effective is DyLin in detecting problems in real-world software?
RQ2 How severe are the problems found by the approach?
RQ3 How many of the issues found by DyLin can be found by existing tools or testing?
RQ4 How much of the executed code is analyzed by DyLin's checkers?
RQ5 How efficient is the approach?

3.1 Experimental Setup

We compile three sets of Python programs to evaluate DyLin. The first dataset is a set of *micro-benchmarks*, which are small programs with programming problems that DyLin looks for. This dataset, which is created by the authors, consists of 704 lines of code, with 83 known problems. The main purpose of the micro-benchmark is to serve as a ground truth of known occurrences of the problems DyLin tries to detect, so we can assess the recall of the approach.

The second dataset are open-source Python projects, called *GitHub projects*. We select 37 projects by sampling from highly popular projects in different application domains, such as web applications, machine learning, python utilities, and multimedia processing. While selecting projects we focus on projects that come with executable test suites. To run DyLin in reasonable time, we limit the execution of each test suite to one hour. These projects have between 288 and 342,115 lines of code, with an average of 26,940 and median of 8,519 lines per project.

The final dataset are submissions to Kaggle competitions, called *Kaggle submissions*. Kaggle[4] is a popular website focused on data science. It hosts machine learning competitions and users participate by submitting their predictions for the test set. Some users also submit their code as a Python notebook for others to see. We select three competitions, namely, "Titanic - Machine Learning from Disaster", "Spaceship Titanic", and "Binary Prediction of Poisonous Mushrooms", where the first two are introductory competitions designed for new users, and the third competition is the most recent competition hosted by Kaggle. We download submissions with more than 2 votes up to 50 notebooks per competition using Kaggle's API, and after transforming them to `.py` files, add them to this dataset. The dataset contains 91 files with over 14k lines of code.

We run our experiments in a docker container with Ubuntu 22.04 and Python 3.10. The host machine has a 48 core Intel Xeon CPU with 2.2 GHz per core.

Table 2. GitHub projects and their domains.

Domain	Projects
Text processing	sumy, translate, html2text
Python utilities	rich, sphinx, mashumaro, typer, funcy, python-diskcache, schedule, dh-virtualenv, python-patterns, arrow, pyfilesystem2
Machine learning	keras, adversarial-robustness-toolbox
Distributed applications	keripy, openleadr-python
Data analysis and visualization	clodius, akshare, seaborn, cerberus
Process management	supervisor, delegator.py
CLI tools	thefuck, click, pudb
Automation & Robotics	requests, grab, PythonRobotics
Web	flask-api, uvicorn, pyjwt, wtforms
Video	moviepy
Geo-coding	geopy
Security	gato

Table 3. Number of problems found by each checker in each dataset.

Checkers	Micro-bench	GitHub	Kaggle
Python programming constructs:			
InvalidFunctionComparison	4	1	0
RiskyFloatComparison	8	0	0
WrongTypeAdded	6	10	0
ChangeListWhileIterating	4	3	0
ItemInList	2	11	0
InconsistentComparison	8	15	0
Python standard library:			
InPlaceSort	3	1	0
AnyAllMisuse	4	1	0
StringStrip	3	0	0
StringConcat	7	1	0
InvalidTypeComparison	12	11	0
NondeterministicOrder	12	0	0
Machine learning:			
DataLeakage	3	0	5
NonFiniteValues	4	7	0
GradientExplosion	2	1	1
Total	82	62	6

3.2 RQ1: Effectiveness

DyLin finds multiple issues in each of our three datasets. Table 3 shows that our approach finds all 82 problems in the micro-benchmark, 62 problems in the GitHub projects, and 6 problems in the Kaggle submissions. We count a finding at a specific code location once, even if it is detected multiple times at runtime. The fact that the approach finds all problems in our micro-benchmark shows that each of the checkers is effective. Moreover, finding various problems in real-world code written by others confirms that the problems DyLin looks for occur in practice. A comparison across datasets shows that a wide range of checkers have findings in the GitHub projects, while only the machine learning checkers have findings in the Kaggle submissions. The reason is that Kaggle submission typically have a simple structure and mostly interact with machine learning libraries.

3.3 RQ2: Severity of Detected Problems

To better understand the severity of the detected problems, we manually inspect each warning and classify it into one of the following five categories. The number indicates the severity level, with 4 being the most severe, 1 being the least severe, and 0 indicating the false positives.

4 *Incorrect behavior in an existing execution.* Issues that cause an incorrect behavior and at least one instance of this incorrect behavior happens during an execution analyzed by DyLin. For example, the bug in Figure 2 causes an infinite value during the execution of a test. Another example is the Kaggle submission code in Figure 9, which preprocesses data in a way that causes data leakage from the test set to the training set [46], which undermines the integrity of the results.

3 *Incorrect behavior in a plausible execution.* Issues resulting in incorrect behavior, which do not happen during an execution analyzed by DyLin, but could happen in another, plausible execution. "Plausible" here means that the incorrect execution is in line with the use-case and domain of the analyzed code. For example, the buggy code in Figure 1 is an example of incorrect behavior in a plausible execution.

2 *Performance issues.* Issues that do not affect the correctness of the program, but slow down the running time due to an occurrence of a programming anti-pattern. For example, DyLin detects an inefficient string concatenation in the "translate" project.

1 *Code quality issues.* Issues that neither affect the correctness, nor the running time at the current state of the project. However, these anti-patterns make maintaining the software more difficult, by using bad coding practices. For example, inserting a value with a different type into a type-homogeneous list does not necessarily cause a bug, but it makes future code changes more difficult as different types need to be handled.

0 *False positive.* Findings that do not correspond to any of the above categories. These include issues that are intentionally implemented for testing purposes and issues related to the dependencies of the project under analysis. For example, testing code that passes non-finite inputs is in this category. Apart from the intentionally implemented anti-patterns, some warnings are incorrectly raised because of shortcomings in DynaPyt. One such issue is detected by the *InconsistentComparison* checker, where in the underlying library specifying a filter uses the same notation as comparisons. Another issue is a warning caused by a library incompatible with DynaPyt, for which we notified the developers of DynaPyt.

Figure 8 shows the number of issues in the GitHub dataset by severity category and their report status. We have reported all 42 true positive problems. By combining related problems into a

[4] https://www.kaggle.com/

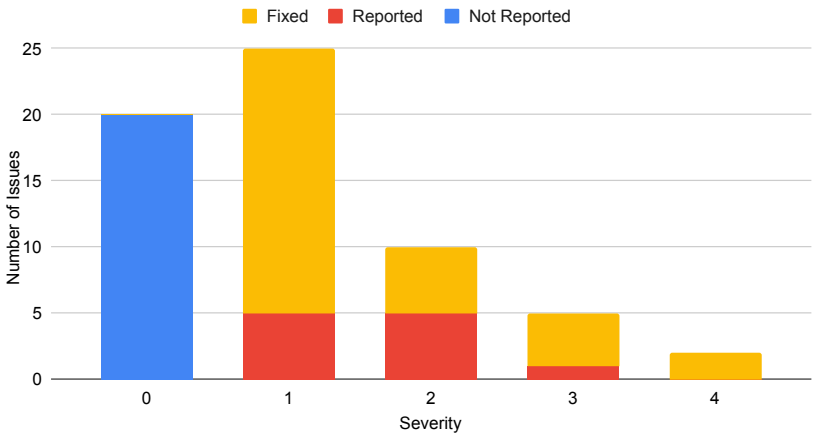

Fig. 8. Number of issues within each severity category, and the status of issue reports to the developers.

```
Impute=SimpleImputer(missing_values=np.nan, strategy="mean")
X_train=Impute.fit_transform(X_train)
data_test=Impute.fit_transform(data_test)
X_train,X_val,y_train,y_val=train_test_split(X_train, y_train,test_size=0.1)
```

Fig. 9. Example of data leakage from test data to training data.

single report, this results in 21 GitHub issues and pull requests. At the time of writing, 31 of the reported problems have been fixed and the fixes got merged into the main branch of the respective repositories. In the Kaggle dataset, all issues are of severity category 4. The data leakage from validation data into training, which is the majority of issues in the Kaggle dataset, causes the model to perform better than expected on validation metrics, but not as well during testing. We did not report issues in the Kaggle dataset because the competitions have ended and it is less likely for their developers to continue to maintain them.

Most of the warnings reported by DyLin are true positives, i.e., severity category 1 to 4. Overall, the precision is 48/68=70.6%, or more precisely, 42/62=67.7% in the GitHub projects and 6/6=100% in the Kaggle submissions. High precision is an advantage of a dynamic linter as it has access to runtime information, such as values, types, and control flow. Notably, this distinguished DyLin from static linting, which often suffers from low precision [21, 22, 40].

3.4 RQ3: Comparison with Existing Tools and Testing

We design DyLin to complement static linters and type checkers with warnings that they cannot detect. To evaluate this hypothesis, we compare the findings of DyLin to the findings by two popular static linters, and one static type checker. At the time of writing this paper, two of the most popular static linters are PyLint[5], and Ruff[6]. We choose Mypy[7] as the type checker, as it is the most popular type checker for Python. We run Ruff with all checkers enabled, i.e., `ruff check --select ALL`. We run PyLint and Mypy with their default settings. Then we match findings by DyLin and these tools

[5] https://pylint.readthedocs.io/en/stable/
[6] https://docs.astral.sh/ruff/
[7] https://mypy-lang.org/

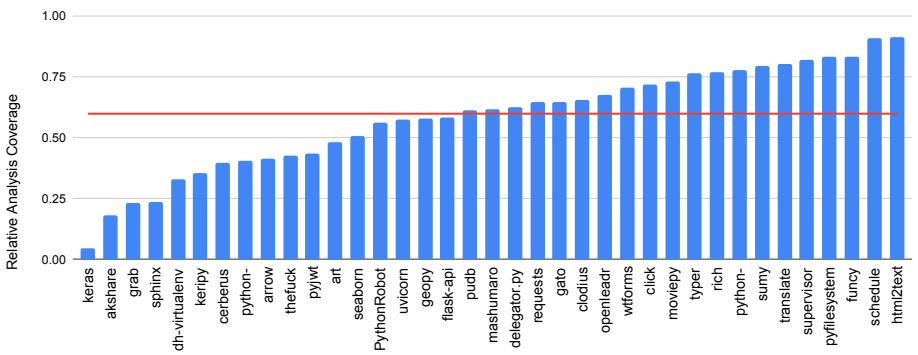

Fig. 10. Analysis coverage for the 37 GitHub projects.

based on the location of issues found and manually check if the finding by a static analysis tool represents the same issue as the problem found by DyLin. We observe that none of the findings by DyLin are found by the checkers in Ruff, PyLint, or Mypy in the GitHub projects and Kaggle submissions. Only five out of the 84 issues in the micro-benchmark are found by Ruff, 16 are found by PyLint, and none of them is found by Mypy. These results show the complementary value of using DyLin alongside static linters and type checkers.

Another approach to finding the issues addressed by DyLin is testing. Because our evaluation applies DyLin to the executions of existing, developer-written tests, a reader may wonder to what extent these tests reveal the same issues as DyLin. Interestingly, none of the issues found by our approach was previously uncovered by the tests. Moreover, the issues that we report and fix have been present in their respective repositories, on average, for 1,416 days, i.e., for over 3.5 years. There are multiple reasons why DyLin finds issues that tests do not, despite exercising the same code. First, tests can reveal only those problems anticipated by the assertions in the test code. DyLin can be seen as an additional test oracle, as it checks for other kinds of problems. Second, a test execution may hint at a particular problem without actually exposing it (see severity level 3 in Section 3.3). Finally, the test cases themselves occasionally are wrong. For example, in the large and well-maintained Keras project, a condition was implemented as `backend.backend == "tensorflow"` instead of `backend.backend() == "tensorflow"` inside a test function, and remained so for over a year. Overall, the comparison with testing shows that our dynamic linter complements state of the art testing by checking test executions against general anti-patterns.

3.5 RQ4: Analysis Coverage

To assess how many of all executed lines are actually analyzed by DyLin, we apply our novel analysis coverage metric (Section 2.6). Figure 10 shows the analysis coverage for each project in the GitHub projects dataset. Because the analysis coverage depends on how much of the executed behavior is in scope for at least one of DyLin's checkers, the coverage varies across projects. For many projects, the approach analyzes a large portion of the executed code, with an average analysis coverage of 59.9%. Similar to the way that traditional test coverage is used as an estimate of testing effectiveness, this metric can be used to estimate the amount of code quality checks performed on top of the existing test suite by DyLin. The coverage is lower for some projects, which is partly due to tests' timeout while running DyLin and measuring analysis coverage.

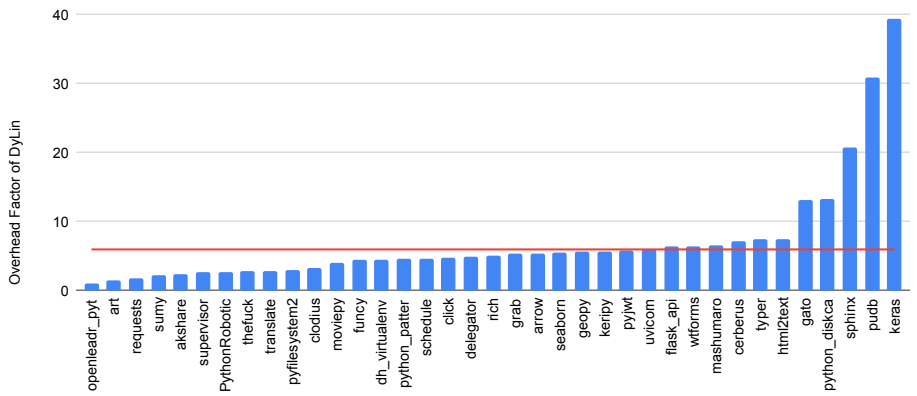

Fig. 11. Overhead factor of DyLin.

3.6 RQ5: Efficiency

To assess the overhead imposed by DyLin, we measure the time it takes to run the original test suite of each project, and the time it takes to run the test suite with all checkers of DyLin enabled. All the measurements are done using Python's `time.time()` function. To ensure the consistency of the time measurements, we run the test suite of the ten smallest projects (in terms of lines of code) with all DyLin checkers enabled ten times each. The coefficient of variation across the ten repeated runs is 1.21%, on average, and at most 2.16%. Because each run is independent of the others, we observe no statistical difference between the initial runs and later runs.

Overall, as shown in Figure 11, the runtime overhead imposed by DyLin is about 6x, on average across all projects. While this overhead is likely too high for applying DyLin in all production scenarios, we consider it acceptable for performing checks as part of a continuation integration pipeline, e.g., during selected runs of a regression test suite, which is where lint-style tools are commonly deployed. To validate the practicality of DyLin in continuous integration, we develop a reusable GitHub action that can be easily integrated into the regression tests performed by an existing GitHub workflow. This deployment has successfully revealed some of the issues reported earlier, with the maximum running time on the free-tier GitHub-hosted machines being 2 hours and 12 minutes. Future improvements can reduce this time by only analyzing changed code. Moreover, maintainers can save time by only running a subset of checkers based on their needs.

4 Threats to Validity

There are several factors that may influence the validity and generalizability of our results. First, as for any empirical result, the selection of benchmarks may influence the results. We mitigate this threat by considering two datasets of real-world code, including various popular and widely used open-source projects, as well as a collection of Kaggle submissions. Second, due to its dynamic nature, DyLin is unable to analyze all code but is constrained by the executions it analyzes. This affects our results in two ways. On one hand, non-deterministic executions of the code under analysis can affect the reproducibility of warnings. On the other hand, our evaluation uses existing regression test suites, which – by definition – exercise code that is well tested. That is, our results may underestimate the set of warnings DyLin could find on code that is not covered by a regression test suite, e.g., when triggering executions via a test generator or when applying DyLin to a production run. Finally, our inspection of warnings and the categories of severity levels may be

subjective. To mitigate this threat, we carefully discuss the levels and the inspection results among the authors. Moreover, we report all true positive findings to the respective developers, and 73.8% of them have already been fixed in response to our report.

5 Related Work

Bug Detection in Python. The widespread use of Python has led to a wide range of tools and techniques for detecting bugs in Python. Most of the existing technique focus on scalable static analysis, e.g., the lint-like static checkers Pylint and flake8, or gradual type checkers, such as Mypy and Pyre. CodeQL [4, 19] is a general-purpose static analysis framework that also provides a range of checks to detect bugs in Python. In addition to the above general-purpose bug detectors, several more specialized techniques have been proposed, including learning-based bug detector CMI-Finder [6], a lightweight static analysis to find instances of 17 test-related code smells in Python [41], and a static analysis that tracks shape of tensors across library calls and warns about potential mismatches [23]. DyLin complements these approaches by offering a dynamic analysis that can detect bugs that are difficult or impossible to find statically.

Dynamic Analysis for Python. Besides the many static analyses for Python, there also are several dynamic analyses, e.g., to slice programs [8], to detect type-related bugs [43], and to enforce differential privacy [1]. DyLin differs from these approaches by offering a general-purpose, dynamic linting framework and a diverse set of, so far, 15 checkers built on top of it. LExecutor [38] is an approach for learning-guided execution, which enables dynamic analysis by injecting otherwise missing runtime values via machine learning. Pynguin [26] offers a unit-level test generator for Python. Both LExecutor and Pynguin could be combined with DyLin, where the existing tools create and enable executions of the code under analysis, while DyLin checks for occurrences of common programming anti-patterns exposed by these executions.

Other Work for Python. Several techniques for supporting Python developers have been proposed. These include type prediction [2, 32, 44, 45], which is motivated by the lack of mandatory static type annotations, inferring dependencies [48], and repairing exceptions due to type errors [31]. Dilhara et al. [10] propose an approach for mining recurrent code changes and show how it applies to Python. Finally, there are several studies on Python, including a study of type annotations [17], a study on the performance benefits (or lack thereof) of using Python idioms [49], and a study of gradual type systems for Python [35]. All this work shows a growing interest in techniques that support Python developers, to which our work contributes the first dynamic linter for Python.

Dynamic Analysis for Other Languages. Beyond Python, various dynamic analyses have been proposed, e.g., to detect type-related problems [3, 34], concurrency bugs [13, 30, 36], and other common bug patterns [15, 16]. Compared with the closely related work on DLint [16], DyLin targets a fundamentally different language (Python vs. JavaScript), and hence, offers different checkers that address different anti-patterns. The design and implementations of the two languages are so different that the anti-patterns cannot just be reused. For example, all anti-patterns in Table 1 of DLint [16] are related to object prototypes, which do not exist in Python. Inversely, none of the "Python standard library" anti-patterns in DyLin are applicable to JavaScript. Moreover, our work contributes novel technical ideas, e.g., how to reduce instrumentation effort via name-based filtering, and a metric to assess analysis coverage. Other analyses focus on finding optimization opportunities [39, 42], infer API usage protocols [33, 47] and input grammars [20], or perform security-oriented analyses, e.g., taint analysis [9] and detectors of similar functions [11]. The breadth and depth of such prior work underlines the importance of dynamic analysis.

Dynamic Analysis Frameworks. Because building a dynamic analysis from scratch is a complex and time-consuming task, several frameworks have been proposed. There are frameworks for most widely used languages, such as Jalangi [37] for JavaScript, Wasabi [24] for WebAssembly, DynamoRIO [7], Pin [25], and Valgrind [29], which all target x86 binaries, DiSL [27] for Java, and RoadRunner [14], which specifically targets concurrency-related dynamic analyses. This work builds upon the recently released DynaPyt [12], which takes care of the subtleties of instrumenting Python code, and hence, ensures that DyLin can be easily extended with new checkers.

6 Conclusion

This paper presents the first dynamic linter for Python. The core of our contribution is a set of programming anti-patterns that are hard or impossible to detect with static analysis, along with a set of dynamic checkers to detect their occurrences. During an evaluation with real-world code, DyLin finds 48 previously unknown true positives, both in popular open-source projects and in machine learning code submitted to Kaggle. Due to its dynamic nature, DyLin complements existing static analysis techniques, providing a valuable addition to the toolbox of Python developers.

7 Data Availability

Our source code, all experimental scripts and required datasets, and the list of pull requests and bug reports are available at https://anonymous.4open.science/r/DyLin-7FFA.

References

[1] Chike Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–15. https://doi.org/10.1109/CSF51468.2021.00043

[2] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*. 91–105. https://doi.org/10.1145/3385412.3385997

[3] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby.. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 459–472.

[4] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[5] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A Benchmark of Executable Python Software. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

[6] Islem Bouzenia and Michael Pradel. 2023. When to Say What: Learning to Find Condition-Message Inconsistencies. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 868–880. https://doi.org/10.1109/ICSE48619.2023.00081

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[8] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

[9] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 196–206.

[10] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *ICSE*.

[11] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 303–317.

[12] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[13] Cormac Flanagan and Stephen N. Freund. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 256–267.

[14] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 1–8.

[15] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 357–368.

[16] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[17] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *ESEC/FSE*.

[18] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 317–328.

[19] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 2–27. https://doi.org/10.1007/11785477_2

[20] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 720–725.

[21] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[22] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we?. In *Proceedings of the 44th International Conference on Software Engineering*. 698–709.

[23] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming, ECOOP*, Vol. 166. 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[24] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ASPLOS*.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[26] Stephan Lukasczyk. 2019. Generating Tests to Analyse Dynamically-Typed Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1226–1229. https://doi.org/10.1109/ASE.2019.00146

[27] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel (Eds.). ACM, 239–250. https://doi.org/10.1145/2162049.2162077

[28] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 532–543.

[29] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 89–100.

[30] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 167–178.

[31] Wonseok Oh and Hakjoo Oh. 2022. PyTER: Effective Program Repair for Python Type Errors. In *ESEC/FSE*.

[32] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 209–220. https://doi.org/10.1145/3368089.3409715

[33] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *International Conference on Automated Software Engineering (ASE)*. 371–382.

[34] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*.

[35] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *DLS*.

[36] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15, 4 (1997), 391–411.

[37] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 488–498.

[38] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. 1522–1534. https://doi.org/10.1145/3611643.3616254

[39] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 607–622.

[40] Kristín Fjóla Tómasdóttir, Mauricio Finavaro Aniche, and Arie van Deursen. 2020. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Trans. Software Eng.* 46, 8 (2020), 863–891. https://doi.org/10.1109/TSE.2018.2871058

[41] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A Test Smell Detector For Python. In *ASE*.

[42] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 419–430.

[43] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 121–132. https://doi.org/10.1145/2950290.2950357

[44] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 607–618. https://doi.org/10.1145/2950290.2950343

[45] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *ICSE*.

[46] Chenyang Yang, Rachel A Brower-Sinning, Grace Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[47] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining temporal API rules from imperfect traces. In *International Conference on Software Engineering (ICSE)*. ACM, 282–291.

[48] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. In *ICSE*.

[49] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. In *ICSE*.