Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

A Dynamic Analysis-Based Linter for Python

Felix Burk

Course of study:	Software Engineering
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Aryaz Eghbali
Started:	October 10, 2022
Completed:	April 15, 2023

University of Stuttgart
Germany

SOLA
SoftwareLab

Abstract

The rising popularity of the Python language yields a wide range of programs, spanning across a large assortment of domains. Common domains include, data science, machine learning, web development, as well as database and networking applications. To implement such applications properly, developers rely on a range of analysis tools which assist them in terms of correctness, performance, and code quality. Both in academia and in the industry analysis tools have been proposed. However, general purpose analyses of Python programs often rely on static analysis, which is inherently limited in analysing dynamic languages, such as Python.

This work presents an approach to analyse Python programs during their execution. As a result, our analyses are able to infer additional information, such as the exact control flow, the precise value, and type of each object allocated. We directly address the dynamic nature of the Python language by applying dynamic analysis concepts to realise a general purpose linter. We present a total of 22 rules to analyse general Python programs, which address correctness, performance and code quality. Moreover, we supplement those rules with 10 additional machine learning specific rules, which analyse the usage of popular machine learning libraries within Python's ecosystem, such as scikit-learn and TensorFlow. Additionally, this work includes a novel approach to track and annotate objects observed during the execution, facilitating the future development of dynamic analysis tools.

We evaluate our prototype by analysing submissions to the Kaggle platform, which include a wide range of programs related to data science and machine learning. Our findings include at least 4 violations with high risk regarding correctness. Additionally, we evaluate general purpose rules on 11 real world GitHub repositories of various domains. Our results include 8 medium risk rule violations and 1 rule violation with high risk regarding correctness. Most rules imposed by our prototype report few false positives. Furthermore, the overhead is at most a 7x increase, which we regard as as acceptable for practical use. Overall, the results show that dynamic analysis yields promising results for both general purpose and machine learning applications within Python's ecosystem.

Kurzfassung

Die weiterhin zunehmende Popularität der Programmiersprache Python bringt eine große Menge an Programmen hervor, welche breite Bereiche abdecken. Zu diesen gehören Data Science Applikationen, maschinelles Lernen, Web-, Datenbanken- und Netzwerkapplikationen. Damit diese Applikationen den erforderlichen Standards entsprechen, verwenden Entwickler eine Fülle von Programmen, welche Ihnen in den Bereichen Korrektheit, Performanz und Code Qualität Unterstützung leisten. Viele solcher Programme wurden sowohl im akademischen Bereich, als auch von der Industrie vorgestellt. Jedoch beruhen die Meisten auf den Prinzipien der statischen Analyse, welche grundsätzlichen Einschränkungen unterliegen, besonders bei dynamischen Programmiersprachen wie Python.

Diese Arbeit stellt einen Ansatz vor, Python Programme während der Ausführung zu analysieren. Aufgrund dessen, erlaubt es unser Ansatz zusätzliche Informationen über ein gegebenes Programm zu sammeln. Diese beinhalten den exakten Kontrollfluss des Programms, den Inhalt von Variablen und Attributen, und deren Typen. Wir adressieren damit direkt die dynamische Natur der Python Programmiersprache, indem wir Prinzipien der dynamischen Analyse anwenden. Diese Arbeit präsentiert 22 allgemeine Regeln für Python Programme, welche die Korrektheit, Performanz und Code Qualität des jeweiligen Programms analysieren. Weiterhin stellen wir 10 Regeln zur Analyse der Nutzung von Bibliotheken im Bereich des maschinellen Lernens vor. Um die zukünftige Entwicklung von dynamischen Analysen zu erleichtern, stellen wir außerdem einen neuen Ansatz vor, wie zusätzliche Informationen zu Objekten während der Laufzeit gespeichert und verwendet werden können.

Wir evaluieren unseren Ansatz anhand von Einreichungen für die Plattform Kaggle. Diese beinhalten Python Programme im Bereich des maschinellen Lernens und Data Science. Unsere Befunde beinhalten mindestens 4 Regelverstöße mit einem hohen Risiko zur Verletzung der Korrektheit des Programms. Zudem evaluieren wir 11 Programme, aus unterschiedlichsten Bereichen, von der Plattform GitHub, welche in der Praxis verwendet werden. Unsere Resultate beinhalten 7 Befunde mit einem mittelgroßen Risiko bezüglich der Korrektheit und einen Befund mit hohem Risiko. Insgesamt zeigen unsere Befunde, dass dynamische Analyse von Python Programmen vielversprechende Resultate liefert, sowohl im Bereich des Maschinelles Lernen, als auch im Allgemeinen.

Contents

1 Introduction **13**

2 Background **17**

2.1 Static & Dynamic Analysis . . . 17

2.2 DynaPyt . . . 18

3 Approach **21**

3.1 Selection of Rules . . . 21

3.2 Analyses & Rationales Behind Them . . . 23

3.3 Machine Learning Analyses & Rationales . . . 34

3.4 Executing Analyses & Collecting Findings . . . 39

3.5 Testing Analyses . . . 39

3.6 Identifying and Analysing Objects on the Heap . . . 40

3.7 Object Marking . . . 43

4 Implementation **47**

4.1 Overview . . . 48

4.2 Reporting of Findings . . . 48

5 Evaluation **51**

5.1 Experimental Setup . . . 51

5.2 (RQ1) Prevalence of Rule Violations . . . 53

5.3 (RQ2) Severity of Rule Violations . . . 56

5.4 (RQ3) Runtime Overhead . . . 60

5.5 Threats to Validity . . . 61

5.6 Discussion . . . 61

6 Related Work **63**

6.1 Analysis of Python Programs . . . 63

6.2 Works on Dynamic Analysis . . . 64

7 Conclusion **65**

Bibliography **67**

List of Figures

2.1 DynaPyt Overview 19

3.1 Heap Mirror 40
3.2 Example Marking Propagation 44

4.1 Overview of the Approach 47

5.1 Analyses Overhead 61

List of Tables

3.1 List of General Python Rules 23
3.2 List of Rules which Address Security 23
3.3 List of Machine Learning Rules 34

5.1 Prevalence of Machine Learning Rule Violations Kaggle Submissions 53
5.2 Analysed GitHub Repositories 54
5.3 General Rule Violations of GitHub Repositories 54
5.4 Machine Learning Rule Violations of GitHub Repositories 55
5.5 Severity of Rule Violations Kaggle Submissions 56
5.6 Severity of Rule Violations Github Repositories 57

List of Listings

1.1 Example difficulty when applying static analysis 13
1.2 Example finding from the sumy repository 15
2.1 Example DynaPyt Analysis Class 18
3.1 Proper File Closing Example 24
3.2 Potentially Unintended Result For any / all Function Calls 25
3.3 Compare with Function Example 26
3.4 Example In List Mismatch Type 28
3.5 Example Mutable Default Arguments 29
3.6 Example Unintended Results for string.strip 30
3.7 Inconsistent Set Ordering Example 32
3.8 Examples of Changing a List While Iterating On It 32
3.9 Faulty Cosine Similarity 34
3.10 Faulty Cross Entropy 35
3.11 Data Leakage Example 36
3.12 Inconsistent Preprocessing Example 38
3.13 Object Marking Definition for Weak Hash Analysis 43
4.1 Example finding from the python-telegram-bot repository 49
5.1 Issue with High Risk in rich Repository 58
5.2 Issue with Medium Risk in Adversarial-Robustness-Toolbox Repository 59
5.3 Issue with Medium Risk in python-telegram-bot Repository 59
5.4 Issue with Medium Risk in translate Repository 60

1 Introduction

According to Stack Overflow's most recent survey on popularity of programming languages, Python is the fourth most popular language for professional developers and the third most popular for people learning to code [22]. Due to its popularity a plethora of frameworks, modules, and tools exist to allow developers to build a wide variety of applications. This is also reflected in Stack Overflows's 2022 survey [22], where more than 4 of the most popular frameworks and libraries exist inside Python's ecosystem, including Numpy [HMW+20], TensorFlow [MAP+15], Pandas [tea20], and scikit-learn [PVG+11]. These libraries are utilized heavily in scientific computing and machine learning. Moreover, Python is used in a great range of diverse domains, such as Systems Programming, Database Programming, Network Programming, and more, as described by Srinath [Sri17].

To prevent issues in Python code, several tools have been proposed in academia [CCX14; XLZX16], including, DDUO [ASDN21], and Pythia [LDG+20], from open source developers, e.g., MyPy [23e] flake8 [23c], PyFlakes [23d], and PyLint [23g], or companies, including, Pyre [23i], and Pysa [BC22]. Tools range from static analyses which find code smells, bad practices, and bugs, to more sophisticated tools, e.g., predictive analysis engines. However, none of these tools address difficulties regarding Python's dynamic nature, as described by Srinath [Sri17], directly. By evaluating rules during the execution of a given Python project under analysis, additional information can be obtained and leveraged to evaluate a greater range of rules, compared to static analysis tools often utilized in practice, such as Pylint [23g] or flake8 [23c]. Dynamic analysis is able to gather information about the project's exact control flow and its values by accessing both during the execution of the program.

Listing 1.1 Example difficulty when applying static analysis

```python
class JSON:
  def write_file(self, input_file):
    self.file_contents = write_file.write(self.content)
    input_file.close()

class XML:
  def write_file(self, input_file):
    self.file_contents = input_file.write(self.content)

def write(x):
  input_file = open("testfile", "w")
  x.write_file(input_file) # Which function is called?
```

An example can be found in Listing 1.1, where developers introduce two classes, called `JSON` and `XML`. Both implement a method called `write_file`, with a file instance as an argument, called `input_file`. If a static analysis tool employs a rule which enforces that files should be closed properly, it has to approximate which function is called in Line 12. The rule ensures that no partial writes to disk occur, as described by [23j]. However, due to Pythons dynamic nature, it is non trivial to infer the type of `x` without executing the given program, as explained by Bleaney and Cepel [BC22]. Because dynamic analysis executes the given program, both the exact control flow and values of objects can be observed. Therefore, a dynamic analysis tool observes directly if Line 7 or Line 2 is invoked. If `write_file` in the `JSON` class is executed, no issue should be reported because all files are closed properly. Invocations of Line 7 might close the file outside the method in a different part of the execution flow.

Recent work by Eghbali and Pradel [EP22] introduces DynaPyt, a dynamic analysis framework, that allows the development of dynamic analyses for Python programs. Our approach leverages DynaPyt to propose a dynamic analysis-based linter for Python. Due to the dynamic analysis approach, our linter, called DyLin, is able to evaluate precisely the control flow and values of all objects shown in Listing 1.1. DyLin employs a wide range of rules, regarding both general and machine learning applications written in Python, which benefit from the advantages of dynamic analyses.

An example for a machine learning specific rule follows the observations of Wang et al. [WWC+23], that a large number of bugs in deep learning projects stem from missing or repeated data normalization, which can lead to non finite values. We analyse such issues by notifying developers whether non finite values occur after calculations have been applied during the execution. Moreover, because such values can propagate throughout the application, we observe whether such values occur during comparison operations.

A real world example of such a normalization issue has been found by our approach in the sumy GitHub repository, which automatically summarizes texts. An excerpt from the repository is shown in Listing 1.2, it approximates the largest eigenvalue of a square matrix, as described by Kong et al. [KSB20]. However, due to a lack of normalization in Line 8, `p_vector` overflows during the iteration, and the result produces a vector with both `inf` and `NaN` values. The problem can be fixed, by normalizing `next_p` in Line 8, e.g. by diving by `np.linalg.norm(np.dot(matrix,p_vector))`.

When applying DyLin to the sumy repository by executing its' test suite, DyLin reports 4 issues for the excerpt, as illustrated in Listing 1.2. Three of them are violations of rule M-32, in Line 8, Line 9, and Line 10, where non finite values occur both in the arguments of the invoked functions and in their returned values. Moreover, an issue is reported for Line 7, where non finite values are utilized in a comparison operation. This example illustrates how additional information obtained though dynamic analysis can benefit analysis tools in reporting in depth information about a given program under analysis. By applying fine granular rules to given applications, detailed reports can be created, which allow developers to see exactly what happens during the execution, and why an issue might occur.

[1]https://github.com/miso-belica/sumy/blob/5f271e9679bfdd53d0680d47b3cdf1f0d2de5873/sumy/summarizers/lex_rank.py#L159

Listing 1.2 Example finding from the sumy[1]repository, slightly altered for illustration purposes

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

    while lambda_val > epsilon: # M-31 NaN utilised in comparison
        next_p = np.dot(transposed_matrix, p_vector) # M-32 returns NaN and NaN in argument
        tmp = np.subtract(next_p, p_vector) # M-32
        lambda_val = np.linalg.norm(tmp) # M-32
        p_vector = next_p

    return p_vector

matrix = np.array([
    [0.1,  0.2,   0.3,  0.6,   0.9],
    [0.45, 0,     0.3,  0.6,   0  ],
    [0.5,  0.6,   0.3,  1,     0.9],
    [0.7,  0,     0,    0.6,   0  ],
    [0.5,  0.123, 0,    0.111, 0.9],
])
power_method(matrix, 0.1)
>>>> [inf inf nan inf inf]
```

Approaches which utilize dynamic analysis include DLint [GPSS15] and TypeDevil [PSS15]. Both address the dynamic nature of JavaScript and show impressive results. However, as of yet, no such approach is available for Python. This work leverages the properties of dynamic analyses to address a range of issues which can occur both in general and machine learning Python programs by, (1) Gathering general rules which leverage the strengths of dynamic analysis to find issues in Python programs, (2) Supplementing the rules with additional ones, specific to machine learning libraries, (3) Additional concepts, which support dynamic analyses for Python programs, and (4) A real world evaluation on more than 500 Kaggle Submissions and 11 GitHub repositories.

2 Background

In the following, we discuss how static and dynamic analysis tools function, their differences, and advantages of dynamic analyses over conventional static analysis tools. Furthermore, we provide examples in which cases dynamic analysis tools can infer additional information. Afterwards, we introduce DynaPyt, as described by Eghbali and Pradel [EP22], and how it can be leveraged for our approach.

2.1 Static & Dynamic Analysis

Static analysis tools examine code and infer information about it without execution, as described by Ayewah et al. [APH+08]. Such tools aide programmers in providing rules which validate whether given correctness, performance or code quality assumptions are met. According to Gulabovska [Gul], techniques to implement static analysis tools include, pattern matching, Abstract Syntax Tree (AST) matchers, and symbolic execution. Popular Python static analysis tools include Pylint [23g], flake8 [23c], Mypy [23e], and Pyre [23i]. All of them provide a plethora of rules to guide developers while programming. Example rules include, type checking, encouraging proper documentation, detection of unused arguments or variables, code quality checks, and possible misspellings of commonly used builtin functions.

A typical example rule, is checking whether file objects are closed properly. Python's documentation explicitly encourages this behavior to prevent partial writes to disk [23j]. However, because static analysis does not execute the program, approximations about the execution have to be made, as pointed out by Landi [Lan92]. Due to such approximations static analysis tools tend to generalize the data flow [SC07]. Therefore, static analysis tools may over or underestimate the functionality of the given code. Following our example, of analysing closing files properly, Listing 1.1 illustrates the limitations of static analysis. Line 10 shows a function, called `write`, which utilizes two objects, a file and one which either contains an object of type `JSON` or `XML`. The behavior of both classes differs in their particular `write_file` implementation. The first function, pictured in Line 2 closes the file after it writes its contents. However, the second function, shown in Line 7, does not close the file. If `write` is only called with objects of type `JSON`, no issue should be reported. A static analysis tool now has to approximate whether `write` is called with an object of type `JSON` or `XML`.

Dynamic analysis tools execute the given code and therefore do not need to approximate its control flow, objects' types or their particular values [Ern; SC07]. Such tools are able to examine the exact behavior during runtime. Following the example in Listing 1.1, dynamic analyses can infer both the type of `x` and the control flow, revealing which function is utilized, and thus if the file is closed properly when executing `test`. Due to its dynamic nature, dynamic analysis is of particular interest for Python, similar to JavaScript, as pointed out

Listing 2.1 Example DynaPyt Analysis Class

```python
class ExampleAnalysis(BaseAnalysis):
    def string(self, dyn_ast: str, iid: int, val: str):
        return "example"
```

by Sen et al. [SKBG13]. Because of these these advantages, dynamic analyses may complement existing static analysis tools [Ern; GPSS15]. This work utilizes the general-purpose framework DynaPyt [EP22], to implement rules which leverage the strengths of dynamic analyses, while addressing shortcomings of existing static analysis tools.

2.2 DynaPyt

DynaPyt, as described by Eghbali and Pradel [EP22], is a general-purpose framework for dynamic analysis of Python programs. In contrast to CPython's [Ros23] `sys.settrace`, it allows selecting fine grained events based on 97 hooks. For example, `enter_control_flow`, allows analyses to get notified as soon as the application to analyse enters a control flow. This allows developers to select hooks on source code level. Additionally, DynaPyt analyses can be faster, compared to `sys.settrace` [EP22].

Figure 2.1 illustrates how DynaPyt works. The first step is to execute the instrumentation of a given Python program. This changes the source code of the original implementation to include DynaPyt specific hooks back into its runtime. Consider the line `print("hello world")` and an analysis which is interested in any given string literal of the given program. DynaPyt's instrumentation checks which hooks are implemented by the given analysis and structures the instrumentation accordingly. Listing 2.1 shows an example analysis class, which implements the `string` hook. Therefore, each string literal in the original program is replaced by DynaPyt's `_str_` function, as pictured in Figure 2.1. If the runtime then receives the callback from `_str_`, it calls the corresponding hook function, i.e. `string` in our example, of the analysis class. The hook function contains additional arguments, shown in Line 2. Hooks provide additional information to analyses and allow analysis developers to extract further information for their particular needs.

The left side of Figure 2.1 shows the files produced by the instrumentation. First, the original Python code is kept in a `*.py.orig` file, to allow analyses to extract information from its abstract syntax tree. Second, the instrumented file which will be executed by DynaPyt's runtime. And last, a JSON file which allows mapping of unique instruction ids, in short iids, back to the original source code.

The runtime executes the instrumented Python file and receives callbacks from each hook. If the selected analysis implements the corresponding method, the runtime calls it during execution. Moreover, because DynaPyt replaces parts of the source code directly, analyses are able to replace values during the execution. In the example program, illustrated in Listing 2.1, the analysis is able to replace the string value which is printed by returning a value from its `string` method. The runtime then relays the value to the program currently under analysis. Furthermore, DynaPyt provides control flow hooks, which allows analyses to directly influence the control flow during the execution.

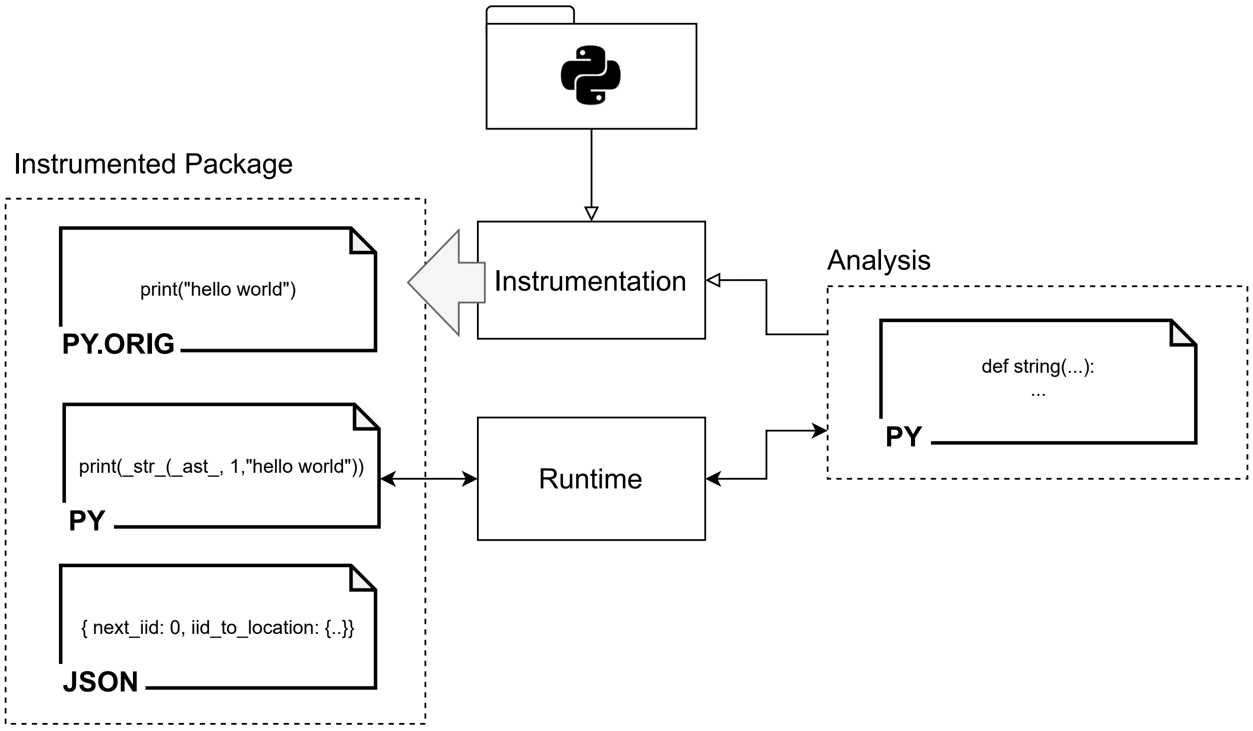

Figure 2.1: This figure shows the process of how DynaPyt achieves dynamic analysis for arbitrary Python programs. First, a Python package to analyse is selected, pictured on the top. Afterwards, the package is instrumented via DynaPyt's instrumenter. The instrumenter produces three files for each `*.py` file of the original program. A JSON file, which allows mapping from iids to the location in the original file, an instrumented version, and a copy of the original Python file. Next, the runtime executes the instrumented file and relays hooks back to the given analysis class, shown on the right.

This work utilizes DynaPyt by building a variety of analyses, which implement DynaPyt's hooks. Furthermore, we provide a set of tools which can be reused by other developers seeking to implement dynamic analyses for Python programs, as described in Chapter 4. These tools address common use cases, such as executing multiple analyses at once, testing analyses, collecting issues, and storing additional information for objects encountered during the overall execution.

3 Approach

In the following, we present how dynamic analysis can be leveraged to analyse Python projects, with regard to correctness, performance, code quality, and security. We begin by explaining how we select a range of different rules suitable for dynamic analysis in Section 3.1. Next, we provide the rationale of each of the selected rules and describe an example of how they might occur, for general rules in Section 3.2, and for machine learning rules in Section 3.3. Afterwards, in Section 3.4, we show how the analysis is executed and issues are collected. To evaluate analyses, we introduce a test suite in Section 3.5. Next, Section 3.6, describes how additional information for any given object encountered during the execution can be managed and stored. Lastly, we derive a high level analysis which can be adapted to the needs of a given Python project, depending on security requirements and correctness rules in Section 3.7.

3.1 Selection of Rules

The aim of this work is to analyse potential problems in Python code and to report them to developers in a way that makes it possible to identify where the issue occurs. Because Python already has a plethora of tools to analyse issues, such as static code analysis tools, e.g., Pylint [23g], or querying tools, e.g., CodeQL [23b], this work presents DyLin, a Python package that leverages the properties of dynamic analyses by either extending existing rules or by creating new rules that are useful to analyse in a dynamic analysis context.

Consider CodeQL's [23b] rule regarding closing files properly[1]. The query is able to analyse if a file is closed in the current scope. However, some developers choose to implement functions, which return the result of calling `open`. An example, adapted from the hug repository, can be found in Listing 3.1. Static analysis tools such as CodeQL cannot detect in the above example if the file object will be closed at some point during the execution because the control flow of the application may be non trivial to analyse statically. However, dynamic analysis tools can keep a reference to each opened file during the execution. With that information we can check at the end of the execution if the opened file has been closed properly. Therefore, we can analyse this rule in cases which cannot be detected by CodeQL.

[1] https://codeql.GitHub.com/codeql-query-help/python/py-file-not-closed/

Other rules are selected based on the programming recommendations of PEP-8 [Gui01]. For example, PEP-8 indicates reflexivity rules are assumed by Python interpreters, regarding the operators < and >. Therefore, we implement a range of checks which ensure reflexivity, in case developers overload operators which are commonly assumed to be reflexive. Moreover, we extend the recommendation by evaluating whether additional consistency rules apply, explained in Section 3.2.4.

Another source is PythonWTF's [Kan22] GitHub repository. The repository contains a wide range of Python code samples with unexpected behavior, from which allows us to derives general rules. For example, changing a lists length by adding or removing items while iterating on it can lead to unexpected changes to the list and to shorter iterations. Listing 3.8 shows a few examples, regarding such behavior. Corresponding analyses are described in Section 3.2.15.

The documentation of scikit-learn [PVG+11] includes a section explaining common pitfalls. We selected two which we found can be analysed during execution time and might be hard to detect by existing tools if non trivial control flows are used in the program to analyse. Yang et al. [YBLK22] analyse data leakage with a static analysis tool by utilizing Datalog rules and statements. The issue of data leakage is part of the named common pitfalls by scikit-learn's documentation. Our approach to detect such issues during execution without additional heuristics is shown in Section 3.3.3.

Wang et al. [WWC+23] conducted a study on numerical bugs in deep learning programs. More than 50% of code relevant bugs they analysed are categorized to be of Invalid Range, e.g., dividing by zero or calculating the derivative of $\sqrt{0}$. Furthermore, more than 42% of code relevant bugs they analysed are gradient explosions. Gradient explosions may occur if a gradient has been calculated with an error, which continues to accumulate through the model during the training process. This can lead to severe training difficulties and limits the depth to which networks can be trained [PSC18]. Another study conducted by Di Franco et al. [DGR17] found that 25% of numerical bugs they analysed exist due to non-finite values. Both [WWC+23] and Di Franco et al. [DGR17] indicate a need for tooling which detects such bugs. Therefore, DyLin implements a variety of analyses regarding deep learning programs. We analyse non-finite, i.e. special values, of numpy [HMW+20], pandas [tea20], and TensorFlow [MAP+15] objects and directly address the problem of gradient explosion and diminish by monitoring gradient values during the training of TensorFlow and PyTorch [PGM+19] applications.

The rules extracted from those sources can be put into three distinct categories. First, Python anomalies, these rules refer to general anomalies in Python programs, e.g., not closing files properly before they are garbage collected, shown in Table 3.1. Second, machine learning rules which only occur in a subset of Python programs, which utilize specific machine learning related libraries and concepts, such as gradient explosion in programs utilizing PyTorch, listed in Table 3.3. And lastly, security rules which cover security aspects, such as storing weak hash values in cookies, shown in Table 3.2. A rule has a unique ID and an analysis which implements the rule, as shown in Table 3.1. To further differentiate, each rule has a particular category associated with it, namely correctness, performance, code quality or security. In the following we discuss the different analyses, their rules, and rationales behind them.

Analysis	Category	ID	Description
Comparison Behavior	Correctness	A-01	Bad symmetry
		A-02	Bad stability
		A-03	Bad identity
		A-04	Bad reflexivity
Concat Strings	Performance	A-05	Inefficient usage of + operator for strings
Dunder Side Effects	Correctness	A-06	Wrote to global variable in double underline method
		A-07	Wrote to attribute [deactivated]
Files Closed	Code Quality	A-08	File was not closed properly
In Place Sort	Performance	A-09	Sorting in place is possible, but `sorted` is used
Mutable Default Args	Code Quality	A-10	Default arguments have been mutated and are used multiple times
Unusual Type in List	Correctness	A-11	Checks if an unexpected type is added to an existing list
Float Comparison	Correctness	A-12	Floats are compared, which are close to each other but not equal
Unidiomatic Type Check	Code Quality	A-13	Types have been compared with ==
		A-14	Types have been compared with `is` [deactivated]
Compared with Function	Correctness	A-15	A comparison between a function and another object type has been performed
Comparison Operators	Code Quality	A-16	`is` returns something different, compared to ==
In List Mismatch Type	Correctness	A-17	`in` returns False, but flattend list versions return True
Forced Order	Correctness	A-18	String operation depending on order is utilized, order of String is non deterministic
String Strip Misuse	Correctness	A-19	Might have removed something not intended, Arg contains duplicate characters
		A-20	`strip` arg contains duplicate characters
Unexpected Result any/all	Correctness	A-21	Builtins `all` and `any` return unintended results
Mutate While Iterating	Code Quality	A-22	List length changed during iteration

Table 3.1: List of General Python Rules

Analysis	Category	ID	Description
Weak Hash Stored	Security	S-29	Weak hash value stored in cookie

Table 3.2: List of Rules which Address Security

3.2 Analyses & Rationales Behind Them

In the following, the list of proposed analyses regarding general Python issues is discussed. Each analysis may cover one or multiple rules. For each section we begin by explaining the general context and how issues violating the given rules can occur, by showing examples. Afterwards, implications of the given issues are

Listing 3.1 Proper File Closing Example, adapted from [2]

```python
def open_csv(filename):
    return open(filename, 'w', encoding="utf-8")

csv = open_csv("example.csv")
csv.write("1,2,3") # csv is never closed

with open_csv(filename) as csv: # with ensures csv is closed
    csv.write("1,2,3")
```

explained. Lastly, we show how DyLin applies rules in its analyses to catch such issues and how developers are able to fix them. An overview of general Python rules is shown in Table 3.1. Additionally, we discuss security rules, illustrated in Table 3.2.

3.2.1 Files Closed A-08

Reading and writing to files in Python can be achieved by utilizing Python's builtin `open` method. The method returns a file object which can be manipulated further. Python's documentation explicitly mentions that not closing a file may result in only partial writes to disk [23j]. The documentation encourages calling `open` within a context manager, e.g., `with`, which automatically handles the closing of files, illustrated in Listing 3.1.

CodeQL [23b] employs a query which checks if file objects are closed properly. However, due to limitations of static analysis tools, we can improve on checking the rule by applying dynamic analysis concepts to find such issues. For example, if developers implement a method which returns a file object instance, the returned object may follow non trivial execution flows, which is difficult to analyse statically, an example is shown in Listing 3.1. Furthermore, developers might use a reference to the `open` method, to call the function, making it hard to predict statically what method is actually called. Therefore, we propose the analysis Files Closed with the rule A-08. The rule enforces that all file objects used during execution are closed before the program terminates.

To report an issue if the rule is violated, we employ DynaPyt's `pre_call` hook, to get notified as soon as `open` is called. A reference to the file object is saved in the analysis class, to prevent it from getting garbage collected. As a result, we can check all file references, as soon as the `end_exectution` hook is called, to ensure they have been closed properly.

3.2.2 In Place Sort A-09

Python provides multiple ways of sorting lists. The built in method `sorted` returns a new list with its items sorted, as defined by their `__lt__` implementation [23j]. The method supports iterables and can therefore be used for a variety of objects inheriting from `iterable`. Another way of sorting list objects is using the `sort`

[2] https://github.com/hugapi/hug/blob/8b5ac00632543addfdcecc326d0475a685a0cba7/hug/output_format.py#L300

Listing 3.2 Potentially Unintended Result For any / all Function Calls

```
1 >>> all([])
2 True
3 >>> any([[]])
4 False
5 >>> any([[[]]])
6 True
7 >>> all([[[]]])
8 True
```

method provided by the list class. It returns `None` and modifies lists in-place. This means that interpreters, such as CPython [Ros23], do not have to allocate a new list on the heap, but instead modify the existing list. As a result in-place sorting is more space and time efficient, especially for large lists. However, if both the unsorted and sorted lists have to be used later on it makes sense to use `sorted`, because a copy has to be created.

Therefore, the analysis In Place Sort recognizes the use of `sorted` and raises an issue if the original, unsorted, list is not used later. Its rule is part of the performance category, with the corresponding rule A-09. Usage of `sorted` is tracked during the execution, if the unsorted list is garbage collected and has not been referenced, our analysis raises an issue. To avoid overreporting, a threshold is set to ensure a sufficiently large number of items is present in the given list, resulting in a possible speed up.

3.2.3 Unexpected Result any/all A-21

As part of Python's built in functions, the language provides the utility functions `any` and `all`. Both evaluate respectively whether at least one of the items of a given iterable, is `True` or if all of them are [23j]. The documentation intentionally provides code snippets, which function equivalently to `any` and `all` to avoid confusion [23j].

Moreover, Kansal [Kan22] points out that some results of the described functions can be unexpected. A selection of them can be found in Listing 3.2. For example, Line 1 evaluates to `True`, because the list is empty and does not contain any Boolean value. However, the second example in Line 3 evaluates to `False`. Because `bool([])` evaluates to `False`, the outer list in Line 3 does not contain any truthy value. Line 5 returns `True` because the outer list contains another list which is not empty and non empty lists in Python are evaluated as `True`.

If developers work with both methods in complex code bases where empty and nested lists can occur frequently, some of the results might not be expected, as the examples in Listing 3.2 show. Debugging such issues takes time if the developer does not know about such peculiarities. Analysing such issues fits our dynamic analysis approach, because it is possible to check the returned value and the given arguments value to either `any` or `all` calls. Therefore, our analysis tracks the usage of `any` and `all` calls. We flatten iterables inside the argument of the respective function to a one dimensional list. If the flattened iterable is empty, the returned value of `any` or `all` might not be expected. Therefore, DyLin raises an issue, with the rule ID A-21.

Listing 3.3 Compare with Function Example

```python
def comp_strings(a,b):
    if a == b.strip: # function is not invoked
        return True
    return False # always returns False
comp_strings("a", "b")
```

3.2.4 Comparison Behavior A-01 A-02 A-03 A-04

Python allows overloading of operators, in the following we only consider `==`, `<`, `>`, `<=`, and `>=`. Overloading of operators can be achieved by implementing their corresponding special methods in a given class, e.g., `__eq__` for the `=` operator [23j]. Python operators allow comparing different types of objects. If a comparison between two given types is not supported, it is expected that `NotImplemented` is returned [23j]. Furthermore, van Rossum and Ascher [RA00] propose rich-comparisons, which have been added in Python 2.1. Rich-comparisons allow developers to return any type of value, not just objects of type Boolean, when overloading the operators mentioned above. If such values are utilized in a Boolean context, Python will call `bool()` to determine the result. Libraries such as numpy [HMW+20] and TensorFlow [MAP+15] make extensive use of this feature. Both rich-comparisons and the ability to compare different object types limit the ability of interpreters to warn developers if overloaded operators might not behave as they expect them to.

If a comparison is overloaded, the corresponding special method of the left side argument is evaluated first [23j]. Therefore, because different types can be compared, overloaded operator functions have to check the type of the right side argument of the operator, i.e. the given argument of the special method. If a comparison cannot be implemented properly, `NotImplemented` has to be returned, or in many cases simply `False`. In addition, Python's documentation suggests enforcing consistency rules [23j]. Therefore, we propose them as rules for our implementation. First, A-01 references whether symmetry rules apply. Second, A-02 checks if the same result is returned when utilizing an operator multiple times with the same arguments, which ensures stability. Additionally, rule A-03 ensures identity, i.e. comparing an object to `None` returns `False`. And last, reflexivity, where comparing identical objects should return `True`, which is checked by the rule with id A-04. By following the above conventions, developers ensure that basic operators work as commonly expected. Therefore, DyLin tracks the usage of such operators and raises an issue if consistency rules which are commonly expected to hold, for a given operator, do not apply.

There are a few exceptions where the above conventions cannot be achieved, namely comparing with `float` values. This is the case because float values may have the value `NaN` which cannot follow these conventions. Another example is comparing `None` values. Comparing such a value with its identity will always return `True`, because in general `None` is treated as Python's identity value. Therefore we maintain a list of objects where such conventions do not apply, and accordingly, do not report issues for them.

3.2.5 Compared with Function A-15

Python allows comparing different object types to each other. Due to the dynamic nature of the language, developers may not know for certain what type a given object may have during execution. This may lead to comparisons which always return `False`, because the given types cannot be compared in a meaningful way.

An Example can be found in Listing 3.3, where the developer forgot to invoke the function. Accordingly, Line 2 always evaluates to `False` and the wrong value may be returned. Pylint [23g] implements a rule to prevent such issues [3]. However, due to the static nature of Pylint, it is not able to infer the type of `b.strip`, for the given example in Listing 3.3 [4].

Therefore, we leverage dynamic analysis properties, to check the types precisely during execution, allowing to catch cases in which Pylint is not able to infer the type correctly. We propose rule A-15, if a function is compared to an object which is not of type `FunctionType`, we raise an issue. To analyse such issues, we utilize DynaPyt's `comparison` hook. Its arguments contain both the left and right hand side of the comparison, we then check the type for both objects.

3.2.6 Float Comparison A-12

Comparisons between float types can lead to unexpected results, due to precision errors. This is not only the case for Python, but for other programming languages as well [Mon08]. A comparison via == between two `float` types which are sufficiently close to each other may be evaluated as equal because of precision issues. For example, due to limited precision a given float number might not be able to be stored properly, as a result a slight error occurs. This error might then accumulate during the execution of the program if more computations are performed using such values [Bar15]. Therefore Barker [Bar15] proposes an addition to the standard math module which allows developers to check if floats are approximately equal. The proposed addition has been added to Python in version 3.5.

However, due to Python's dynamic typing system, developers might not realise that two float values are compared when utilizing ==. Therefore we propose rule A-12, which enforces that, floats sufficiently close to each other, should not be compared via == if the result of the comparison evaluates to `False`. We leverage DynaPyt's `comparison` hook, to receive both arguments of the comparison and its result. If the comparison evaluates to `False`, we evaluate whether both values are sufficiently close, if so, an issue is reported. The relative tolerance regarding the closeness can be set by developers to ensure it matches their particular needs, depending on the architectures utilized.

3.2.7 Unidiomatic Type Check A-13 A-14

Another comparison which can be considered as unusual is comparing types via the == operator. In general the idiomatic way to compute equality between types is to use with Python's built in `isinstance` operator, because it considers inheritance in contrast to == [Gui01].

[3] `https://Pylint.pycqa.org/en/latest/user_guide/messages/warning/comparison-with-callable.html`
[4] Tested with Pylint Version 2.13.9

Listing 3.4 Example In List Mismatch Type

```
1  class NumberContainer:
2    [...]
3    def get_number(self, condition):
4          # returns list instead of number
5          return [x for x in self.numbers if condition(x)]
6          # fix to ensure only a single element is returned
7          # return next(x for x in self.numbers if condition(x))
8
9  c = NumberContainer()
10 a = [1,2,3]
11
12 c.numbers = a
13 b = c.get_number(lambda a: a == 1)
14
15 if b in a:
16   print("b in a!")
```

Pylints [23g] basic checker implements a similar rule [5], which disallows the usage of == or `is` when comparing types. However, Pylint's type inference is inherently limited due to limitations of static analysis tools. Therefore, we propose rule A-13, which disallows comparing types via the == operator. Furthermore, rule A-14 disallows the usage of `is` when comparing types.

However, when analysing this rule dynamically we found that `is` is commonly used to ensure that types are exactly equal and do not inherit from each other. Therefore, our implementation deactivates rule A-14, per default. This follows the findings of Witowski [Wit20], which encourage the usage of `isinstance` but allow the usage of `is` when having to compare types exactly.

3.2.8 In List Mismatch Type A-17

Listing 3.4 shows an example where a method in Line 3 does not return the type of value which would be expected by its name. The developer chose to return a list where the given condition is fulfilled, instead of returning a single number. The example illustrates how Python's dynamic nature can lead to problems if type annotations are missing or a method name indicates behavior which differs from its actual behavior during execution. Because a list is returned, the condition in Line 15 never evaluates to `True`.

To prevent such issues, we propose rule A-17, which ensures an issue is reported if the following conditions are met when `in` is utilized. (1) both arguments are of type `list`. (2) the left argument contains a single item. (3) the single item contained in the left argument is also contained in the right argument. This may indicate that developers indented to use an element of the respective item type on the left side, instead of an object of type `list`. Consequently, DyLin reports an issue if the above conditions are met.

[5] https://Pylint.pycqa.org/en/latest/user_guide/messages/convention/unidiomatic-typecheck.html

Listing 3.5 Example Mutable Default Arguments

```
1 def a(x=[]):
2     x.append("test")
3     return x
4 
5 a()
6 >>> ["test"]
7 a()
8 >>> ["test", "test"]
```

3.2.9 Mutable Default Arguments A-10

Functions in Python may have arguments with default values. An example of this can be found in Listing 3.5. However, the object used for the default value is computed once and kept on the heap. Line 5 returns `["test"]`. Calling `a()` a second time Line 7 returns `["test", "test"]`. Other programming languages, such as JavaScript, recreate the object for the specified default value every time the function is called. Therefore, this behavior might not be expected by some developers new to Python.

Existing static analysis tools provide rules to check if mutable objects are used as default values [6] [7]. However, DyLin is able to infer if the default argument changed after each invocation during the runtime. This allows developers to continue to use mutable values as default arguments without the recommended mitigation steps by Pylint [23g] or CodeQL [23b], while still warning if the default values has changed after invoking the function more than once. Furthermore, if the function is only used once our analysis will not report an issue, whereas existing static analysis tools will.

Therefore, we propose rule A-10 which disallows changing mutable objects when used as a default argument. We utilize DynaPyt's `post_call` hook to see if the object, used as default value has changed. Default values are obtained from the functions callables `__default__` values.

3.2.10 Concat Strings A-05

Efficient string concatenation is important for many performance sensitive parts of Python programs. strings are immutable in Python, therefore they cannot be changed during runtime without creating a new object. This might be inefficient in case many sub strings need to be concatenated. For example, creating a loop which utilizes += on strings many times, forces some interpreters to create a new object in each iteration [Gui01].

However, interpreters which utilize reference counting can optimize this behavior [Ros23]. For example, CPython [Ros23] checks during runtime if a string can be modified in place [8]. This is the case if (1) the reference count is one, proving that the given string is not referenced anywhere else. (2) a hash value has

[6] https://codeql.GitHub.com/codeql-query-help/python/py-modification-of-default-value/
[7] https://Pylint.pycqa.org/en/latest/user_guide/messages/warning/dangerous-default-value.html
[8] https://GitHub.com/Python/cPython/blob/8a1bab92915dd5c88832706c56af2f5611181d50/Objects/unicodeobject.c#L1855

Listing 3.6 Example Unintended Results for string.strip

```
"abc".strip("bc")
>>> 'a'
"abc".strip("cb")
>>> 'a'
")()()(".strip("()")
>>> ''
```

already been computed. (3) if the string is not interned, meaning that the string is referenced somewhere else, but due to memory optimization only a single copy is stored, therefore the reference count is still 1. And lastly (4) if the instance is not a subtype of a Unicode object. In general if the above mentioned conditions are `True`, linear runtime regarding only the concatenation can be achieved.

Other interpreters such as pypy [23h] do not utilize reference counting and can therefore not implement the same optimization [Gui01]. Furthermore, as stated above, several conditions have to hold to utilize such optimizations. In general, as stated by [Gui01], developers should not rely on such optimizations specific to only a subset of interpreters. Therefore, for performance sensitive parts `''.join()` should be utilized.

Consequently, we propose rule A-05 which checks if a given line utilizes either the `+` or `+=` operator for strings. The number of times the operator is utilized must exceed a threshold, which can be set by developers, depending on their individual performance requirements. If this is the case an issue is reported to the developer, indicating that performance could be increased by collecting a list of strings and then utilizing `''.join(collected_string)` on the list, instead of `+` or `+=`.

3.2.11 String Strip Misuse A-19 A-20

Python's string objects give developers the ability to remove leading and trailing white space via its `strip` function. By providing an argument to the function, characters other than white space can be removed as well. However `strip` will remove pre- and suffixes of any combination of the provided characters in the argument, as illustrated in Listing 3.6. According to Pylints documentation [9] it is a common misconception that the exact string argument of `strip` will be removed. Pylint provides a check which reports an issue in case a string argument of `strip` is used which contains the same character more than once, indicating that the developer made the wrong assumption as described before.

But, because Pylint is a static analysis tool it can only warn about string literals used as the argument. Values computed during execution cannot be evaluated by Pylint. Therefore, we propose rule A-20 if arguments of `strip` contain duplicate characters for arguments which are computed during execution. Additionally, propose A-19, which checks if any characters have indeed been removed in the result, in addition to duplicate characters in the argument. We intentionally split both rules to allow developers a high level of granularity when selecting which rules are important to their particular project.

[9]https://Pylint.pycqa.org/en/latest/user_guide/messages/error/bad-str-strip-call.html

3.2.12 Unusual Type in List A-11

Unlike other programming languages Python's `list`, `set` data types allow adding different types of objects. However, if such types are aggregated from different sources, mixed object types might be stored in a given container. In many cases this is not an issue and intended behavior. But, if the `list` or `set` contains sufficiently many objects of the same type and later on an object of another type is added only once this might be unintended.

Therefore we propose rule A-11. For any existing `list` or `set` with a length exceeding a given threshold, which only contain items of a single data type, DyLin monitors if new elements are added during execution. If the new item type differs from all others, and is not a subtype, an issue is reported.

3.2.13 Dunder Side Effects A-06 A-07

Python's specification [23j] defines a range of special methods which can be invoked by special syntax. For example `__str__` is invoked by the built in `str` function. By overwriting the special method `__str__` developers may specify the string representation of the respective class. Other examples include `__abs__`, `__mul__`, and `__int__`. Operators or built in methods which invoke such special methods, such as `str()`, `*`, and `int()` are commonly used.

Because special methods are part of the class definition and utilize `self` they allow mutating the state of the respective class instance. Because several special methods are invoked by operators which in general do not mutate the state of a given class instance and instead only return the result, such mutations might lead to unintended behavior. Moreover, the global state could be mutated as well by referencing global variables inside special methods and changing them during their execution.

We collected a list of special methods which are not expected to lead to state mutations of a given class. If those methods do not follow our proposed rule A-06, an issue is reported. We monitor calls to such functions, if they are user defined, and execute them twice. Between the first and last execution the state of the given class is monitored. If the state changes, an issue is reported. However, we found that writing to attributes in such methods is commonly done for caching reasons. Therefore, our implementation deactivates A-06, per default. Developers which do not utilize such techniques may still activate the rule. Furthermore, we propose A-07, which disallows only changing global variables in our collected list of methods, to allow developers to choose which of those rules apply best to their specific code base.

3.2.14 Forced Order A-18

The Python language specification [23j] defines the type `set`, which differs from other iterators such as lists. A `set` does not define a fixed order, which means that calling, e.g., `str` on a given set returns a different value during each execution if the set contains more than a single item. By calling functions which force an order on a given `set` object, the results are not consistent each time the program is executed. Due to Python's dynamic nature, developers might not expect such behavior from an arbitrary object.

Listing 3.7 Inconsistent Set Ordering Example

```
1 a = set()
2 a.add("1")
3 a.add("2")
4
5 b = str(a)
6 if b.startswith("1"):
7     print("condition is true")
```

Listing 3.8 Examples of Changing a List While Iterating On It

```
 list = [1,2,3,4]
for item in list:
    list.remove(item)
>>>> [2,4]

for item in list:
    del item
>>>> [1,2,3,4]

for item in list[:]:
    list.remove(item)
>>>> [1,2,3,4]

for item in list:
    list.pop()
>>>> [1,2]
```

Listing 3.7 shows an example where Line 6 evaluates differently for multiple executions. Consider the example in non trivial execution flows. Developers might not know that `b` has been forced into an inconsistent order or that a is of type `set`. By passing such values through arbitrary logic and execution flows, they might occur in unexpected places. Such issues can be especially difficult to track down because of the lack of consistency.

Therefore, we propose rule A-18. DyLin monitors sets during the execution and attaches a label to objects which were forced into an inconsistent order. Objects are labelled by utilizing DyLin's Object Marking analysis Section 3.7. If a labelled object is used in the context of a function which depends on the order of elements, e.g., `startswith`, an issue is reported to developers.

3.2.15 Mutate While Iterating A-22

Python allows adding new values to lists while they are being iterated on. Its reference implementation CPython [Ros23] reports runtime errors for dictionaries which change during iteration, but not for lists. As [Kan22] states, this can lead to surprising results, as shown in Listing 3.8. SonarQube [23a] defines a similar rule [10] but is limited in enforcing it for non trivial control flows overarching different name spaces, as it is a static analysis tool.

Line 3 removes items from `list` by calling `remove` on the current item. Because the first element is removed the list shifts back and the element at index zero is now `2`. But as the list already iterated over the first element `3` is removed next. This results in the list `[2,4]`. Line 7 does not change this list because `del` only removes a copy of the current item and is therefore not considered in our analysis. A similar behavior can be observed in Line 11, because the list is copied via `[:]` and thus should not be reported. Lastly Line 15 results in two iterations where `list` contains `[1,2]` as a result, which will be reported as well.

In summary, rule A-22 enforces that lists do not change their length during iteration. Our analysis will report any change in length of the list being iterated on, we observe the list by utilizing DyLin's Heap Mirror implementation, as explained in Section 3.6. Cases which mutate the list but do not change its length are not considered. To mitigate issues like this developers may either copy the list which is iterated on, as stated in Line 11 or utilize list comprehensions. Both alternatives can be regarded as less error prone, compared to the examples shown before.

3.2.16 Weak Hash Stored — S-29

When using hash functions to store secrets, e.g., in cookies, developers have to ensure the usage of secure hash function. CodeQL [23b] provides a rules which raises an issue if developers store values hashed using the SHA-256 algorithm, as recommended by OWASP [23f]. However, due to limitations of static analysis tools as mentioned in Section 3.1, DyLin implements a similar rule, warning developers to use more secure hash function, e.g., Argon2 as described by Wetzels [Wet16].

Because the analysis utilizes DyLin's YAML configuration language for sources and sinks, as explained in 3.7, other insecure hashing algorithms can be added. Furthermore, sources and sinks can be customized for specific requirements, depending on the use of the given program to analyse. Therefore, developers may customize in which context not to use a specific cryptographic function and in which contexts a security issue would not arise.

We propose rule S-29, which tracks values hashed using the SHA-256 algorithm. An issue is reported if such objects are stored as a cookie using the flask [Gri18] framework. In the future further hash functions as sources and additional sinks can be added by enhancing the list of insufficiently secure hash functions and potentially dangerous sinks, such as storing insufficiently secure values in databases.

[10] https://rules.sonarsource.com/Python/type/Bug/RSPEC-6417

Analysis	Category	ID	Description
Inconsistent Preprocessing	Correctness	M-23	Not all inputs were transformed in the same manner
Leak in Preprocessing	Correctness	M-24	A data leak occurred during pre-processing, when using `CountVectorizer`
		M-25	A data leak occurred during pre-processing, when using `SamplerMixin`
Tensor Non Finites	Correctness	M-26	Function call with Tensor containing NaN in argument and returned Tensor contains NaN
		M-27	NaN value in returned Tensor
Gradient Explosion	Correctness	M-28	Gradient exceeded threshold during training
Numpy Floats Comparison	Correctness	M-30	Numpy Float is NaN and is compared
		M-31	Numpy Float is -inf / inf and compared
Non Finites	Correctness	M-32	Function input contains non-finite value(s)
		M-33	Function output contains non-finite value(s)

Table 3.3: List of Machine Learning Rules

Listing 3.9 Faulty Cosine Similarity

```
1 import numpy as np
2
3 def cosine_similarity(x,y):
4     return (x*y)/(np.sqrt(np.sum(x))*np.sqrt(np.sum(y)))
5
6 cosine_similarity(np.array([2,2]), np.array([0,0]))
7 >>>> array([nan, nan])
```

3.3 Machine Learning Analyses & Rationales

In this section, we discuss the set of machine learning rules, specified in Table 3.3. These rules can aide developers in catching issues during trial runs, before the actual, often expensive training takes place. Consider Listing 3.9 as an example, where developers calculate the cosine similarity and omitted squaring `x` and `y`. This causes a zero division error, which can impact the training. In the following, we explain how our approach addresses such issues directly.

3.3.1 Non Finites **M-32 M-33**

When utilizing Python packages such as numpy [HMW+20] or pandas [tea20] developers may introduce non-finite values during their computations. Consider the example in Listing 3.9 which contains a faulty cosine similarity function where the developer omitted squaring `x` and `y` in Line 4. Because the function is not implemented properly, a division by zero occurs. Per default numpy does not raise a `ZeroDivisionError` for the above example, like Python does. The library only raises an error if developers decide to specify that all numpy errors should be raised, i.e. by calling `np.seterr(raise='all')`.

Listing 3.10 Faulty Cross Entropy

```
1 import TensorFlow as tf
2
3 def cross_entropy(x,y):
4     return -tf.reduce_sum(x*tf.math.log(y))
5
6 y = tf.constant(float(0))
7 x = tf.constant(float(42))
8 cross_entropy(x,y)
9 >>>> <tf.Tensor: shape=(), dtype=float32, numpy=-inf>
```

Non-finite values can propagate through large parts of the code base and can be difficult to track down because no exceptions are raised per default and issues which cause such values can be subtle in nature. Therefore, we propose rules M-32 and M-33. A violation of those rules tells developers exactly where non-finite values such as `inf`, `-inf`, and `NaN` occur.

Function calls returning an object, e.g., `np.ndarray` or `pandas.Dataframe`, which contain a non-finite value are reported by rule M-33. Furthermore, if a function call contains arguments of which one or more contain a non-finite value, rule M-32 is violated. This allows developers to determine precisely where problematic values occur.

Non-finite values are not a bug per se and can be used intentionally. But they can cause issues if they propagate to parts of the code base where they are not expected. By utilizing DyLin developers may verify that non-finite values only occur where they should and are filtered out in parts where they are not expected.

By applying heuristics on the resulting issues, for example, only reporting if the number of issues exceeds a certain threshold or if particular parts of the program are affected, reporting can be improved further in the future. Another approach might be to only report if a certain set of functions contain non-finite values in their arguments. However, a list of such functions has to be constructed, which might vary heavily between different code bases and use cases.

3.3.2 Tensor Non Finites **M-26 M-27**

As described in the previous section, non-finite values can occur for a variety of reasons and can be difficult to track down. This is especially the case for programs which utilize TensorFlow [MAP+15] to build machine learning applications, because many of them rely heavily on float operations, as discussed in Chapter 5.

Consider the example shown in Listing 3.10. The snippet demonstrates a faulty function to calculate cross entropy. Problems occur if, as shown in Line 9, `x` is zero or smaller, resulting in `-inf` or `NaN` respectively, after calling `tf.math.log`. The example is taken from a popular StackOverflow post [11] and slightly altered for

[11] https://stackoverflow.com/questions/33712178/TensorFlow-nan-bug

Listing 3.11 Data leakage example during pre-processing, adapted from [12]

```python
# generate random data
n_samples, n_features, n_classes = 200, 10000, 2
rng = np.random.RandomState(10)
X = rng.standard_normal((n_samples, n_features))
y = rng.choice(n_classes, n_samples)

# leak test data through feature selection
X_selected = SelectKBest(k=25).fit_transform(X, y)

X_train, X_test, y_train, y_test = train_test_split(
    X_selected, y, random_state=42)

gbc = GradientBoostingClassifier(random_state=1)
gbc.fit(X_train, y_train)
y_pred = gbc.predict(X_test)
accuracy_score(y_test, y_pred)
>>> 0.76
```

illustration purposes. Such functions are commonly used in machine learning applications as loss functions. If non-finite values are not taken into consideration during the implementation, as shown in Listing 3.10, the training process can be impacted severely, resulting in a worse model performance than expected.

A possible fix for the example in Listing 3.10 is to add a small constant value to y, while ensuring $y > 0$. As a result, `tf.math.log` will not return non-finite values. As mentioned in Section 3.3.1, non-finite values can occur in a variety of calculations. Because their nature can be subtle, such errors can be difficult to detect.

Therefore, DyLin analyses TensorFlow `Tensor` types during the execution of a given program which utilizes the TensorFlow library [MAP+15], following rules M-26 and M-27. Our analysis differentiates between utilizing tensors, which contain non-finite values, as an argument to an arbitrary method, and function calls which return an object containing a non-finite value. As a result, developers can track precisely where such objects occur, how they are used and where they propagate to. We intentionally split the analyses between tensor values and general non-finite values, to allow optional activation of either analyses.

3.3.3 Leak in Preprocessing M-24 M-25

Nisbet et al. [NMY17] deemed data leakage as one of the top ten data science mistakes. By introducing information into the test data set which is not be available in practice, a sub optimal solution to a given problem is commonly learned [KRPS12]. According to Kaufman et al. [KRPS12] such issues are subtle and hard to detect.

[12] https://scikit-learn.org/stable/common_pitfalls.html#data-leakage-during-pre-processing

Listing 3.11 shows an example utilizing scikit-learn [PVG+11], adapted from scikit-learns documentation. Line 8 computes `X_selected` by using the `SelectKBest` class. However, by performing feature selection before the split occurs, the model will be over-fitted on the selected subset. The issue is best illustrated in Line 17, where an accuracy score of `0.76` is reported for predicting random data. Without data leakage, the accuracy score should be close to `0.5`. Furthermore, such issues may occur for many different data transformations, e.g., scaling the test and training data set before splitting them.

Yang et al. [YBLK22] propose a static analysis tool which is able to monitor data leakage in machine learning programs which utilize scikit-learn [PVG+11]. Their approach leverages datalog facts and rules to infer statically whether leakage between test and training data occurs. Additionally, several heuristics are leveraged to address inaccuracies.

Because dynamic analysis tools are able to infer directly during runtime whether transformations are occurring before splitting data, no heuristics or datalog evaluations are needed. Therefore, we propose two rules which check if leaks in specific pre-processing steps of scikit-learn occur, rule M-24 and M-25. DyLin implements an analysis that warns users of data leakage, based on a list of dangerous transformations that occur before the data is split. The analysis utilizes Object Marking, as described in Listing 3.13, to allow future expansion for more pre-processing steps. For practicality purposes our analysis implementation only checks for function calls where instances of `CountVectorizer`, `HashingVectorizer`, `TfidTransformer`, or, `PCA` is utilized, regarding rule M-24. And, for rule M-25, we cover instances of `SamplerMixin` or `RandomOversampler`. Because our analysis is dynamic, we also cover all instances of sub classes for both rules. However, because the implementation utilizes configuration files for Object Marking, further function calls can be included in the future.

3.3.4 Gradient Explosion M-28

During the training of machine learning applications, backpropagation has to be performed in order to change the current weights, such that the cost function can be minimized. To achieve this, the gradient of the cost function is calculated in regard to the current biases and weights. However, during training a large increase of the calculated norm of the gradient may occur, referred to as gradient explosion, as described by Bengio et al. [BSF94]. The opposite behavior is called gradient diminish, where the gradient decreases rapidly [BSF94]. As Philipp et al. [PSC18] argue, gradient explosions can cause sever training difficulties and limit the depth to which networks can be trained.

To analyse such issues, we propose rule M-28, which disallows gradient values above or below a pre defined threshold. Our implementation monitors the training of machine learning applications utilizing TensorFlow [MAP+15] and PyTorch [PGM+19]. For both packages base classes are monitored, allowing to analyse gradients for a range different optimization methods, e.g., stochastic gradient descent or Adam optimizers. During each training step the analyses for each package respectively investigate the current gradient value. If the gradient vector contains a value exceeding a given threshold, an issue is reported. Additionally the string representation of the gradient is reported as part of the error message. This allows developers to judge whether the gradient might destabilize training.

Listing 3.12 Inconsistent Preprocessing Example

```
1  random_state = 42
2  X, y = make_regression(random_state=random_state, n_features=1, noise=1)
3  X_train, X_test, y_train, y_test = train_test_split(
4      X, y, test_size=0.4, random_state=random_state)
5  
6  scaler = StandardScaler()
7  # X_train data set is scaled but not X_test
8  X_train_transformed = scaler.fit_transform(X_train)
9  model = LinearRegression().fit(X_train_transformed, y_train)
10 model.predict(X_test)
```

Several migrations have been proposed to prevent large weight changes in a single training step due to gradient explosion / diminish. Zhang et al. [ZHSJ20] propose gradient clipping, which supposedly yields better training results. Yang et al. [YPR+19] introduce further methods to reduce the degree of gradient explosion, e.g., tuning activation functions.

3.3.5 Inconsistent Preprocessing **M-23**

A popular machine learning and data science library for Python is scikit-learn [PVG+11], which provides a variety of clustering, regression and classification algorithms. Furthermore scikit-learn provides means for pre- and post-processing of given data sets.

Consider Listing 3.12, which shows an example usage of scikit-learn where Linear Regression is performed. Line 4 splits `X` and `y` into train and test data sets. Afterwards, `X_train` is scaled via `StandardScaler` in Line 8. The issue arises in Line 9 and Line 10. Both `X_test` and `X_train_transformed` are utilized to perform linear regression. However, only the training data set has been transformed by `StandardScaler`, not `X_test`.

This can lead to results which are worse than expected, according to scikit-learn's documentation [13]. Therefore, we propose rule M-23, which enforces that pre-processing steps occurring after the data has been split have to be applied to both training and test data. The analysis monitors objects and marks them if they have been transformed, by utilizing Object Marking, as described in Listing 3.13. If a prediction is performed by any of scikit-learn's estimator classes, marks are compared between train and test sets. In case marks do not match, an issue is reported.

In general, there are two possibilities of fixing such issues. First, both train and test data can be transformed in the same manner. But this can be difficult to track for developers, especially if non trivial control flows are involved. Second, `Pipelines`, provided by scikit-learn can be utilized, which are designed to minimise mistakes during pre-processing.

[13] https://scikit-learn.org/dev/common_pitfalls.html#inconsistent-preprocessing

3.4 Executing Analyses & Collecting Findings

DynaPyt's runtime implementation loads an analysis class at the beginning of the execution and then invokes callbacks to the specified class. Because we propose 23 different analyses types in total but want to allow developers to apply multiple of them at the same time, we propose the usage of a wrapper. Similar to DynaPyt's runtime, it relays the hooks and their arguments to the desired classes. Therefore, our analyses practice separation of concerns, where each class is responsible for verifying its set of rules. This results in a number of advantages.

First, analysis classes are loaded dynamically at the beginning of the execution, allowing developers to only apply a subset of analyses in which they might be interested in. Second, there are no inter dependencies between different analysis types, which means that a given analysis class can be utilized directly with DynaPyt, without the need of our wrapper implementation. Third, existing analysis classes can be integrated into our wrapper class and can take advantage of its features. Features include, Heap Mirror, described in Section 3.6, and our test suite, explained in Section 3.5.

We propose two different wrappers. We call the first one `AnalysisWrapper`, it is responsible for collecting issues across different analysis classes. The collected results are then added to a unified data structure, which can be exported in various formats. Furthermore, it loads the selected set of analyses at the beginning of the execution. Therefore, only a subset may be utilized, which limits performance and memory impact. For example, machine learning analyses responsible to analyse TensorFlow tensors might not be useful if a Django server should be tested. We utilize a second wrapper for automatic testing of different analyses classes, as described in the following section.

3.5 Testing Analyses

To validate the functionality and accuracy of our analyses, we implemented a set of more than 150 unique test cases. Each analysis type has its own test file, which is instrumented and automatically validated by an additional wrapper class called `TestsuiteWrapper`. Because developers declare explicitly where issues should be reported, the class is able to infer the number of false positive findings, false negatives, how many true positives have been found and additionally, a message which indicates what issue could not be discovered if specified. Furthermore, it is independent of our analysis implementations. Developers may use the wrapper class to test their own analyses by inheriting from DyLin's `BasePyssectAnalysis` class.

Our test suite wrapper class functions in a similar manner to `AnalysisWrapper`. The class implements DynaPyt's [EP22] hooks and invokes the actual analysis class automatically. Furthermore, it leverages instrumented test files to gather information about the test itself, allowing developers to specify the analysis to test, test names, and where issues should be reported directly inside the test file. The analysis class to test is specified in the beginning of the test file with the declaration of a dictionary. The dictionary contains the name of the test and the name of the analysis under test. Lines in which a finding should be reported are surrounded by plain strings containing either `"START;"` or `"END;"` to indicate a range of lines in which an issue

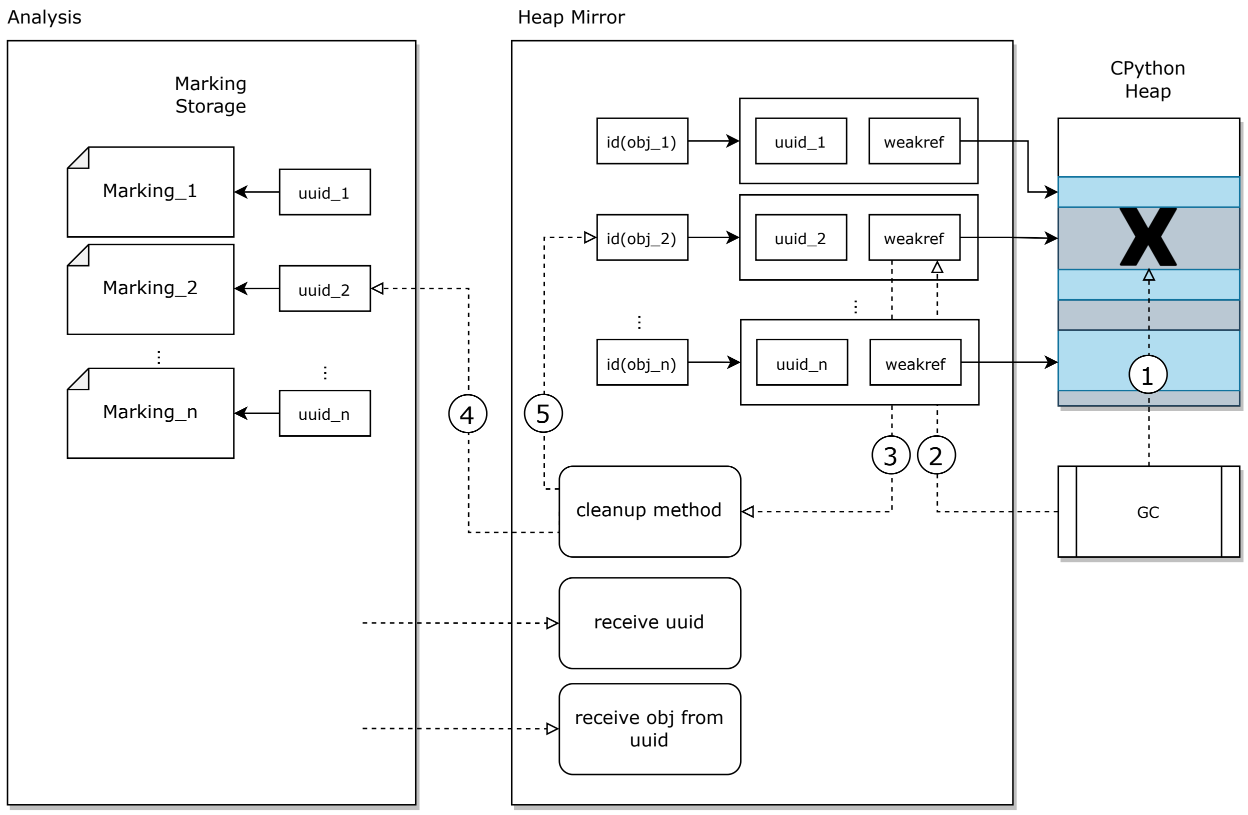

Figure 3.1: Analysis on the left side, shows how an arbitrary analysis is able to obtain a unique id for any given object. These uuids can be used to create a map, where keys are uuids and values are markings related to the given object. Further to the right, the inner working of the Heap Mirror class are shown. The class instance maintains a map where an objects `id` is stored as a key and its uuid and weak reference as value. If the Garbage Collector removes a given object, Heap Mirror is notified and deletes the object from its internal map.

should be reported. Optionally, `"END;"` may contain an error message as a suffix. This allows our test suite to leverage DynaPyt's hooks to collect information what issues have been reported by the particular analysis class. After the execution, developers get a summary of the collected data.

Test cases for DyLin analyses are constructed to be close to actual implementations. For example, the Python test file for our gradient explosion analysis, explained in Section 3.3.4, trains a TensorFlow [MAP+15] model on the mnist data set [Den12], but replaces values of the gradient randomly during the training phase with large values, which should be reported by the analysis.

3.6 Identifying and Analysing Objects on the Heap

Some analyses, such as forced order detailed in Section 3.2.14, need to uniquely identify a given object across the whole execution. Python's specification includes the built in `id` function, which uniquely identifies an object during its lifetime. However, the same id can be reused by another object if the original one is past its lifetime. Regarding CPython's [Ros23] implementation, `id` returns the memory address of the given object. Therefore, other objects may reuse the memory address if the initial one has been garbage collected and thus removed from memory.

In the following, we will focus on Python's reference implementation CPython [Ros23]. Other interpreters might use difference garbage collection techniques not based on reference counting. Furthermore, `id` might not return the objects memory address for some interpreter implementations.

In order to uniquely identify Python objects across the whole execution, we present Heap Mirror, which allows to track objects across executions, by providing a unique id. Furthermore, it notifies analyses right after garbage collection occured whether an object, which is tracked by the Heap Mirror is garbage collected. Even after garbage collection occurs, the unique id stays unique for all future object allocations. Additionally, tracked objects can be retrieved from the interpreters heap at any point during the execution, if it has not been garbage collected. Because DynaPyt [EP22] works by hooking a set of operations which occur during the execution, analysis developers might not be able to obtain a reference to an object relevant to the analysis in the current namespace. Heap Mirror allows developers to retrieve any mirrored object during execution in arbitrary hook methods.

To provide such functionality Heap Mirror leverages Python's `weakref` module, proposed by Drake [Dra00], and added to Python with version 2.1. Weak references are references to objects which do not increase the reference count of the original object. Therefore, garbage collection of the original object is not prevented if a weak reference is stored in memory. This is important if many objects are tracked during execution, keeping them in memory would increase the amount of values on the heap, resulting in worse performance during the execution. Furthermore, assumptions about object lifetimes of the original execution still hold. Another advantage is that a weak reference continues to exist even after its referenced object has been garbage collected, its `ref` value simply returns `None`.

Figure 3.1 explains the inner workings of our Heap Mirror implementation. The right side shows CPython's Heap, containing a variety of objects. If a given analysis on the left wants to receive a unique id, it provides the object as an argument to Heap Mirrors `uniqueid` function. The heap mirror class then mirrors the object by creating a weak reference, utilizing Python's `weakref` module. The weak reference is then stored inside a dictionary, alongside its generated unique id. Keys of the dictionary are the memory address of the given object, obtained by Python's `id` built in function. Additionally, Heap Mirror adds a callback function to each mirrored object, to get notified by CPython's garbage collection if the mirrored object is removed from the heap, shown in the middle in Figure 3.1. The callback is invoked right after the GC removes the object, because finalizers are called after object deletion, as specified by Pitrou [Pit13]. (1) If an object has been garbage collected, i.e. removed from the heap, then (2) the weak reference callback is invoked by CPython. In step (3), Heap Mirrors cleanup method is invoked. Next, (4) analyses which added a cleanup hook are notified by calling hook functions with the objects unique id. (5) Heap Mirror removes the entry of its internal dictionary for the given object. The Heap Mirror class is instantiated as soon as an analysis class imports it, multiple analysis classes share the same Object Mirror instance. As a result, objects are not mirrored multiple times by different analysis classes and the memory footprint is kept small.

However, CPython's `weakref` module cannot provide weak references for all object types [23j]. We could mitigate this issue by wrapping such objects in in a new class which contains the `__weakref__` object in its `__slots__` attribute. We choose to not include such behavior per default because because further execution of the program to analyse might change if some objects are replaced during runtime. For example, calling `type` on a wrapped object will return a different result. Heap Mirror includes a function called `wrap` which implements such functionality if analysis developers choose to add it for their executions. To facilitate execution as close as possible to the original, we utilize two fallback methods for objects which to not support weak references per default. The first fallback identifies the object by its hash value. Such values are not unique across the execution, but Python's language reference defines equal objects as having the same hash value [23j]. Therefore, if multiple objects contain the same hash value, analyses will retrieve the same unique id for both of them. We consider this is an acceptable drawback for our implementation, possible collisions can be mitigated by analyses if large amounts of values which can only be hashed are utilized. Additionally, our Heap Mirror implementation provides the ability to remove a given object from its internal dictionary from outside. Therefore, developers may remove objects, which can only be identified by its hash value, periodically to minimize overlaps. If a hash value cannot be computed, e.g., for `list` types, Heap Mirror utilizes Python's `id` function to uniquely identify objects. As mentioned before, objects id values are only unique across their lifetime. Therefore developers have to utilize Heap Mirrors internal clean functionality to remove such values periodically. Fortunately most Python objects support weak references, outliers include lists and dictionaries, which can be wrapped if developers choose to do so.

An alternative approach might be to create a fork of CPython which adds a weak reference pointer to every object. The current maintainers of CPython deliberately chose not to do so because of the added performance and memory overhead [Het05]. However, for our purposes this might be a worthwhile trade off, because it would eliminate the limitations of our Heap Mirror approach.

Valgrind [NS07] is a dynamic analysis tool for binary analysis, which allows its' plugins to store additional information for values in memory. A related concept is utilized in the dynamic analysis framework for JavaScript, called Jalangi, as described by Sen et al. [SKBG13]. In Shadow Execution Mode, Jalangi wraps objects encountered during execution in a new class, which holds additional information for that value. The class instance is then unwrapped during the instrumentation, to prevent interference with the original execution. This allows both approaches to store additional information for given objects during the analysis.

Our approach instead utilizes weak references to uniquely identify values in memory to analyses. This allows analysis classes to manage shadow values, including their creation, independently. Moreover, our approach allows analysis classes to retrieve any mirrored object at any point in the program, after it has been mirrored. As a result, analyses can monitor objects at any point, without having to rely on the given hooks' arguments or on the current namespace of the application under analysis. Because our approach happens on analysis level, instead of at instrumentation level, no additional modes have to be added to DynaPyt's instrumentation. We deliberately choose a different terminology, to empathizes the differing approaches.

In summary, Heap Mirror allows developers to get additional insight into Python programs when utilizing DynaPyt [EP22]. By mirroring objects during the execution analyses do not need to store them themselves, but can instead retrieve a unique id, which allows them to retrieve the value from the heap directly, while not

Listing 3.13 Object Marking Definition for Weak Hash Analysis

```yaml
name: "Weak hash stored Analysis"
markings:
  weakHash:
    name: "weakHash"
sinks:
  "werkzeug.sansio.response.Response.set_cookie":
    associated_markings:
      - weakHash
      - strongHash
    args:
      - self
      - key
      - value
    validate:
      name: "firstNotLast"
    error_msg: "SSECT-29"
sources:
  "_hashlib.openssl_sha256":
    associated_markings:
      - weakHash
  "argon2._password_hasher.PasswordHasher.hash":
    associated_markings:
      - strongHash
```

increasing reference count and thus decreasing overall memory usage. Furthermore, objects which cannot be retrieved in the namespace of the current hook of the given analyses can still be retrieved, without the need to store the value inside the analysis class. Analyses can check if any object has been encountered before, without having to keep in mind the objects lifetime across execution. The advantages of having a unique id for each object across the whole exectution are manyfold. One particular example is storing additional information for a given object. In the following Object Marking is explained, which utilizes our Heap Mirror implementation heavily to mark unique objects across the execution.

3.7 Object Marking

By facilitating the capabilities of our Heap Mirror implementation, described in Section 3.6, we present a high level analysis which can be adapted by developers for their particular needs. A new Object Marking analysis can be created by setting up a YAML configuration file, where markings, sources, and sinks are defined. Such analyses allow marking arbitrary objects in memory if a given source method is encountered. In case the marked object is used in one of the provided sink methods, an issue is reported.

The example, pictured in Listing 3.13, shows the configuration file of a Object Marking analysis instance, utilized for issues as described in Section 3.2.16. Its purpose is to warn developers if insecure hash values are stored inside cookies. In the following we model a high level approach of how objects on CPython's heap can be annotated with additional information, depending on which functions interact, create or change them.

$$\overbrace{\text{h}}^{M_h = \{m_1, m_2, m_3\}} = \text{hashlib.}\underbrace{\text{sha256}}_{M_{sha256} = \{m_3\}}(\overbrace{\text{pw}}^{M_{pw} = \{m_1, m_2\}})$$

Figure 3.2: Example Marking Propagation

We intentionally model on a high level basis to allow extensions of our approach in the future and to allow the definition of configuration files to control how a particular Object Marking analysis can function. In the following Listing 3.13 will serve as a running example.

We define a set of markings M_x, related to a particular object on the heap, where x is a unique identifier of a given object or function. The set consists of markings $M = \{m_1, ..., m_n\}$, where m_i contains some information of interest. In our implementation m_i is a unique string, but can be replaced with an arbitrary object. A given marking set M_x does not contain duplicate markings. Additionally, we define a set of source functions $\hat{S} : I \times M_f \longrightarrow M_x$, where $\hat{S}_i$ is a source function and M_f is a set of markings which corresponds to a Python function f. $I = \{M_1, ..., M_n\}$ are markings of input objects and M_x denotes the resulting set of markings for the returned value, uniquely identified by x. We define $\hat{S}$ as a set of functions to allow arbitrary changes to markings, e.g., markings associated with a particular function can be removed as well, see Equation (3.2). Furthermore, $\hat{S}$ functions can be Python functions with arbitrary logic, depending on the use case of the given analysis. Our implementation provides a set of simple set operations on markings.

Given our example in Listing 3.13, the function call `h = hashlib.sha256(pw)`, utilizes the source function in Equation (3.1), with the input markings $M_{pw} = \{m_1, m_2\}$, and function markings $M_{sha256} = \{m_3\}$. After `sha256` is invoked, the markings for the object referenced by h are set to $M_h = \{m_1, m_2, m_3\}$, illustrated in Figure 3.2. Listing 3.13 shows that the associated marking for `sha256` is m_3 = `"weakHash"`.

$\hat{S}$ functions cannot only be applied to objects returned by a particular function. Consider the following code snippet `l.sort()`, where l is an arbitrary list. Python's `sort` function does not return a value, it sorts a list in place. Therefore, we have to apply a source function $\hat{S}$ to `l` directly. Another example is a function which utilizes Python's call by reference implementation, it does not return a value but may change an object from its list of parameters. Because Python utilizes call by reference for some objects, the change persists afterwards. For such special cases we can apply $\hat{S}$ functions to input values as well. However, we argue that this can be controlled by developers directly by specifying if the source function should be applied, to `self`, the returned value or to one (or many) of its arguments.

$$\hat{S}_{merge}(I, M_f) = \bigcup_{i \in I} M_i \cup M_f = M_x \tag{3.1}$$

$$\hat{S}_{remove}(I, M_f) = \bigcup_{i \in I} M_i \setminus M_f = M_x \tag{3.2}$$

To model explicit information flow, markings are propagated, per default, throughout the program to analyse, by applying Equation (3.1) to each function call. Subsequently, markings are propagated between objects which relate to each other. If an object is garbage collected, a callback from our Heap Mirror implementation, explained in Section 3.6, is invoked, which allows the Object Marking class to remove the markings of the object which has been garbage collected. This reduces the overhead of the current number of markings kept in memory. Because we utilize Heap Mirror, propagation through complex control flow structures is given inherently, because of CPython's inner object model.

Source functions are inherently associated with our definition of sink functions. We define a set of sink functions $\breve{S} : I \times M_f \longrightarrow \{\emptyset, e\}$, where $\breve{S}_i$ is a sink function, which maps a set of input markings I and M_f to an optional issue message e. Sink functions define when issue messages are being thrown, based on the markings associated with a particular object on CPython's heap. An example can be found in Equation (3.3). The error message e^1 is reported if the the argument for `set_cookies`, does contain m_3, i.e. `"weakHash"` and does not contain m_5, i.e. `"strongHash"`.

$$\breve{S}_{firstNotLast}(I, M_f) = \begin{cases} e^1 & \text{if } m_3 \in I \land m_5 \notin I \\ \emptyset & \text{else} \end{cases} \tag{3.3}$$

Both source and sink functions are utilized as soon as DynaPyt's `post_call` hook is invoked. We identify the function called by their qualified name, as defined in the YAML specification, see Line 21. Afterwards, the input markings of the functions arguments (and optionally of `__self__`) are retrieved and the corresponding source or sink function is applied. If the invoked function is not a source or sink function, markings are propagated by utilizing the source function, defined in Equation (3.1).

In this section, we described how objects on CPython's heap can be marked during execution by utilizing DynaPyt's dynamic analysis approach for Python and by leveraging the properties of our Heap Mirror implementation. Future enhancements may include modelling implicit information flow, i.e. objects which change the control flow, may influence objects which are changed differently depending on the given control flow. This is possible by utilizing DynaPyt's `enter_control_flow` and `exit_control_flow` hooks and by checking the markings of objects which influence the current control flow. Furthermore, a simple domain specific language to formulate source and sink functions can allow developers to add arbitrary logic operations for sink functions, to decide when a given issue should be reported, without having to implement them in Python.

4 Implementation

The following chapter begins by explaining, in Section 4.1, the overall approach how issues are analysed dynamically using DynaPyt, while ensuring extensibility and reusability of our implementation. Next, we describe how rule violations from different analyses are gathered and presented to developers.

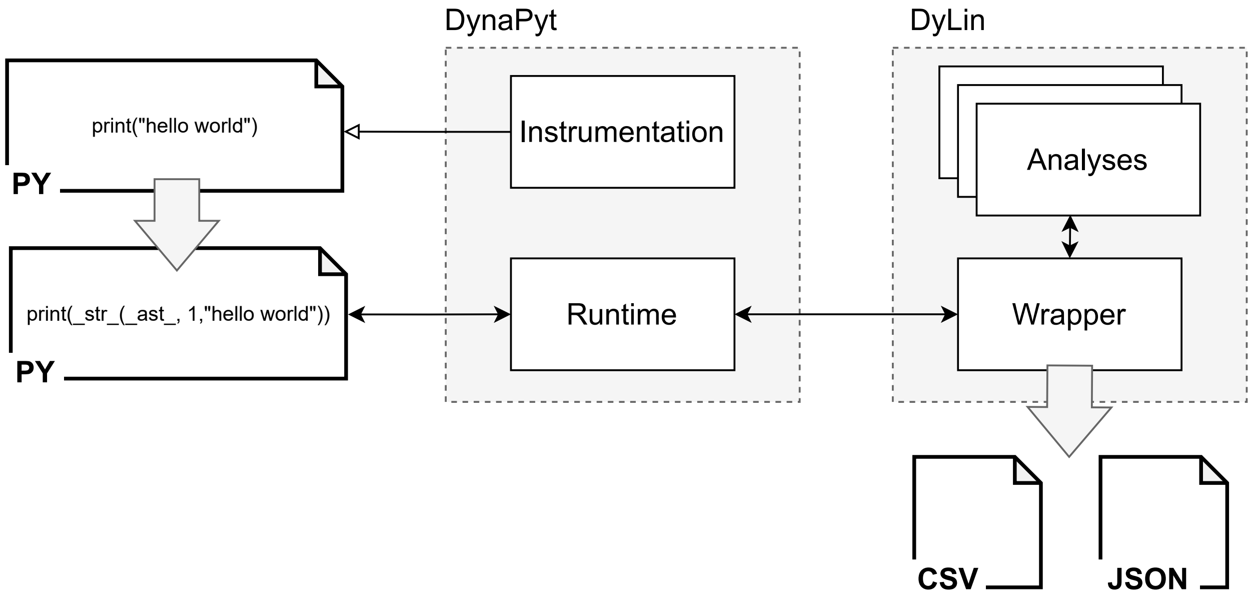

Figure 4.1: Pictured is an overview of the overall approach. On the top left side the original Python file is pictured, which then gets instrumented by DynaPyt [EP22]. DynaPyts runtime, shown in the middle, is utilized to execute instrumented files. Hooks are invoked from the instrumented Python file, to the runtime, which then calls one of DyLin's wrapper classes. Wrapper classes ensure hooks are passed to analyses which are selected to run. After DynaPyt's runtime calls the `end_execution` hook, a wrapper class, pictured on the right, collects issues reported by analysis classes and aggregates them into two report files.

4.1 Overview

To ease the usage for developers DynaPyt [EP22] is distributed as a package and published on PyPi [1]. Developers can install DynaPyt via Python's package manager pip [2] and afterwards use it via the command line. In order to ease the usage of DyLin in a similar manner, the implementation provides a Python package which contains the analyses described. The test suite, described in Section 3.5, is not part of the DyLin package to keep it lightweight.

To setup and execute DyLin developers follow the same approach as when using DynaPyt, i.e. the first step is to instrument the given code-base. Secondly the runtime is invoked execute the given code and the analyses, shown on the left in Figure 4.1. An additional argument to DynaPyt's instrumentation and execution command line option allows to use classes from the DyLin package. The additional argument is an extension to DynaPyt which allows it to load classes which implement the provided hooks from any installed Python package. The option is called `--module`, its argument is the name of the Python package from which to load the class which implements the hooks. This eliminates the need to fork DynaPyt if a new analysis should be added and can be used for future packages other than DyLin.

DynaPyt implements an event hierarchy which allows selecting hooks to events in which an analysis is interested in, other hooks are ignored. The event hierarchy changes how the instrumentation is done. If only a subset of hooks is implemented in a given analysis class, the instrumentation process skips several instrumentation steps if they are not necessary. To run all analyses DynaPyt has to take into account the selected events for each analysis contained in the DyLin package. Therefore, a wrapper class, pictured on the right in Figure 4.1, is needed which implements each hook and relays any calls to the corresponding analysis class. The class is called `AnalysisWrapper`, it allows developers to run multiple analyses provided by DyLin during the same run. Furthermore, after the execution ends, `AnalysisWrapper` collects issues flagged by each given analysis class. Issues are then exported to two different files, as shown on the bottom right in Figure 4.1. Both files can be used by developers to identify, locate, and fix any of the issues reported by DyLin. Additionally, both formats are machine readable to facilitate further extensions, e.g., IDE integration.

4.2 Reporting of Findings

In order to ensure developers are able to locate, identify, and fix issues DyLin includes two output formats. Both contain information regarding the number of issues found and their unique issue code. Data is gathered by the `AnalysisWrapper` class and written once the execution is finished.

First, a JSON file is generated, an example can be found in Listing 4.1. The file consists of a list of objects for each analysis class. The key is the name of the analysis. The object itself contains the number of findings and meta information, for example, statistics of how many comparisons have been checked. Furthermore, a

[1] `https://pypi.org/project/DynaPy/`
[2] `https://packaging.Python.org/en/latest/key_projects/#pip`

Listing 4.1 Example finding from the python-telegram-bot repository

```
1  "FloatComparison": {
2      "nmb_findings": 300,
3      "is_sane": true,
4      "meta": {
5          "total_comp": 263299
6      },
7      "results": {
8          "A-12": [
9              {
10                 "finding": {
11                     "msg": "compared floats nearly equal 1679843759.7261982 and 1679843758.0084958",
12                     "trace": "  File \"<path>/_hooks.py\", line 265, in __call__\n
13                             return self._hookexec([...])\n
14                             [...]"
15                     "location": {
16                         "file": "telegram/ext/_callbackdatacache.py.orig",
17                         "start_line": 453,
18                         "start_column": 59,
19                         "end_line": 453,
20                         "end_column": 94
21                     },
22                     "iid": 389
23                 },
24                 "n": 1
25             }
26         ]
27     }
28 }
29 [...]
```

Boolean value, indicating the sanity of a given analysis, allows analyses to omit findings if they are deemed as not accurate. Detailed results are categorized based on the unique rule number. For each rule number, a list of findings is provided. If a finding occurs multiple times for the same instruction id, i.e. because of the control flow of the program to analyse, it is only reported once and its `n` value is increased, as shown on line Line 24. We include a message, created during execution time, which allows analyses to report findings in greater detail, e.g., by including string values of objects analysed. Furthermore, the filename and the line where the respective issue occurred is included, a stack-trace, shortened for readability purposes, is shown in Line 12. The location is reported precisely to ease building tooling on top of DyLin results and to allow developers to locate the issue as precisely as possible. Lastly, a stack-trace for detailed debugging is contained inside each object.

The second file provides a summary of which findings have occurred for multiple analysis runs. This is useful if large data sets of Python programs are analysed, without having to parse JSON files during each run or as a post processing step. We chose CSV as a data format because it allows appending to the file without having to parse it after each run. The result is a table where the first column specifies the name of the project to analyse, if none is provided the name of the first Python file which is analysed is used. Additional columns refer to specific rule IDs. Each row of the table shows how many times a particular rule violation has been reported for a particular run, omitting duplicate issues which occur on the same line, and with the same Rule-ID.

Both formats are machine readable to ease the development of tooling on top of DyLin. For example, for continuous integration support or IDE integration.

5 Evaluation

We evaluate our approach by applying our analyses to more than 500 submissions from 3 different Kaggle competitions. Furthermore, we analysed 11 public GitHub repositories of varying areas and popularity. Lastly, we evaluate the performance overhead by measuring the time for test execution of the selected GitHub repositories. In the following, we address the research questions below.

- **RQ1**: How prevalent are violations of our rules?
- **RQ2**: How severe are rule violations?
- **RQ3**: How large is the performance overhead imposed by our analyses?

5.1 Experimental Setup

To evaluate RQ1 and RQ2, first, we apply our analyses to more than 500 Kaggle submissions. Kaggle [1] is a community of developers who try to solve data science problems in a competition format. The website provides a variety of different competitions, which can be created by its users or companies interested in letting developers try different approaches on their data sets. Furthermore, Kaggle provides a Python environment used for executing submissions, so that developers do not have to spend time setting up their particular dependencies for each submission. Developers may execute their scripts directly within Kaggle's environment. The environment is used together with Jupyter Notebook [2], an interactive platform where code can be written, annotated, and executed. Kaggle allows to use languages other than Python and some submissions only contain a write up about a topic concerning the competition. However, many submissions use Python and because the same environment is used across many submissions we can leverage these properties to execute a large number of Kaggle submissions concurrently. Because of Kaggle's focus on data science problems and its reproducible environment, we selected three different competitions to evaluate the prevalence of machine learning rule violations.

We provide an approach to automatically analyse a large number ob submissions, utilizing Python, for a given Kaggle competition by reproducing Kaggle's environment. The approach can be broken down into 7 steps. First, a containerized environment, using Docker, is created, based on Kaggle's public docker images [3] and a

[1]https://www.kaggle.com/
[2]https://jupyter.org/
[3]httpw://gcr.io/kaggle-images/python

competition is selected. The containerized environment includes the Python virtual environment with all dependencies necessary to execute the given scripts. However, additional steps have to be taken to analyze submissions and reproduce Kaggle's environment sufficiently. In step 2, our implementation downloads submissions for the given competition, while filtering submissions which have more than two up-votes, to filter out empty scripts. Additionally, metadata for each submission, called kernels by Kaggle, is saved. The metadata allows us to find the submission on Kaggle's website to validate results by hand. Step 3 downloads the competition data-sets and saves them to four different paths, commonly used by the Kaggle community to access the provided data sets. Because Kaggle utilizes Jupyter Notebook, we convert all downloaded `*.pynb` files to regular Python files, using Jupyter Notebooks toolchain. A common occurrence in such files are calls to Python's interactive shell. However, this allows developers to change the current Python environment which can break the execution of future scripts. Therefore, in step 5, we remove such lines in all files. Step 6 instruments all Python scripts in parallel with DynaPyt. Lastly, files are executed in parallel with DynaPyt's runtime and our analyses. We impose a 30 minute time window to each submission and kill the process if the time is exceeded to stop scripts with either endless loops or long execution times. Even if scripts do not finish their execution in the given time window, our analyses are able to infer rule violations and report the results accordingly. Furthermore, if a submission crashes during execution, our analyses are still able to infer rule violations up to the point of the crash. Our implementation which reproduces Kaggle's environment is part of the artefacts accompanying this work, to leverage future research and to ensure reproducibility.

Additionally, we evaluate RQ1 and RQ2 by executing our analyses on 11 GitHub repositories. The list of repositories can be found in Table 5.2. We selected repositories, from GitHub's explore page, by filtering for Python repositories which most recently have been changed, to analyse projects in active development, which may reflect current usage patterns of the Python language. Furthermore, we picked projects which utilise the PyTest [4] package, we leverage its properties to create a reproducible environment. Moreover, Xu et al. [XLZX16] found an almost linear correlation between the number of bugs detected and test coverage. Therefore, we filter additionally, for projects which contain a test suite with sufficiently large coverage of the code base and execute the analysis by utilizing the test suite. To ensure reproducibility, the artefacts accompanying this work contain the commit hash of the project and a detailed list of our findings, including the location, rule ID, and stacktrace.

Lastly, to answer RQ3, we evaluate the performance of our implementation by executing the test suite on all selected repositories. We utilize the tool multitime [5] to execute 10 consecutive runs of each repository. As described in Section 3.4, our approach utilizes a wrapper class which implements methods for corresponding DynaPyt hooks. For evaluation purposes, we implement a second wrapper class, called BaselineWrapper, which contains the same methods and therefore receives the same runtime hook calls from DynaPyt during execution. However, BaselineWrapper does not invoke any analysis. Therefore, a run in which BaselineWrapper is utilized serves as a baseline to compare the overhead of the proposed analyses, without

[4] https://docs.pytest.org
[5] https://tratt.net/laurie/src/multitime/

Analysis Name	Rule ID	Titanic	Spaceship-Titanic	Store Sales	# Submissions
Inconsistent Preprocessing	M-23	13	10	0	**23**
Leak in Preprocessing	M-24	8	9	0	**17**
	M-25	1	1	0	**2**
Tensor Non Finites	M-26	0	0	0	**0**
	M-27	0	0	0	**0**
Gradient Explosion	M-28	0	1	0	**1**
Numpy Floats Comparison	M-30	10	17	0	**27**
	M-31	0	0	0	**0**
Non Finites	M-32	33	69	21	**123**
	M-33	50	76	27	**153**
Total	-	266	213	65	554

Table 5.1: Prevalence of Machine Learning Rule Violations Kaggle Submissions
Shows the number of submissions which violate the specific rule at least once. In total 554 submissions for three competitions are analysed. The competitions are called *Titanic - Machine Learning from Disaster*[6], *Spaceship Titanic* [7], and *Store Sales - Time Series Forecasting*[8].

taking into account the overhead imposed by DynaPyt. To allow reproducibility, detailed results, the BaselineWrapper class and the specific commit hash of each repository are part of the artefacts, for this work.

All experiments are executed on a virtual machine, which utilizes a Intel(R) Xeon(R) Gold 6230 CPU (16 cores @2.10GHz) and 32 GB of RAM. The machine runs Ubuntu 22.04.2 LTS, Python version 3.10.6, and pip version 22.3.1.

5.2 (RQ1) Prevalence of Rule Violations

To evaluate RQ1, we execute our analyses with DynaPyt, as described in Section 5.1, on 554 Kaggle submissions, from three competitions, shown in Table 5.1. Each count denotes a submission which violates the given rule at least once. Our findings show that 49.8% of submissions violate rule M-32 and M-33 at least once, meaning that non-finite values occur at some point during the execution as input or output of a given function call. Manual inspection of these violations show that some submissions filter non-finite values in the beginning of the execution. However, in 41 and 43 submissions, for rule M-32 and M-33 respectively, non-finite values occurred at least twice, indicating that such values are not filtered directly after their occurrence. Additionally, we found 27 submissions, where NaN values where used at least once in comparisons.

[6]`https://www.kaggle.com/competitions/titanic`
[7]`https://www.kaggle.com/c/spaceship-titanic`
[8]`https://www.kaggle.com/c/store-sales-time-series-forecasting`

ID	Name	Stars	LOC	# Rule Violations
#1	miso-belica/sumy	3071	4459	**7**
#2	higlass/clodius	33	11431	**62**
#3	adbar/trafilatura	790	19264	**1**
#4	WebOfTrust/keripy	27	80315	**1**
#5	OpenLEADR/openleadr-python	95	7672	**2**
#6	Trusted-AI/adversarial-robustness-toolbox	3506	91166	**31**
#7	Fatal1ty/mashumaro	490	12711	**7**
#8	Textualize/rich	42352	34373	**3**
#9	tiangolo/typer	10617	11001	**10**
#10	python-telegram-bot/python-telegram-bot	21236	85683	**2**
#11	translate/translate	739	73750	**11**

Table 5.2: Analysed GitHub Repositories

Analysis Name	Rule ID	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	Total
Comparison Behavior	A-01	0	0	0	0	0	0	0	0	0	0	0	**0**
	A-02	0	0	0	0	0	0	0	0	0	0	0	**0**
	A-03	0	0	0	0	0	5	0	0	0	0	0	**5**
	A-04	0	0	0	0	0	0	0	0	0	0	0	**0**
Concat Strings	A-05	0	0	0	0	0	0	1	0	0	0	2	**3**
Dunder Side Effects	A-06	0	0	0	0	0	0	0	0	0	1	2	**3**
Files Closed	A-08	0	4	0	0	0	0	0	0	0	0	5	**9**
In Place Sort	A-09	0	0	0	0	0	0	0	0	0	0	0	**0**
Mutable Default Args	A-10	0	0	0	0	0	0	0	0	0	0	0	**0**
Unusual Type in List	A-11	0	1	0	0	0	0	0	0	0	0	0	**1**
Float Comparison	A-12	0	0	0	0	0	1	0	0	0	1	0	**2**
Unidiomatic Type Check	A-13	0	1	0	0	0	0	6	1	9	0	0	**17**
Compared with Function	A-15	0	0	0	0	0	1	0	0	1	0	0	**2**
Comparison Operators	A-16	0	0	0	0	0	0	0	0	0	0	0	**0**
In List Mismatch Type	A-17	0	0	0	0	0	0	0	0	0	0	0	**0**
Forced Order	A-18	0	0	0	0	0	0	0	0	0	0	0	**0**
String Strip Misuse	A-19	0	0	0	0	0	0	0	0	0	0	0	**0**
	A-20	0	0	0	0	0	0	0	0	0	0	0	**0**
Unexpected Result any/all	A-21	0	0	0	0	0	0	0	0	0	0	0	**0**
Mutate While Iterating	A-22	1	0	1	1	1	0	0	2	0	0	2	**8**
Total	-	**1**	**6**	**1**	**1**	**1**	**7**	**7**	**3**	**10**	**2**	**11**	**50**

Table 5.3: General Rule Violations of GitHub Repositories

Analysis Name	Rule ID	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	Total
Inconsistent Preprocessing	M-23	0	0	0	0	0	0	0	0	0	0	0	**0**
Leak in Preprocessing	M-24	0	0	0	0	0	0	0	0	0	0	0	**0**
	M-25	0	0	0	0	0	0	0	0	0	0	0	**0**
Tensor Non Finites	M-26	0	0	0	0	0	0	0	0	0	0	0	**0**
	M-27	0	0	0	0	0	0	0	0	0	0	0	**0**
Gradient Explosion	M-28	0	0	0	0	0	1	0	0	0	0	0	**1**
Numpy Floats Comparison	M-30	1	0	0	0	0	0	0	0	0	0	0	**1**
	M-31	1	0	0	0	1	8	0	0	0	0	0	**10**
Non Finites	M-32	3	26	0	0	0	4	0	0	0	0	0	**33**
	M-33	1	30	0	0	0	11	0	0	0	0	0	**42**
Total	-	**6**	**56**	**0**	**0**	**1**	**24**	**0**	**0**	**0**	**0**	**0**	**87**

Table 5.4: Machine Learning Rule Violations of GitHub Repositories

We found one submission which produces a gradient explosion, where the gradient contained values exceeding our threshold of 10. Such large changes in the submission occurred in this particular submission a total of 6984 times, indicating severe training difficulties, as described by Philipp et al. [PSC18]. Moreover, we found 17 submissions which violate rule M-24 at least once and 2 submissions which violate rule M-25, indicating leaks occurring during pre-processing steps. Because our analysis only includes a small number of function calls, as described in Section 3.3.3, which can lead to leakage if combined, future implementations may report a larger number of submissions which contain such issues. Lastly, violations of rule M-23, were reported for 23 unique submissions at least once. The findings indicate inconsistency during pre-processing, where, e.g., the training but not the test data have been transformed.

Additionally, we execute our analyses on 11 GitHub repositories of various domains and popularity, listed in Table 5.2. In total, we found 137 rule violations. Findings with the same rule violation on the same line are regarded as duplicates and are only reported once. Table 5.3, shows rule violations of rules A-01 to A-22. Regarding general rules, most issues we found violate rule A-13, where types are compared via `==`. Comparing types via the equals operator instead of utilizing Python's builtin `isinstance` function can lead to unintended behavior. For example, one of the classes may apply a meta class which overwrites the behavior of the equals operator.

We found a total of 9 rule violations for rule A-08, across 2 repositories, where files have not been closed properly. According to Python's documentation [23j], this can lead to partial writes to disk. Another frequent rule violation is rule A-03. All of the findings are based on a known numpy bug [9]. If developers compare the `dtype` of a float type, e.g., `np.dtype(np.float) == x`, our analysis reports a rule violation, because `np.dtype(np.float) == None` returns `True`, contradicting the identity rule. This leads to a variety of problems in the numpy ecosystem and other projects which utilize numpy [10] [11]. In general, the behavior of the equals

[9] https://github.com/numpy/numpy/issues/18434
[10] https://github.com/data-apis/array-api/issues/582
[11] https://github.com/google/jax/issues/3001

Analysis Name	Rule ID	High	Medium	Noteworthy	False Positive	Total
Inconsistent Preprocessing	M-23	0	2	0	0	2
Leak in Preprocessing	M-24	2	0	0	0	2
	M-25	1	0	0	0	1
Gradient Explosion	M-28	1	0	0	0	1
Numpy Floats Comparison	M-30	0	0	0	3	3
Non Finites	M-32	0	0	3	8	11
	M-33	0	1	4	9	14
Total	-	**4**	**3**	**7**	**20**	34

Table 5.5: Severity of Rule Violations Kaggle Submissions

operator regarding numpy `dtypes` is deemed problematic [12] and there are plans to deprecate some of its behavior. Our analysis warns developers of this behavior and encourages the usage of numpys `issubdtype` method to compare dtypes, instead of utilizing inconsistent behavior from the package. Furthermore, our results show 8 findings, where list lengths are changed during iteration, violating rule A-22, in 6 different repositories.

Following the results from Kaggle submissions, we evaluate the prevalence of rule violations for machine learning rules based on the GitHub repositories listed in Table 5.2. Repository #1, #2, #5, and #6, utilize data science libraries and are therefore susceptible to machine learning rule violations. In total, we found 87 machine learning rule violations, in 4 different GitHub projects, as illustrated in Table 5.4. Most findings stem from non-finite values, violating rules M-32 and M-33. Furthermore, following rule M-30 and M-31, comparisons with non-finite values were reported 11 times. We found one gradient explosion, which occurred 597 times during the execution, this may lead to sever difficulties during training, as shown by Philipp et al. [PSC18].

Lastly, no security rule violations have been observed in the 11 GitHub repositories analysed.

5.3 (RQ2) Severity of Rule Violations

We evaluate RQ2, by manually inspecting 34 rule violations of Kaggle submissions and all 137 rule violations found in GitHub repositories, listed in Table 5.2. First, we evaluate whether the finding is false positive. We define false positive for our evaluation purposes as a finding which, either (1) does not violate the rule in practice or (2) does not have any impact on the given repository or Kaggle submission. Findings, which are not evaluated as false positive are then added to one of three categories. *High*, indicates a large risk on the correctness of the computed result. Furthermore, *Medium* risk denotes findings which might influence correctness in some cases or have an impact on the performance. Lastly, *Noteworthy* expresses findings which do not pose an immediate risk on correctness or performance, but indicate something could be suitable for changes.

[12] https://github.com/numpy/numpy/issues/7242

Repository ID	High	Medium	Noteworthy	False Positive	Total
#1	0	6	0	1	7
#2	0	1	5	56	62
#3	0	0	0	1	1
#4	0	0	1	0	1
#5	0	1	0	1	2
#6	0	17	6	8	31
#7	0	1	6	0	7
#8	1	0	0	2	3
#9	0	0	9	1	10
#10	0	1	0	1	2
#11	0	2	5	4	11
Total	**1**	**29**	**32**	**75**	137
Excluding M-30-33	**1**	**8**	**31**	**11**	51

Table 5.6: Severity of Rule Violations Github Repositories

The outcome of our manual inspection regarding 34 rule violations in Kaggle submissions is shown in Table 5.5. We regard four inspected rule violations as high risk. Three of which are leaks which occur during prepossessing, they indicate a large risk on the correctness of the computed result. Another high risk finding is a gradient explosion, indicating sever training difficulties. Next, three findings are observed as violations with medium risk. Two of them contain inconsistencies during the pre-processing step, which can have impacts on predictions or training. Moreover, one submission utilizes non-finite values explicitly in the prediction phase, which may have an impact. During our review, we found that most submissions remove non-finite values before prediction or training by either imputing such values or filtering them explicitly. This explains the number of false positives regarding rules M-30, M-32, and M-33. Analyses report occurrences of non-finite values before predictions or training takes place, if they are filtered out properly before, we regard such findings as false positives. Lastly, we found 7 noteworthy rule violations, all of them are reports of non-finite values, they occur during data visualization or could have been removed earlier on.

Next, we examine the severity of rule violations in GitHub repositories, they are illustrated in Table 5.6. Following the observations of rule violations in Kaggle submissions, we observe a large number of false positives for rules M-30, M-31, M-32, and M-33. Most cases, of findings we regard as false positives, occur in repository #2, which deliberately utilizes non-finite values as placeholders. Moreover, repository #6, compares non-finite values in 8 observed cases, to handle possible miscalculations and edge cases. If we only take into account the number of false positives not reported by rules M-30, M-31, M-32, and M-33, we find 11 false positive findings, 7 of which occur due to an instrumentation bug in DynaPyt.

We regard 1 rule violation as high risk. Repository #8, violates rule A-13, where classes are compared via `==`. The repository allows developers to generate string representations of classes automatically, for readability purposes. To automatically generate such string representations, they utilize the method `auto_rich_repr`, which uses the inspect module to infer the signature of `__init__`, illustrated in Listing 5.1. We found that the left hand side of the equality comparison in Line 12 is a user controlled class. Therefore, users are able to overwrite

Listing 5.1 Issue with high risk, found in the rich[13]repository

```python
def auto_rich_repr(self: Type[T]) -> Result:
    """Auto generate __rich_rep__ from signature of __init__"""
    try:
        signature = inspect.signature(self.__init__)
        for name, param in signature.parameters.items():
            if param.kind == param.POSITIONAL_ONLY:
                yield getattr(self, name)
            elif param.kind in (
                param.POSITIONAL_OR_KEYWORD,
                param.KEYWORD_ONLY,
            ):
                if param.default == param.empty: # A-13 compares classes via equals
                    yield getattr(self, param.name)
                else:
                    yield param.name, getattr(self, param.name), param.default
    except Exception as error:
        raise ReprError(
            f"Failed to auto generate __rich_repr__; {error}"
        ) from None
```

the equality operator for their specific class, breaking the equality check. As a result, parameters which use such classes as default values will miss both the name and default value from their string representation. By instead utilizing `isinstance` to compare both types, such issues can be prevented.

Our evaluation shows one occurrence of a gradient explosion, observed 597 times during the training, in repository #6. An excerpt is shown in Listing 5.2, it illustrates how the model is initialized and trained with PyTorch [PGM+19]. The training loop begins in Line 1, DyLin reports a violation of rule M-28 in Line 12, min and max values are reported as part of the message. This issue can occur for a number of reasons [YPR+19; ZHSJ20]. However, such gradients might limit the depth of which the network can be trained, as explained by Philipp et al. [PSC18]. Therefore, gradient clipping might increase the effectiveness of the model [ZHSJ20]. We categorize the finding as medium risk, because our threshold of 10 is only slightly exceeded. Further findings in #6, declared as medium, are occurrence's of non finite values which occur during calculations or pre-processing, but don't propagate widely throughout the program. These findings are regarded as medium risk, because results might be impacted to some degree.

Two violations of rule A-12, where nearly equals floats are compared, are observed in #6 and #10. Repository #6, compares float loss values during training, where inaccuracies can occur due to floating point representation. In the second instance, the comparison controls the threshold where data last accessed is cut off. An excerpt, where the issue is found is shown in Listing 5.3. DyLin, reports a rule violation for Line 13, causing either additional or less data to be cut off in Line 15. The issue could be mitigated by instead transforming the timestamp to an integer, in Line 7, which does not pose precision issues.

[13] https://github.com/Textualize/rich/blob/076e0d208eb0b4e74cd8639e11a558b9319bd799/rich/repr.py#L68

[14] https://github.com/Trusted-AI/adversarial-robustness-toolbox/blob/c1859ba74ed815c758a63946aa93a2db0cedb417/art/attacks/inference/membership_inference/black_box.py#L273

Listing 5.2 Issue regarded as medium risk, found in the Adversarial-Robustness-Toolbox[14]repository

```python
for _ in range(self.epochs):
    for (input1, input2, targets) in train_loader:
        input1, input2, targets = to_cuda(input1), to_cuda(input2), to_cuda(targets)
        _, input2 = torch.autograd.Variable(input1), torch.autograd.Variable(input2)
        targets = torch.autograd.Variable(targets)

        optimizer.zero_grad()
        outputs = self.attack_model(input1, input2)
        loss = loss_fn(outputs, targets.unsqueeze(1))

        loss.backward()
        optimizer.step() # M-28 Gradient Explosion min -0.003739866428077221 max 16.22378921508789
```

Listing 5.3 Issue regarded as medium risk, found in the python-telegram-bot[15]repository

```python
def __clear(self, mapping: MutableMapping, time_cutoff: Union[float, datetime] = None) -> None:
    if not time_cutoff:
        mapping.clear()
        return

    if isinstance(time_cutoff, datetime):
        effective_cutoff = to_float_timestamp(
            time_cutoff, tzinfo=self.bot.defaults.tzinfo if self.bot.defaults else None)
    else:
        effective_cutoff = time_cutoff

    to_drop = [key for key, data in mapping.items()
                    if data.access_time < effective_cutoff] # A-12 nearly equal floats compared
    for key in to_drop:
        mapping.pop(key)
```

Another example of findings, which we regard as medium risk, occur in #1. The authors of the repository utilize the power method to calculate the largest eigenvalue of a matrix. Our analysis reports a total of 6 findings on different lines, which indicate an overflow during the calculation because of a lack of normalization. Both rules M-30, M-31 and M-32, M-33 report first infinite values and later on NaN values during the calculation. As a result, some values are not calculated correctly and may have an impact on the correctness of the execution.

We regard most findings which violate rule A-13, as noteworthy, because the majority of them does not have an immediate impact on correctness or performance. Such findings can be regarded as code quality issues. A similar observation is made about violating rule A-08, no immediate impacts on either correctness or performance were observed during our manual observations, therefore we regard such violations as noteworthy.

[15] https://github.com/python-telegram-bot/python-telegram-bot/blob/401b2decce9d7e9ae3a48f760e18913f5173861b/telegram/ext/_callbackdatacache.py#L444

Listing 5.4 Issue regarded as medium risk, found in the translate[16]repository

```python
1 @staticmethod
2 def openoutputfile(options, fulloutputpath):
3     """Opens the output file."""
4     if fulloutputpath is None:
5         return StdoutWrapper()
6     return open(fulloutputpath, "wb") # A-08, file not closed
```

An example can be found in Listing 5.4, where DyLin reports a rule violation in Line 6. During our evaluation we observe that the file is written to. However, we did not observe partial writes in our tests, such issues depend on how the operating system manages files on disk and how long the application operates [23j].

During our manual investigations of analysing GitHub repositories, we reported a total of 2 issues to developers. As of the time of writing we did not receive a response.

5.4 (RQ3) Runtime Overhead

To answer RQ3, we invoke a new wrapper class, called BaselineWrapper. The class utilizes the same DynaPyt hooks as AnalysisWrapper, which invokes all analyses. The new wrapper class function body's simply return right away and do not have any functionality. First, the test suite of the given repository is executed ten times, where DynaPyt invokes BaselineWrapper. The mean of the execution times from these runs serves as a baseline for our evaluation. The same process is repeated with AnalysisWrapper, which invokes each analysis if needed, again we record the mean execution time. Following the suggestions of, Fleming and Wallace [FW86], Figure 5.1, shows the overhead based on both mean values for each repository, as listed in Table 5.2.

Compared to the empty analysis, the overhead of the repositories analysed, is at most 7 times higher, which we consider acceptable for practical use. We note that the overhead differs between repositories. The reason for this is that our analyses inspect items from lists during the execution. If a given repository under analysis utilizes lists with a large amount of items, the overhead may increase.

Even though we believe that the overhead is acceptable, it can be improved in future updates. Due to our modular approach it is possible to only utilize a subset of analyses in which a developer is interested in, further decreasing the overhead.

[16]https://github.com/translate/translate/blob/3cfa20fa6ec9f0f5d6cb0f15f81ea411efd53b5b/translate/misc/optrecurse.py#L646

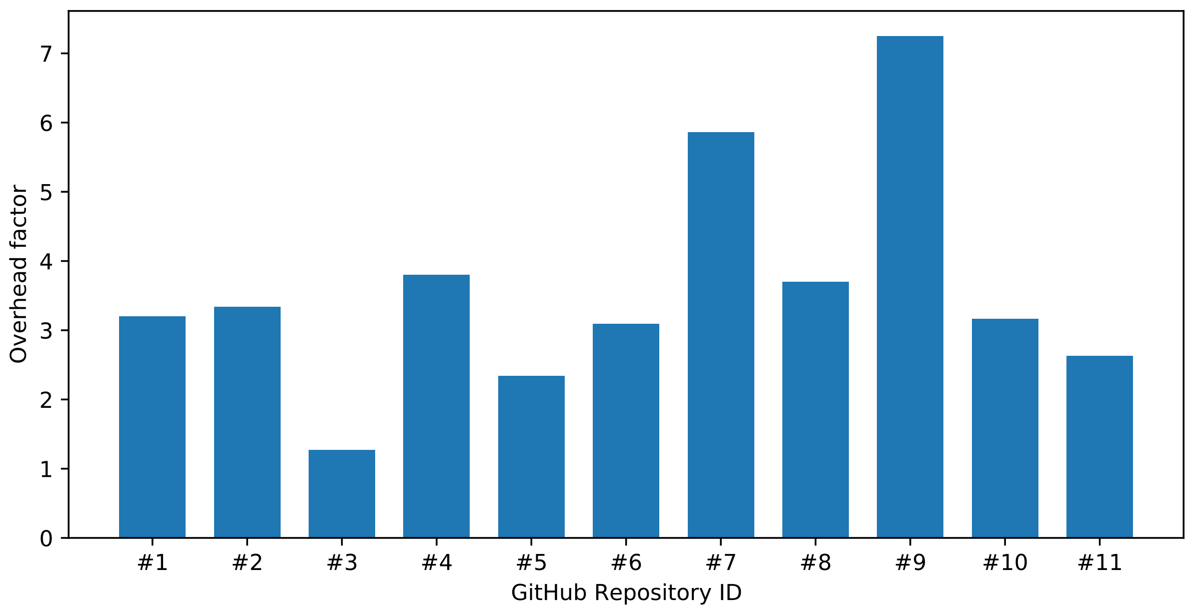

Figure 5.1: Analyses Overhead

5.5 Threats to Validity

The validity of our evaluations are subjects to several threats. We evaluate the prevalence of rule violations on 554 Kaggle submissions and 11 GitHub repositories, which may or may not be representative of general Python projects. To limit this threat, we evaluate a large number of Kaggle submissions from different competitions. Furthermore, we select GitHub repositories which have been changed recently to approximate state of the art usage of the Python language.

Next, our analyses executes the test suite of GitHub projects which may or may not be representative of a general execution of the given project. Therefore, some issues might be missed by our analyses. This is a general issue with dynamic analysis tools which rely on test suites, as stated by Ernst [Ern]. To mitigate this thread, we aimed to select repositories with many test cases and high coverage.

Last, our categorization of severity regarding RQ2 is subjective. To mitigate this threat, we defined how each category is evaluated in Section 5.3 and categorized carefully based on the predefined definitions.

5.6 Discussion

In the following we discuss our results regarding the research questions defined earlier. First, we consider the amount of false positives regarding rules M-30,M-31,M-32, and M-33 as too large. However, some of the rule violations are regarded as medium risk. Our manual inspection of the findings revealed that in many repositories and Kaggle submissions, developers take precautions to prevent non-finite values from interfering with the desired results. Therefore, future work may utilize our findings and add additional heuristics, which control thoroughly, when such findings should be reported to developers. However, we consider raising an issue for occurrences of non finite values as a useful tool for developers, to ensure non-finite values do not propagate by utilizing DyLin to see exactly where such values occur. It can be worthwhile to utilize such analyses, e.g., in a trial run before actually training a model properly. Due to our modular approach, both analyses regarding rules M-30,M-31,M-32, and M-33 can be deactivated.

Many machine learning rules are designed to find issues if certain packages, such as TensorFlow, scikit-learn, or PyTorch are utilized by developers. Therefore, they are inherently limited in how often they can be found in a range of Kaggle submissions or GitHub repositories. However, we think such analyses are beneficial for developers in general and that a greater range of rules regarding specific machine learning libraries may be rewarding to explore in future work.

The majority of findings are regarded as noteworthy, because many of them are related to code quality issues or do not impose an immediate risk to either correctness or performance. However, they reveal patterns which may lead to problems in the future and therefore we still consider them as worthwhile to report. For example, we found several occurrences of changing a lists size during its' iteration, which does not have to affect correctness, but makes future changes susceptible to bugs which are hard to track down. Moreover, it increases the complexity, compared to equivalent approaches. Another example, is comparing types via ==, this can lead to a variety of side effects, which many developers might not have in mind when developing the application. Identifying parts of the applications where code quality is lacking, facilitates future development.

The evaluation of machine learning rules, shows that the usage of popular machine learning libraries can be difficult. Insights from dynamic analysis into the training or pre-processing steps of a given application can bring valuable information to data scientist and developers alike. This includes the propagation of non-finite values, training difficulties through gradient explosion and common pitfalls in libraries, such as scikit-learn [PVG+11], and TensorFlow [MAP+15]. Catching such issues early on before training or predictions are executed can save both time and computing cost.

DyLin can be included in existing continuous integration (CI) environments by executing the test suite of the given program. If general rule violations occur, as shown in the previous sections, the generated JSON file can be parsed by the particular CI suite and be presented to developers. By showing both the messages and the location where the issue is located, developers are able to identify and resolve the rule violation. Depending on the application at hand, analyses can be activated or deactivated as desired. Moreover, with the approach presented in Section 3.7, developers may create their own analyses by simply changing YAML configuration files.

6 Related Work

In the following, we begin by discussing general analysis approaches for Python programs across both academic works and methods utilized by open source developers and companies. Afterwards, we explain related works on dynamic analysis across different programming languages.

6.1 Analysis of Python Programs

The landscape of commonly used Python static analysis tools is described by Gulabovska and Porkoláb [GP19]. Pylint [23g], is named as the most popular Python static analysis tool. It provides a variety of checks, including enforcing PEP-8 [Gui01] rules. To extend the functionality developers built a variety of plugins to check for additional rule violations. An additional tool is flake8 [23c], which wraps Pyflakes [23d], and additional checkers in a single tool. The tool provides an alternative to Pylint and includes checks for code style, complexity and more. Mypy [23e], is a static type checker for Python. Mypy utilizes Pythons annotation system, van Rossum Jukka Lehtosalo Łukasz Langa [Ros14], to check if given types match, while still supporting duck typing if no annotations are present. The tools mentioned are all statically analysing Python code and are therefore limited in their evaluations. Our approach is able to infer types, values and the exact control flow during the execution.

Pyre [23i] is another type checker for Python by Meta with a focus on performance. The tool ships with Pysa, as described by Bleaney and Cepel [BC22], which focuses on analysing data-flows. This allows the tool to provide a variety of security checks. Regarding Pysa's analyses, the developers explicitly mention the limitations of statically analysing a dynamic language, such as Python [BC22]. For example, function calls to modules imported during runtime cannot be analysed by Pysa. Our approach addresses these issues directly by following a dynamic analysis approach.

Xu et al. [XLZX16], describe a predictive analysis engine for Python. The tool begins by executing the program under analysis and collecting its trace. Based on the trace, symbolic variables are introduced, allowing variations of the previous run. As a result, the tool is able to explore neighbouring states, allowing to find previously undetected bugs. Another approach by, Chen et al. [CCX14], proposes analyzing the information flow between values. To achieve this, a hybrid of static and dynamic analysis is used. Both implicit, i.e. control flow, and explicit, i.e. data flow analysis is utilized. The approach mostly focuses on security implications of problematic flows between values. DDUO [ASDN21] is a dynamic analysis approach, with a focus on differential privacy. The approach ensures privacy assumptions hold for a program under analysis. In contrast to the aforementioned works, our approach takes into account general rules regarding code quality, performance, and correctness of Python programs.

Lagouvardos et al. [LDG+20] statically analyse the shape in TensorFlow programs by first transforming Python source code into an intermediate representation and then defining analyses via Datalog rules. The approach presented in our work addresses a variety of issues which can arise in different machine learning libraries, including TensorFlow.

6.2 Works on Dynamic Analysis

Valgrind, as described by Nethercote and Seward [NS07], is an instrumentation tool for binary analysis. It executes a program under analysis by instrumenting parts via just-in-time compilation. Furthermore, Valgrind shadows the memory of the original program to allow its plugins to deduct as much information as possible. Memcheck [SN05], is a tool which detects memory errors during execution by utilizing shadow values. Common use case cases for Valgrind include profiling and memory leak detection.

A more closely related tool is Jalangi, described by Sen et al. [SKBG13], which is a dynamic analysis framework for JavaScript. Jalangi directly addresses the dynamically typed nature of the language and provides a framework to implement specific analyses on top of its instrumentation. Moreover, it allows users to record and replay selected parts of the code base.

DLint, as described by Gong et al. [GPSS15], is a dynamic analysis approach to lint JavaScript code. The approach provides a generic framework to add rules which cover bad coding practices specific to JavaScript. DLint specifies rule violations declaratively, allowing third parties to extend their rule set. its runtime event predicates can be compared to DynaPyt's hooks. Another dynamic analysis for JavaScript is TypeDevil, depicted by Pradel et al. [PSS15]. TypeDevil observes types during the execution and warns developers of type inconsistencies. Observed types are gathered in a Type Graph, which is then taken into consideration to report functions, variables or properties which have multiple inconsistent types.

Both DLint [GPSS15] and TypeDevil [PSS15] directly address the dynamic nature of JavaScript, by analysing properties of the code under analysis during its' execution. Our work directly addresses similar properties of the Python language, regarding both general and machine learning specific issues.

7 Conclusion

This work presents DyLin, a dynamic analysis tool for Python, which includes a wide variety of rules for both general purpose and machine learning applications. Compared to the existing tool landscape it deliberately leverages the properties of dynamic analyses to infer additional information about the project under analysis. Thus, our approach is able to evaluate rules which are not possible to analyse statically. The modularity of our approach allows selecting a subset of analyses in which a developer might be interested in. Moreover, due to the modular approach, together with a high level analysis, which can be configured with YAML configuration files, DyLin can be enhanced in the future with little effort. Additionally, we present a novel approach to store shadow values by utilizing weak references. As a result, analyses with an even wider range of rules can be implemented in the future.

We apply our approach on submissions to Kaggle competitions, leading to the discovery of at least 4 issues with high risk regarding correctness and 3 which we regard as medium risk. Moreover, we apply our analyses to 11 GitHub repositories of different domains and sizes. We find 1 rule violation, which we regard as high risk and 8 with medium risk, regarding both correctness and performance.

Our approach illustrates the practicality of applying dynamic analysis tools to highly dynamic languages, such as Python. It demonstrates how dynamically analysing general Python code can aid developers with regard to correctness, performance, and code quality. For example, DyLin can be utilized in continuous integration environments, to notify developers whether rule violations occur. Due to the modular approach, developers may choose which rules should be taken into account for their particular needs. Furthermore, this work lays the foundation for future tools, by providing vital insights and additional tooling to support dynamic analyses for the Python language. Moreover, our approach helps developers with the usage of machine learning libraries, such as scikit-learn or TensorFlow in ensuring their application works as expected, for example, during initial training runs, to avoid unstable training.

Bibliography

[22] *Stack Overflow Developer Survey 2022*. https://survey.stackoverflow.co/2022/. 2022 (cit. on p. 13).

[23a] https://docs.sonarqube.org. 2023 (cit. on p. 33).

[23b] *CodeQL*. https://github.com/github/codeql. 2023 (cit. on pp. 21, 24, 29, 33).

[23c] *flake8*. https://github.com/pycqa/flake8. 2023 (cit. on pp. 13, 17, 63).

[23d] *flake8*. https://github.com/PyCQA/pyflakes. 2023 (cit. on pp. 13, 63).

[23e] *mypy*. https://github.com/python/mypy. 2023 (cit. on pp. 13, 17, 63).

[23f] *Password Storage Cheat Sheet*. Cheatsheet. 2023. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (cit. on p. 33).

[23g] *pylint*. https://github.com/PyCQA/pylint. 2023 (cit. on pp. 13, 17, 21, 27–29, 63).

[23h] *pypy*. https://foss.heptapod.net/pypy/pypy. 2023 (cit. on p. 30).

[23i] *pyre-check*. https://github.com/facebook/pyre-check. 2023 (cit. on pp. 13, 17, 63).

[23j] *The Python Standard Library*. Documenatition. 2023. URL: https://docs.python.org/3/library/index.html (cit. on pp. 14, 17, 24–26, 31, 42, 55, 60).

[APH+08] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (Sept. 2008). Conference Name: IEEE Software, pp. 22–29. ISSN: 1937-4194. DOI: 10.1109/MS.2008.130 (cit. on p. 17).

[ASDN21] C. Abuah, A. Silence, D. Darais, J. P. Near. "DDUO: General-Purpose Dynamic Analysis for Differential Privacy". In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. ISSN: 2374-8303. June 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00043 (cit. on pp. 13, 63).

[Bar15] C. Barker. *PEP 485 – A Function for testing approximate equality*. PEP 485. 2015. URL: https://peps.python.org/pep-0485/ (cit. on p. 27).

[BC22] G. Bleaney, S. Cepel. *Pysa: An open source static analysis tool to detect and prevent security issues in Python code*. Tech. rep. 2022. URL: https://engineering.fb.com/2020/08/07/security/pysa/ (cit. on pp. 13, 14, 63).

[BSF94] Y. Bengio, P. Simard, P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994). Conference Name: IEEE Transactions on Neural Networks, pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181 (cit. on p. 37).

[CCX14] Z. Chen, L. Chen, B. Xu. “Hybrid Information Flow Analysis for Python Bytecode”. In: *2014 11th Web Information System and Application Conference*. Sept. 2014, pp. 95–100. DOI: 10.1109/WISA.2014.26 (cit. on pp. 13, 63).

[Den12] L. Deng. “The mnist database of handwritten digit images for machine learning research”. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142 (cit. on p. 40).

[DGR17] A. Di Franco, H. Guo, C. Rubio-González. “A comprehensive study of real-world numerical bug characteristics”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, pp. 509–519. DOI: 10.1109/ASE.2017.8115662 (cit. on p. 22).

[Dra00] F. L. Drake. *PEP 205 – Weak References*. PEP 205. 2000. URL: https://peps.python.org/pep-0205/ (cit. on p. 41).

[EP22] A. Eghbali, M. Pradel. “DynaPyt: A Dynamic Analysis Framework for Python”. en. In: (2022), p. 11 (cit. on pp. 14, 17, 18, 39, 41, 42, 47, 48).

[Ern] M. D. Ernst. “Static and dynamic analysis: synergy and duality”. en. In: (), p. 4 (cit. on pp. 17, 18, 61).

[FW86] P. J. Fleming, J. J. Wallace. “How not to lie with statistics: the correct way to summarize Benchmark results”. English. In: *Communications of the ACM* (1986). ISSN: 0001-0782 (cit. on p. 60).

[GP19] H. Gulabovska, Z. Porkoláb. “Survey on Static Analysis Tools of Python Programs”. In: *Proceedings of the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2019, Ohrid, North Macedonia, September 22-25, 2019*. Ed. by Z. Budimac, B. Koteska. Vol. 2508. CEUR Workshop Proceedings. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2508/paper-gul.pdf (cit. on p. 63).

[GPSS15] L. Gong, M. Pradel, M. Sridharan, K. Sen. “DLint: dynamically checking bad coding practices in JavaScript”. en. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. Baltimore MD USA: ACM, July 2015, pp. 94–105. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771809. URL: https://dl.acm.org/doi/10.1145/2771783.2771809 (visited on 06/28/2022) (cit. on pp. 15, 18, 64).

[Gri18] M. Grinberg. *Flask web development: developing web applications with python*. Ö’Reilly Media, Inc.", 2018 (cit. on p. 33).

[Gui01] N. C. Guido van Rossum Barry Warsaw. *PEP 8 – Style Guide for Python Code*. PEP 8. 2001. URL: https://peps.python.org/pep-0008/ (cit. on pp. 22, 27, 29, 30, 63).

[Gul] H. Gulabovska. “Survey on Static Analysis Tools of Python Programs”. en. In: () (cit. on p. 17).

[Het05] R. Hettinger. *Weakrefs to classes that derive from str*. https://mail.python.org/pipermail/python-list/2005-March/346298.html. 2005 (cit. on p. 42).

[HMW+20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on pp. 13, 22, 26, 34).

[Kan22] S. Kansal. *wtfpython*. https://github.com/satwikkansal/wtfpython. 2022 (cit. on pp. 22, 25, 33).

[KRPS12] S. Kaufman, S. Rosset, C. Perlich, O. Stitelman. "Leakage in data mining: Formulation, detection, and avoidance". In: *ACM Transactions on Knowledge Discovery from Data* 6.4 (Dec. 2012), 15:1–15:21. ISSN: 1556-4681. DOI: 10.1145/2382577.2382579. URL: https://doi.org/10.1145/2382577.2382579 (visited on 02/04/2023) (cit. on p. 36).

[KSB20] Q. Kong, T. Siauw, A. Bayen. *Python Programming and Numerical Methods: A Guide for Engineers and Scientists*. Elsevier Science, 2020. ISBN: 978-0-12-819549-9. URL: https://books.google.de/books?id=cZ4LEAAAQBAJ (cit. on p. 14).

[Lan92] W. Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems* 1.4 (Dec. 1992), pp. 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: https://doi.org/10.1145/161494.161501 (visited on 03/13/2023) (cit. on p. 17).

[LDG+20] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, Y. Smaragdakis. "Static Analysis of Shape in TensorFlow Programs". en. In: (2020), p. 29 (cit. on pp. 13, 64).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 13, 22, 26, 35–37, 40, 62).

[Mon08] D. Monniaux. "The pitfalls of verifying floating-point computations". In: *ACM Transactions on Programming Languages and Systems* 30.3 (May 2008), 12:1–12:41. ISSN: 0164-0925. DOI: 10.1145/1353445.1353446. URL: https://doi.org/10.1145/1353445.1353446 (visited on 02/20/2023) (cit. on p. 27).

[NMY17] R. Nisbet, G. Miner, K. Yale, eds. *Handbook of Statistical Analysis and Data Mining Applications*. 2nd ed. London: Academic Press, 2017. ISBN: 978-0-12-416632-5 (cit. on p. 36).

[NS07] N. Nethercote, J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". en. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego California USA: ACM, June 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: https://dl.acm.org/doi/10.1145/1250734.1250746 (visited on 03/10/2023) (cit. on pp. 42, 64).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on pp. 22, 37, 58).

[Pit13] A. Pitrou. *PEP 442 – Safe object finalization*. PEP 442. 2013. URL: https://peps.python.org/pep-0442/#disposal-of-cyclic-isolates (cit. on p. 41).

[PSC18] G. Philipp, D. Song, J. G. Carbonell. *The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions*. en. arXiv:1712.05577 [cs]. Apr. 2018. URL: http://arxiv.org/abs/1712.05577 (visited on 01/31/2023) (cit. on pp. 22, 37, 55, 56, 58).

[PSS15] M. Pradel, P. Schuh, K. Sen. "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript". en. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 314–324. ISBN: 978-1-4799-1934-5. DOI: 10.1109/ICSE.2015.51. URL: http://ieeexplore.ieee.org/document/7194584/ (visited on 03/17/2023) (cit. on pp. 15, 64).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 13, 22, 37, 38, 62).

[RA00] G. van Rossum, D. Ascher. *PEP 207 – Rich Comparisons*. PEP 207. 2000. URL: https://peps.python.org/pep-0207/ (cit. on p. 26).

[Ros14] G. van Rossum Jukka Lehtosalo Łukasz Langa. *PEP 484 – Type Hints*. PEP 484. 2014. URL: https://peps.python.org/pep-0485/ (cit. on p. 63).

[Ros23] G. van Rossum. *cpython*. https://github.com/python/cpython. 2023 (cit. on pp. 18, 25, 29, 33, 41).

[SC07] Y. Smaragdakis, C. Csallner. "Combining Static and Dynamic Reasoning for Bug Detection". en. In: *Tests and Proofs*. Ed. by Y. Gurevich, B. Meyer. Vol. 4454. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–16. ISBN: 978-3-540-73769-8 978-3-540-73770-4. DOI: 10.1007/978-3-540-73770-4_1. URL: http://link.springer.com/10.1007/978-3-540-73770-4_1 (visited on 06/27/2022) (cit. on p. 17).

[SKBG13] K. Sen, S. Kalasapur, T. Brutch, S. Gibbs. “Jalangi: a selective record-replay and dynamic analysis framework for JavaScript”. en. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia: ACM, Aug. 2013, pp. 488–498. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491447. URL: https://dl.acm.org/doi/10.1145/2491411.2491447 (visited on 03/12/2023) (cit. on pp. 18, 42, 64).

[SN05] J. Seward, N. Nethercote. “Using Valgrind to detect undefined value errors with bit-precision”. en. In: (2005), pp. 17–30 (cit. on p. 64).

[Sri17] K. R. Srinath. “Python – The Fastest Growing Programming Language”. en. In: *International Research Journal of Engineering and Technology* 4.12 (2017), pp. 354–357 (cit. on p. 13).

[tea20] T. pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134 (cit. on pp. 13, 22, 34).

[Wet16] J. Wetzels. *Open Sesame: The Password Hashing Competition and Argon2*. arXiv:1602.03097 [cs]. Feb. 2016. DOI: 10.48550/arXiv.1602.03097. URL: http://arxiv.org/abs/1602.03097 (visited on 01/30/2023) (cit. on p. 33).

[Wit20] S. Witowski. *type() vs. isinstance()*. Tech. rep. 2020. URL: https://switowski.com/blog/type-vs-isinstance/ (cit. on p. 28).

[WWC+23] G. Wang, Z. Wang, J. Chen, X. Chen, M. Yan. “An Empirical Study on Numerical Bugs in Deep Learning Programs”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–5. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3559561. URL: https://doi.org/10.1145/3551349.3559561 (visited on 02/05/2023) (cit. on pp. 14, 22).

[XLZX16] Z. Xu, P. Liu, X. Zhang, B. Xu. “Python predictive analysis for bug detection”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 121–132. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950357. URL: https://doi.org/10.1145/2950290.2950357 (visited on 03/13/2023) (cit. on pp. 13, 52, 63).

[YBLK22] C. Yang, R. A. Brower-Sinning, G. A. Lewis, C. Kästner. *Data Leakage in Notebooks: Static Detection and Better Processes*. arXiv:2209.03345 [cs]. Sept. 2022. DOI: 10.48550/arXiv.2209.03345. URL: http://arxiv.org/abs/2209.03345 (visited on 10/24/2022) (cit. on pp. 22, 37).

[YPR+19] G. Yang, J. Pennington, V. Rao, J. Sohl-Dickstein, S. S. Schoenholz. *A Mean Field Theory of Batch Normalization*. en. arXiv:1902.08129 [cond-mat]. Mar. 2019. URL: http://arxiv.org/abs/1902.08129 (visited on 01/31/2023) (cit. on pp. 38, 58).

[ZHSJ20] J. Zhang, T. He, S. Sra, A. Jadbabaie. *Why gradient clipping accelerates training: A theoretical justification for adaptivity*. arXiv:1905.11881 [cs, math]. Feb. 2020. DOI: 10.48550/arXiv.1905.11881. URL: http://arxiv.org/abs/1905.11881 (visited on 01/28/2023) (cit. on pp. 38, 58).

All links were last followed on April 10, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature