# DynaPyt: A Dynamic Analysis Framework for Python

Aryaz Eghbali
aryaz.eghbali@iste.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Stuttgart, Germany

## ABSTRACT

Python is a widely used programming language that powers important application domains such as machine learning, data analysis, and web applications. For many programs in these domains it is consequential to analyze aspects like security and performance, and with Python's dynamic nature, it is crucial to be able to dynamically analyze Python programs. However, existing tools and frameworks do not provide the means to implement dynamic analyses easily and practitioners resort to implementing an ad-hoc dynamic analysis for their own use case. This work presents DynaPyt, the first general-purpose framework for heavy-weight dynamic analysis of Python programs. Compared to existing tools for other programming languages, our framework provides a wider range of analysis hooks arranged in a hierarchical structure, which allows developers to concisely implement analyses. DynaPyt features selective instrumentation and execution modification as well. We evaluate our framework on test suites of 9 popular open-source Python projects, 1,268,545 lines of code in total, and show that it, by and large, preserves the semantics of the original execution. The running time of DynaPyt is between 1.2x and 16x times the original execution time, which is in line with similar frameworks designed for other languages, and 5.6%–88.6% faster than analyses using a built-in tracing API offered by Python. We also implement multiple analyses, show the simplicity of implementing them and some potential use cases of DynaPyt. Among the analyses implemented are: an analysis to detect a memory blow up in Pytorch programs, a taint analysis to detect SQL injections, and an analysis to warn about a runtime performance anti-pattern.

## CCS CONCEPTS

• **General and reference** → *Cross-computing tools and techniques*; • **Software and its engineering** → **Software maintenance tools**.

## KEYWORDS

dynamic analysis, python

**ACM Reference Format:**
Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549126

## 1 INTRODUCTION

Python has evolved into one of the most important programming languages. It is widely used across many application domains, e.g., machine learning, scientific computing, server-side web applications, game development, and natural language processing. In addition to serving as a language to develop glue code and scripts, large-scale Python code bases are behind widely used websites and apps used by billions of people. Moreover, highly popular frameworks and libraries, such as pandas, numpy, Django, and Pytorch, are implemented in Python. In 2019, Python has become the second-most popular of all languages across code hosted at GitHub.[1]

The popularity of the language, combined with its dynamic language features, such as the absence of statically declared types or the ability to add and remove object attributes at runtime, make Python a prime target for dynamic analysis. Indeed, several dynamic analyses have been proposed, e.g., to detect bugs [36], to enforce differential privacy [1], or to slice programs [6]. However, compared to other popular dynamic languages, e.g., JavaScript, which have seen a surge of dynamic analyses over the past few years [3], dynamic analysis for Python has not yet become mainstream.

We argue that the sparsity of dynamic analyses for Python is, at least partially, due to a lack of suitable frameworks to build on. To ease the implementation of dynamic analyses, general-purpose dynamic analysis frameworks have been created for other languages, e.g., Pin [18] and Valgrind [21] for native binaries, DiSL [20] and RoadRunner [10] for Java, Jalangi [29] for JavaScript, and Wasabi [17] for WebAssembly. These frameworks all share the idea of providing to analysis developers an easy to implement interface to observe events during a program's execution, typically in the form of callbacks or hooks an analysis implements. These hooks are then invoked by the analysis framework during the program's execution, typically by instrumenting the program before its execution. Yet, despite the importance of Python, there currently is no such framework for Python, hindering the development of dynamic analyses.

This paper presents DynaPyt, the first general-purpose dynamic analysis framework for Python. From the perspective of an analysis developer, the framework offers a set of hooks into specific kinds of runtime events, such as function calls, writes of object attributes, and control flow decisions. Given an analysis that implements some of these hooks and a program to analyze, DynaPyt instruments the program's code via source-to-source transformation. The instrumentation code wraps the existing code with calls into the DynaPyt

[1]https://octoverse.github.com/#top-languages-over-the-years

**Table 1: Comparison of general-purpose dynamic analysis frameworks.**

| Framework | Target language | Pay per use | Nb. of hooks | Hierarchy of hooks | Behavior manipulation |
|---|---|---|---|---|---|
| Jalangi2 | JavaScript | ✓ | 28 | ✗ | ✓ |
| Wasabi | WebAssembly | ✓ | 23 | ✗ | ✗ |
| `sys.settrace` | Python | ✗ | 1 | ✗ | ✗ |
| DynaPyt | Python | ✓ | 97 | ✓ | ✓ |

runtime engine, which dispatches the runtime events to the corresponding analysis hooks. By default, the instrumentation preserves all behavior of the original program. However, DynaPyt also allows analyses to manipulate the behavior, e.g., to modify the value that gets written into an object attribute or to flip the outcome of a control flow decision.

DynaPyt addresses several challenges that prior dynamic analysis frameworks handle only partially or not at all. One of them is the fact that there are many different runtime events of potential interest. For example, there are nine kinds of memory accesses, eleven different control flow events, and 15 kinds of binary operations. To enable an analysis writer to easily select events of interest, DynaPyt arranges a total of 97 events into an *event hierarchy* and allows the analysis writer to select events at an appropriate level of abstraction. Another challenge is to impose as little overhead as needed to perform a specific analysis. To this end, DynaPyt follows a *pay-per-use* principle, which adds instrumentation code only when needed to keep track of exactly those runtime events an analysis is interested in. As a result, the runtime overhead imposed by the framework scales with the complexity of the analysis.

To put DynaPyt's contributions in perspective, Table 1 compares existing dynamic analysis frameworks and our work. Jalangi [29] and Wasabi [17] are two frameworks for other languages, which have inspired our design. Both lack the event hierarchy that DynaPyt offers, but instead offer only a "flat" set of 23 and 28 hooks for analyses to implement, respectively. The only other technique available for Python is `sys.settrace`, a built-in function offered by the CPython implementation of the language. It allows for observing every executed opcode, but does not follow the pay-per-use principle. Moreover, it focuses only on observing behavior, but in contrast to DynaPyt lacks the ability to manipulate an execution. We empirically compare with `sys.settrace` in Section 3 and provide a detailed discussion of it in Section 4.

In addition to language-agnostic challenges, DynaPyt needs to address unique challenges due to specific features of Python and the characteristics of their semantics. For example, these challenges include the way `self` and `super()` are resolved, which can easily lead to undefined references in instrumentation code, constraints on the order of imports, which must be considered when adding imports of the DynaPyt runtime engine, and name mangling, which implicitly modifies the names of "private" class attribute.

Our evaluation applies DynaPyt to 9 popular Python projects and other real-world code. We show that the instrumentation does not change the semantics of a program but is faithful to the original

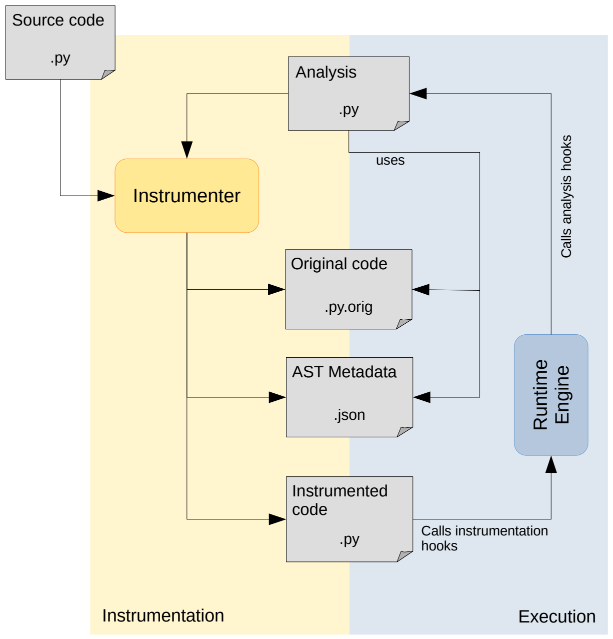

**Figure 1: Overview of the DynaPyt's framework.**

execution. Evaluating the performance of the instrumented code, we show that the pay-per-use principle leads to an overhead that scales with the runtime events an analysis is interested in, and that DynaPyt is significantly faster than `sys.settrace`. We also implement several analyses on top of DynaPyt: a branch coverage and a call graph analysis, which each take only a few lines of code; a dynamic bug detector that identifies shape mismatches in deep learning code; and a taint analysis that reveals real-world vulnerabilities.

In summary, this paper contributes the following:

- The first general-purpose dynamic analysis framework for Python.
- An event hierarchy that allows client analyses to register for selected events of interest while adding runtime overhead only for those events ("pay-per-use").
- 6 client analyses that showcase the usefulness of DynaPyt.
- Empirical evidence that the framework allows for analyzing real-world Python code, while preserving the original program semantics and imposing overhead proportional to the selected events of interest.

## 2 APPROACH

### 2.1 Overview

Figure 1 gives an overview of the DynaPyt framework. The approach consists of two phases: *instrumentation* and *execution*. In the instrumentation phase, an instrumenter augments the original source code with code to observe and manipulate the program's execution. The instrumented code calls *instrumentation hooks*, i.e., functions provided by the *runtime engine* of DynaPyt, to notify the engine about specific runtime events. Besides the original code, the instrumenter also takes the analysis written by a user as an input, to instrument only those parts of the code that are of interest to the analysis. In addition to the instrumented code, the instrumenter stores the original code and a JSON file with information on the transformed code, both of which may be later used by the analysis.

In the execution phase, the dynamic analysis is performed alongside the program's execution. Whenever the instrumented code notifies the runtime engine about a runtime event, the engine dispatches the event to the analysis. To this end, the engine calls *analysis hooks*, i.e., API functions implemented by the analysis. Whenever an analysis hook gets called, the analysis can observe, and optionally also manipulate, the corresponding runtime behavior.

The remainder of this section presents the analysis hooks (Section 2.2), the instrumentation (Section 2.3), and the runtime engine (Section 2.4) in detail.

## 2.2 Hierarchy of Runtime Events

*2.2.1 Motivation.* One of our key contributions is to arrange the analysis hooks an analysis can implement into a hierarchy of runtime events. The motivation for this event hierarchy is twofold. First, it enables the analysis writer to precisely specify the runtime events of interest. Previous frameworks provide a fixed set of hooks at a fixed level of granularity, and the burden of matching the framework to the analysis lies on the analysis writer. For example, implementing an analysis interested only in for-loops in Jalangi [29] requires to observe all control flow statements and to check each for whether it is a for-loop. Second, precisely knowing what runtime events an analysis cares about is a prerequisite for limiting the analysis overhead to what is needed by the analysis.

*2.2.2 Design.* Instead of offering a fixed set of analysis hooks at a fixed level of granularity, DynaPyt provides fine-grained hooks in addition to multiple layers of abstraction that combine related subsets of those hooks. The basic idea is to arrange the analysis hooks into a hierarchy. The root of this hierarchy is a generic "runtime event" hook that captures all events DynaPyt supports. The leaves of the hierarchy are specific kinds of runtime events, such as entering a for-loop, performing a bitwise left-shift operation, or writing an object attribute. Figure 2 shows the complete hierarchy of analysis hooks available to be implemented by an analysis. Hooks that share their name with a keyword in Python are prepended by an underscore (i.e. the analysis hook for `raise` is `_raise`). Overall, there are 97 analysis hooks that cover different kinds of operations, control flow events, memory accesses, and special events, such as the beginning and end of the execution.

All analysis hooks, except `begin_execution`, `end_execution`, and `uncaught_exception`, receive at least two parameters. The first parameter is the path to the original source code, and the second parameter is an integer identifier that specifies the location of the relevant code in the original source. In addition, each analysis hook receives parameters specific to itself. For example, the `binary_operation` hook receives the operator, the operands, and the result of the operation. For analysis hooks higher up in the hierarchy, the parameters are those relevant to all runtime events captured by the hook. For example, the `operation` hook, which captures all binary operations, unary operations, and comparisons, receives an operator argument and a list of operands.

- runtime_event
  - begin_execution, end_execution
  - uncaught_exception
  - literal
    - integer, boolean, string, dictionary, ...(4 more)
  - operation
    - binary_operation
      - augmented_assign
        - bit_and_assign, add_assign, ...(11 more)
      - add, divide, bit_and, ...(12 more)
    - unary_operation
      - bit_invert, minus, not, plus
    - comparison
      - equal, greater_than, in, is_not, ...(6 more)
  - control_flow_event
    - conditional_control_flow
      - enter_while, exit_while, ...(4 more)
      - enter_control_flow
        - enter_if, enter_for, enter_while
      - exit_control_flow
        - exit_if, exit_for, exit_while
    - raise, enter_try, pre_call, continue, ...(6 more)
    - function_exit
      - function_exit, return, yield
  - memory_access
    - read
      - read_identifier, read_subscript, read_attribute
    - write, delete

**Figure 2: Hierarchy of analysis hooks.**

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = defaultdict(lambda: 0)

    def enter_control_flow(self, f, iid, condition):
        self.branches[(iid, condition)] += 1
```

**Listing 1: Branch coverage analysis.**

*2.2.3 Examples of Analyses.* This hierarchy of analysis hooks allows analysis writers to choose the exact subset of events they are interested in, and to implement the analysis at an appropriate abstraction level. For example, Listing 1 shows a branch coverage analysis implemented with DynaPyt in only nine lines. The analysis imports and extends the `BaseAnalysis` class, and then implements the `enter_control_flow` analysis hook, which is called on

```python
from .BaseAnalysis import BaseAnalysis

class KeyInListAnalysis(BaseAnalysis):
   def __init__(self):
      self.threshold = 100

   def _in(self, f, iid, left, right, result):
      if (isinstance(right, list) and
              len(right) > self.threshold):
         print('Performance warning')

   def not_in(self, f, iid, left, right, result):
      if (isinstance(right, list) and
              len(right) > self.threshold):
         print('Performance warning')
```

**Listing 2: Analysis to detect the performance anti-pattern of searching a key inside a long list.**

```python
from .BaseAnalysis import BaseAnalysis

class ManipulateExec(BaseAnalysis):
   def enter_if(self, f, iid, cond_value):
       if random() < 0.5:
           return not cond_value

   def write(self, f, iid, old_val, new_val):
       if new_val == 23:
           return 42
```

**Listing 3: Analysis to detect the performance anti-pattern of searching a key inside a long list.**

any branch. Whenever a branching decision is made in the program execution, the analysis updates its branches dictionary, which maps the code location of the branch and the outcome of the branching decision to the number of times this combination has been observed.

Listing 2 shows another example analysis. It detects instances of a performance anti-pattern caused by searching a key in a list instead of a dictionary or set[2]. The analysis uses only the _in hook, which is invoked whenever the program checks whether a value is an element of a data structure, e.g., when checking val in myList. If the right-hand operand of in is larger than a threshold, the analysis reports a warning about the inefficient operation.

As a third example, Listing 3 illustrates two ways of how an analysis can manipulate executions of a program. First, the enter_if hook randomly flips the outcome of any conditional evaluated in an if branch. Instead of randomly flipping conditionals, a more sophisticated version of this analysis could systematically change conditionals, e.g., to implement concolic execution [12, 30] or forced execution [16, 23]. Second, the write hook manipulates values written into variables by replacing any value 23 with the value 42. A more sophisticated version of this analysis could systematically inject failures into a program or test robustness against adversarial inputs.

[2]Inspired by https://docs.quantifiedcode.com/python-anti-patterns/performance/using_key_in_list_to_check_if_key_is_contained_in_a_list.html

## 2.3 Source-to-Source Instrumentation

In this section we describe the instrumentation performed of DynaPyt, which wraps code fragments in the given source code into calls to instrumentation hooks. We describe interesting aspects of the instrumentation based on a relevant subset of the Python language (Section 2.3.1) and a selection of transformation rules (Section 2.3.2). These rules are applied by selectively rewriting those parts of the AST of the code that are relevant for the analysis (Section 2.3.3).

*2.3.1 Grammar.* While our implementation covers the full Python language, we focus on a subset of Python, called *Mini-Python*, in the paper. Figure 3 shows the grammar of Mini-Python, which covers the basic features of the language and specific features that pose interesting challenges for the instrumentation.

*2.3.2 Code Transformation Rules.* The instrumenter modifies statements and expressions of the given source code through a series of code transformation rules. Table 2 presents some of these transformation rules, with minor simplifications to ease the presentation. Each rule injects one or more calls of instrumentation hooks, i.e., calls that inform the runtime engine about particular runtime events. For example, Rule 1 adds a call of the _int_ instrumentation hook, which notifies the runtime engine that an integer literal gets evaluated. As another example, Rule 5 adds calls of two instrumentation hooks related to for-in-loops: the _gen_ hook, which indicates that a generator expression produces another value, and the _exit_for_ hook, which indicates that the loop has terminated.

All instrumentation hooks take at least two arguments. The first argument is the path to the file with the original source code, which is useful to extract the AST at runtime. The second argument specifies the location of the instrumented statement or expression, called *instruction id* or short iid, which is used to locate the appropriate node of the AST at runtime. These two arguments are provided as utilities for the implementation of analyses, and are not used by the runtime engine. For a compact presentation, Table 2 shows these two arguments as f and iid.

The following discusses in more detail some of the transformation rules and other specific language features that DynaPyt must handle.

*Delayed evaluation of expressions.* Whenever an expression gets evaluated in the program, DynaPyt wants to not only give an analysis the opportunity to observe this event, but also to modify the result of the evaluation. To this end, the instrumentation wraps each expression into a lambda function that, when invoked, evaluates the expression and returns its result. Rule 2 shows a simple example, where the original code reads the value referred to by a name, e.g., a local variable. The instrumented code replaces the name with a call of the _read_ instrumentation hook, and passes a lambda function into the hook, which reads and returns the name. The runtime engine hence can decide when to evaluate the expression,

**Table 2: Examples of code transformations rules applied during instrumentation.**

| Id | Original code | Instrumented code | Explanation |
|---|---|---|---|
| 1 | *IntLiteral* | `_int_(f, iid,` *IntLiteral* `)` | Evaluating an integer literal. Similar for other literals. |
| 2 | *Name* | `_read_(f, iid, lambda:` *Name* `)` | Reading the value referred to by a name. |
| 3 | `del` *Name* | `_delete_(f, iid, [(locals() if "` *Name* `" in locals() else globals(), "` *Name* `")])` | Deleting a variable. Checks if the variable to be deleted is local or global, and then passes the appropriate dictionary for deletion. |
| 4 | `del` Expression `.` *Name* | `_delete_(f, iid, [(` Expression `, "` *Name* `")])` | Deleting an attribute. `_delete_` takes a list of tuples as the targets to delete for supporting multiple delete targets. |
| 5 | `for` *Name* `in` Expression `:` Statement+ | `for` *Name* `in _gen_(f, iid,` Expression `):` Statement+ `else:` `_exit_for_(f, iid)` | The `_gen_` method yields the values generated by the Expression. It also calls the `_enter_for_` internally. |
| 6 | `break` | `if _break_(f, iid):` `break` | Wrapping into an if-statement allows the analysis to modify the control flow decision. |
| 7 | *Name0* `(` Expression0 `,` *Name1* `=` Expression1 `)` | `_call_(f, iid, lambda:` *Name0* `, [` Expression0 `], {"` *Name1* `":` Expression1 `})` | The arguments of a function call are split into positional arguments, passed as a list, and keyword arguments, passed in a dictionary. |
| 8 | Expression0 `and` Expression1 | `_binary_op_(f, iid, lambda:` Expression0 `, 13, lambda:` Expression1 `)` | Binary expression. Each operator is mapped to a number (13 for `and`). |

```python
1 class X:
2     a = 0
3     b = a+1
```

**Listing 4: Example for special case of delayed evaluation expressions.**

and an analysis can overwrite the original result of the evaluation, which will then be returned by `_read_`.

*Accessing attribute names in class methods.* Class methods in Python can only access the attributes of an object instance if the first argument of the method is the object instance itself, referred to as `self`. Naively delaying accesses of object attributes using lambda functions as described above would cause `NameErrors`. DynaPyt avoids this problem by adding the names used in the lambda function as parameters with default values. Listing 4 shows an example of this situation, where the binary operation on line 3 is transformed to `_bin_op_(f, iid, lambda a=a: a, 0, lambda: 1)`. The instrumentation performs this transformation only for accesses of names with class scope.

*Resolving* `super`*.* As illustrated by Rule 7, DynaPyt wraps function calls into the `_call_` instrumentation hook and then performs the call in the runtime engine. However, some functions do not behave the same when called from a different call site. For example, `super()`, which resolves to the parent class, would fail if called inside the runtime engine. To remedy this problem, the instrumentation add arguments to calls of `super`, turning the call into `super(C, self)`, where `C` is the enclosing class, and `self` is the object instance itself.

*Special functions.* Other functions, e.g., `exec`, `eval`, `local`, and `global`, would also behave abnormally if called from the runtime engine. The reason is that these functions rely on the current stack frame, which depend on the call site. For built-in functions, we evaluate the function at the call site and pass the resulting value to the runtime engine. Therefore, for these functions, the `pre_call` hook is not called and only the `post_call` hook is executed.

*The* `__future__` *imports.* Import statements in Python can appear anywhere in the code. However, `__future__` imports are an exception to this rule, as they can appear only at the top of the module, before any other code. As DynaPyt's instrumenter wraps the whole file in a `try` statement to capture any uncaught exceptions, these imports must be left outside of this `try` block.

*Formatted strings.* In Python, string literals that are preceded by `f` can use expressions inside curly brackets, which will be evaluated to string values at runtime. For example, `f"{2+3} is 5"` will evaluate to "5 is 5". As formatted strings themselves are also expressions, or can be part of more complex expressions, Python programs can have

Program: Statement*

Statement: SimpleStmt | CompoundStmt

SimpleStmt: AssignmentStmt | DeleteStmt | BreakStmt | ContinueStmt | ReturnStmt

CompoundStmt: IfStmt | ForInStmt | WhileStmt | TryStmt | FunctionDefStmt

Expression: Literal | Call | Access | BinaryOp | UnaryOp | ComparisonOp

AssignmentStmt: AssignTargets "`=`" Expression

Access: *Name* | Attribute | Subscript

AssignTargets: Access ["=" Access]*

Attribute: Expression "`.`" *Name*

Subscript: Expression "`[`" *Slice* ["`,`" *Slice*]* "`]`"

Literal: *IntLiteral* | *FloatLiteral* | *ImaginaryLiteral* | *StringLiteral* | *BooleanLiteral*

DeleteStmt: "`del`" Access ["`,`" Access]*

Call: Access "`(`" [[*Name* "="]? Expression]? ["`,`" [*Name* "="]? Expression]* "`)`"

BinaryOp: Expression *BinaryOperator* Expression

UnaryOp: *UnaryOperator* Expression

ComparisonOp: Expression [*Comparator* Expression]+

IfStmt: "`if`" Expression "`:`" Statement+ ["`elif`" Expression "`:`" Statement+]* ["`else:`" Statement+]?

ForInStmt: "`for`" Access "`in`" Expression "`:`" Statement+ ["`else:`" Statement+]?

WhileStmt: "`while`" Expression "`:`" Statement+ ["`else:`" Statement+]?

BreakStmt: "`break`"

ContinueStmt: "`continue`"

FunctionDefStmt: "`def`" *Name* "`(`" [*Name* ["=" Literal]?]* "`):`" *DocString*? Statement+

ReturnStmt: "`return`" Expression?

TryStmt: "`try:`" Statement+ "`except`" [*Name* ["`as`" *Name*]?]? "`:`" Statement+ ["`finally:`" Statement+]?

**Figure 3: Grammar of Mini-Python. Non-terminals are represented with the regular font, terminals in *italic font*, and code tokens are show in "`quotations and typeset font`"**

arbitrarily deeply nested formatted strings. However, there are only two symbols to specify strings, namely single and double quotations. Since some of our instrumentation depends on passing string literals to the runtime engine, naively wrapping nested formatted strings into instrumentation hooks would yield syntactically invalid code. To avoid this problem, DynaPyt instruments only the expressions on the first level of formatted strings and leaves nested formatted strings uninstrumented.

*Docstrings.* Docstrings are string literals that appear at the top of the body of a module, class, or function, and are used mostly for documentation purposes. However, the docstrings are stored in the `__doc__` attribute of the corresponding module, class, or function during runtime. Modules like doctest use the docstring attributes to test functions. To avoid disrupting the functionality that depends on these strings, DynaPyt's instrumenter ignores these literal.

#### 2.3.3 *Selective Instrumentation.*

DynaPyt applies all code transformation rules via AST-based code rewriting. The instrumenter parses each source code file into an AST and then visits each node in the order in which its tokens appear in the source code. Each transformation rule applies to a specific kind of AST node, and DynaPyt applies transformations upon leaving the node. Our implementation is based on the LibCST library and its Transformer class.[3]

Following the pay-per-use principle, an important property of DynaPyt is to not necessarily instrument all AST nodes. Instead, DynaPyt's instrumenter first determines the set of instrumentation hooks that are required for a given analysis based on the hierarchy of analysis hooks. Then, it performs a selective instrumentation of those instrumentation hooks. As we show in the evaluation, for analyses that only trace a subset of all possible runtime events, selective instrumentation has a significant impact on execution time.

For example, recall the example analysis in Listing 1, which implements only the `enter_control_flow` analysis hook for tracking branches. Given this analysis, DynaPyt inserts instrumentation hooks for keeping track if-statements, for-loops, and while-loops, but none of the many other instrumentation hooks that the framework supports.

Orthogonal to selectively instrumenting only those code fragments that are relevant for the runtime events an analysis is interested in, DynaPyt also supports instrumenting only parts of a larger project. Users of DynaPyt can, e.g., instrument all files of project, include or exclude the code of specific libraries, or instrument only specific files of interest. Because the instrumentation does not change the interfaces of modules, instrumented and non-instrumented code are fully compatible.

#### 2.3.4 *Alternatives to Source-Level Instrumentation.*

As a potential alternative to the source-level instrumentation performed by DynaPyt, we also considered bytecode-level instrumentation, but ultimately decided against it for two reasons. First, source-level analysis allows for writing analyses on a higher level of abstraction. Each of the API hooks offered by DynaPyt corresponds to a construct in the Python programming language, whereas bytecode uses a lower-level representation of values and operators that developers are less familiar with. Second, source-level instrumentation makes DynaPyt independent of specific interpreters and compilers. In contrast to other languages, e.g., Java bytecode or WebAssembly, Python's bytecode is not standardized. If we, e.g., built upon CPython's bytecode format, then DynaPyt would be incompatible with other runtime environments, e.g., PyPy[4], Jython[5], or GraalPy[6].

[3] https://libcst.readthedocs.io/en/latest/
[4] https://www.pypy.org/
[5] https://www.jython.org/
[6] https://www.graalvm.org/python/

### 2.4 Runtime Engine

DynaPyt's runtime engine connects the instrumented source code of the program with the actual analysis. Specifically, the engine has three tasks. First, it performs those parts of the program behavior that are replaced by calls to instrumentation hooks. For example, binary operations are replaced by the `_binary_op_` hook (see Rule 8 in Table 2), by passing the operands and operators to the runtime engine, which then performs the binary operation at runtime.

Second, the runtime engine dispatches calls of instrumentation hooks from the program to the analysis hooks implemented by the analysis, and passes any values modified by the analysis back to the program. DynaPyt analyses can implement any subset of the analysis hooks in the hierarchy, even both an ancestor and some descendants. This feature can be useful when implementing a general behavior for a class of runtime events and adding an exception for some of sub-events. In this case, the analysis can implement the general behavior in the ancestor analysis hook, and the exceptional behavior in the sub-event's analysis hook. The runtime engine first calls the analysis hooks closest to the root of the hierarchy tree and then traverses towards the leaves. In cases that the analysis modifies the execution, the modifications by the lower-level hooks can overwrite the ones from the higher levels in the hierarchy. For example, if an analysis wants to negate all comparison operations, except for `is` and `is not`, then it can implement the `comparison` hook to negate the result, and implement the `_is` and `is_not` hooks to return the original result.

Third, the runtime engine serves as the main entry point of a program analyzed with DynaPyt. When invoked, it initializes the analysis and, if implemented, calls the `begin_execution` analysis hook. It then executes the original entry point of the program under analysis, and finally, if implemented, calls the `end_execution` analysis hook.

In the following, we discuss some non-obvious aspects of the runtime engine.

*Logical operations.* Logical `and` and `or` take two operands, but the second operand might not be evaluated based on the value of the first operand due to short-circuited evaluation. If the first operand is true, the second operand of `or` is not evaluated. If the first operand is false, the second operand of `and` is not evaluated. As the evaluation of the operands is performed in the runtime engine, it implements short-circuited evaluation as it would happen in the original program.

*Private name mangling.* As any other evaluation of expressions, accesses of object attributes are performed by the runtime engine. Python provides "private"[7] attributes of objects through specific naming conventions of attributes. If an attribute name starts with two underscores (`__`) and ends with fewer than two underscores, then the attribute is stored by prepending the name of the class to the attribute name. For example, if class `X` has a private attribute `__y`, then the attribute will be stored under the name `_X__y`. Since Python accesses object attributes using the name of the attribute as a string, when requesting the attribute using `getattr`, the runtime engine must prepend the class name to the attribute name. However, because of inheritance, DynaPyt must search for the correct class name, as shown in Listing 5. The algorithm starts from the class of the object being accessed and moves up the class hierarchy until it reaches the class that owns the private attribute.

```python
def get_private_attribute(base, attr):
  parents = [type(base)]
  while len(parents) > 0:
      cur_par = parents.pop()
      try:
         cur_name = cur_par.__name__
         cur_name = cur_name.lstrip('_')
         val = getattr(base, '_'+cur_name+attr)
      except AttributeError:
         parents.extend(list(cur_par.__bases__))
      else:
         return val
  raise AttributeError()
```

**Listing 5: Resolution of private name mangling.**

[7]The attributes are private by convention, but not truly protected from other accesses.

## 3 EVALUATION

Our evaluation applies DynaPyt, and several example analyses implemented on top of it, to 9 real-world Python projects with total of 1,268,545 lines of code. We address the following research questions:

**RQ1** How efficient is DynaPyt when instrumenting source code?
**RQ2** Does an instrumented program remain faithful to the semantics of the original program?
**RQ3** How complex is the implementation of analyses built on top of the framework?
**RQ4** What runtime overhead does DynaPyt impose when performing an analysis?

### 3.1 Experimental Setup

*Benchmark programs.* To answer the above questions we need a set of Python programs to analyze. We gather this set from popular open-source Python on GitHub. We search for projects implemented in Python and sort them by the number of their stars. Then, from the top of the list we choose projects that cover different applications domains. During the selection process, we keep only repositories with test suites that can be run with the `pytest` framework. This criteria is only for simplifying the evaluation by avoiding to deal with different test execution scripts. As a case study for RQ4, we also use an open-source Python project intended to show vulnerabilities. The list of selected projects is available in Table 3.

*Experimental environment.* All experiments are run on a laptop with an Intel Core i7-8565 CPU (8 cores ×1.8 GHz) and 16 GB of RAM, running Ubuntu 20.04. We use Python 3.9.0, except for the "anxolerd/dvpwa" project used in RQ4, which requires Python 3.6.

### 3.2 Efficiency of Instrumentation (RQ1)

The time it takes to instrument code can be a determining factor on how often new code can be analyzed and how many analyses can be performed in a given time budget. We evaluate the efficiency of the

**Table 3: List of projects used for evaluation.**

| # | Repository | Application domain | Instrument time (mm:ss) | Python files | Lines of Code |
|---|---|---|---|---|---|
| 1 | ansible/ansible | Automation framework | 06:59 | 2,188 | 176,173 |
| 2 | django/django | Web framework | 14:07 | 3,603 | 318,602 |
| 3 | keras-team/keras | Deep learning framework | 05:41 | 678 | 155,407 |
| 4 | pandas-dev/pandas | Data analysis library | 12:32 | 2,727 | 358,195 |
| 5 | psf/requests | HTTP library | 00:16 | 54 | 6,370 |
| 6 | Textualize/rich | Text tool | 00:57 | 178 | 24,362 |
| 7 | scikit-learn/scikit-learn | Machine learning framework | 06:52 | 1,419 | 180,185 |
| 8 | scrapy/scrapy | Web crawler | 01:49 | 505 | 37,181 |
| 9 | nvbn/thefuck | Commandline tool | 01:21 | 620 | 12,070 |

**Table 4: Results from running the TraceAll analysis on the test suites of projects in our dataset.**

| | Passing tests | |
|---|---|---|
| # | # without instrumentation | % after instrumentation |
| 1 | 2,862 | 99.4% |
| 2 | 191 | 98.4% |
| 3 | 363 | 99.7% |
| 4 | 136,898 | 99.8% |
| 5 | 374 | 100.0% |
| 6 | 545 | 99.6% |
| 7 | 9,400 | 97.8% |
| 8 | 1,841 | 99.6% |
| 9 | 1,798 | 100.0% |

instrumentation by measuring the time the instrumenter takes to transform all Python files in a project for the TraceAll analysis. The TraceAll analysis implements all possible hooks, hence forcing the instrumenter to instrument all expressions and statements, which makes it the most expensive instrumentation. Table 3 shows the instrumentation time for each project, averaged over five runs. The time taken ranges between a few seconds and several minutes, depending on the size of the project. It is heavily correlated with the number of files (Pearson correlation coefficient 0.96) and with the number of lines of code (Pearson correlation coefficient 0.99).

In a regression analysis scenario, i.e., where a DynaPyt-based analysis runs on updated versions of a program, the instrumentation time would be even less than reported in Table 3: Since DynaPyt instruments files independently, source code modifications do not need a full re-instrumentation, but only to re-instrument the modified code.

## 3.3 Faithfulness to Original Execution (RQ2)

To test that our transformations and hook routines do not interfere with the semantics of the original execution, we instrument and run the TraceAll analysis on our set of open-source Python projects. Table 4 shows how many tests we run per project, and how many of them are still passing after the instrumentation.

Depending on the project, between 98% and 100% of all originally passing test cases also pass after the instrumentation. Test failures are mostly due to two reasons. One set of tests fail because they use the execution stack and make assertions based on its content. Because DynaPyt adds additional calls, such as switching to the runtime engine and calling analysis hooks, the contents of the stack are not as expected. The rest of test failures occur because of edge cases not handled by DynaPyt. These issues are under investigation to be fixed in future releases.

## 3.4 Analyses Written on Top of DynaPyt (RQ3)

As a general-purpose dynamic analysis framework, DynaPyt enables the implementation of dynamic analyses for a wide range of software engineering tasks. The following presents several analyses we implement to illustrate the abilities of the framework, roughly sorted by increasing complexity. Table 5 gives an overview of the analyses.

*BranchCoverage.* This analysis, shown in full in Listing 1, counts how many times each branch of a program is executed and how often it leads to each of the two possible outcomes of the branching decision. Because the event hierarchy captures all branching events under a single API hook, `enter_control_flow`, the analysis only needs to implement this one hook. The results are stored in a dictionary mapping a branch location and a condition value to the number of times this event was observed.

*CallGraph.* This analysis creates a dynamic call graph. Nodes in the graph are functions, and an edge indicates that one function has been observed to call another function. The analysis can be implemented with only the `post_call` analysis hook within a few lines of code.

*KeyInList.* This analysis warns about a performance anti-pattern. In Python, checking whether a key exists in a collection, e.g., a list, dictionary, or set, is done using the `in` operator. Yet, the performance of this operator depends on the structure of the collection. Specifically, searching a key in a list is a linear operation, whereas performing the same search in a set or a dictionary is sub-linear and therefore more efficient.

Listing 6 shows an instance of this problem, which checks for different query words whether these words are in a list `d`. Each time the `query in d` expression gets evaluated, the entire list `d` gets traversed. A more efficient variant of this example would compute `set(d)` before entire the loop and then check if `query` is in this set.

**Table 5: Example analyses written on top of DynaPyt.**

| Name | Description | Analysis hooks | LoC |
|---|---|---|---|
| BranchCoverage | Measures how often each branch gets covered (Listing 1) | 1 | 6 |
| CallGraph | Computes a dynamic call graph | 1 | 19 |
| KeyInList | Warns about performance anti-pattern of linearly search through a list | 2 | 10 |
| MLMemory | Warns about memory leak issues in deep learning code | 4 | 29 |
| SimpleTaint | Taint analysis useful to, e.g., detect SQL injections | 7 | 53 |
| AllEvents | Implements the `runtime_event` analysis hook to trace all events | 1 | 4 |

```
1 # d is the list of words read from a large file
2 # queries is a list of words to check
3 for query in queries:
4     if query in d: # Slow
5         print(f'Found {query}')
```

**Listing 6: A code piece showing the slow operation of looking up a key in a list.**

```
total_loss = 0
for i in range(10000):
   optimizer.zero_grad()
   output = model(input)
   loss = criterion(output)
   loss.backward()
   optimizer.step()
   total_loss += loss # Operation history is kept
```

**Listing 7: A code piece showing a memory blow up problem in PyTorch.**

The KeyInList analysis warns about the inefficiency of searching a long list for a key. Listing 2 shows the implementation of the analysis, which implements two of DynaPyt’s analysis hooks. It is worth noting that the `in` in Line 3 of Listing 6 does not trigger a warning because the `_in` hook only tracks the comparison operator `in`, but not for-in loops. Future work could implement checks for a comprehensive suite of dynamically detectable programming anti-patterns, in the spirit of existing techniques for other dynamic languages [13, 14], with the KeyInList analysis as a starting point.

*MLMemory.* This analysis warns about a memory leakage in deep learning implementations. In Pytorch, a popular machine learning framework, computations on variables that require gradients keep a history of the operations for gradient calculations. Therefore, such operations should be avoided in loops, e.g., the training loop of a neural model, to not fill up the memory with histories. Listing 7 shows an example that illustrates the problem, which we adapt from the FAQ of the PyTorch documentation. The example adds the `loss` to `total_loss` in 10,000 iterations. If the code keeps the loss, e.g., to compute some statistics about the learning process, the code will also keep the history of all operations that lead to it, because the `loss` variable requires gradients by default. To avoid this unnecessary memory blow-up, the code should access the loss value without keeping its gradients, e.g., by writing `total_loss += float(loss)`.

The MLMemory analysis, shown in Listing 8 tracks such operations and warns if they are used in a loop. More specifically, the analysis tracks the propagation of `requires_grad` by implementing the `binary_operation` and `write` hooks. The analysis checks if such a propagation happens inside a loop more than a specific number of times, using the `enter_control_flow` and `exit_control_flow` hooks. If the threshold is exceeded, a warning gets reported. Implementing this analysis in DynaPyt requires only 29 lines of code, which implement four analysis hooks.

*SimpleTaint.* This analysis showcases the potential of DynaPyt to detect vulnerabilities and check other security properties. We implement a simple train analysis that tracks if data flows from a specific source, e.g., a parameter given to a function, to a specific sink, e.g., the argument of another function. The analysis uses seven analysis hooks and is about 50 lines of code long.

To validate the taint analysis, we apply it to the “Damn Vulnerable Python WebApp”[8], a project that implements a Python web application with multiple security vulnerabilities. One of these vulnerabilities is an SQL injection, which we try to detect using the SimpleTaint analysis by configuring the source and sink appropriately. The analysis detects the SQL injection as an invalid taint flow and reports a warning.

*TraceAll.* This analysis, which is used in RQ1, RQ2, and RQ3, implements all analysis hooks. It illustrates the extreme example of an analysis that keeps track of all runtime events that happen during an execution. Even though not directly useful for a specific task, the TraceAll analysis provides an upper bound on the amount of instrumentation code the framework may add and the runtime overhead that gets imposed as a result.

As a proxy measure for the complexity of implementing the above analyses, Table 5 shows the number of analysis hooks and the number of lines of non-comment, non-empty lines of code per analysis. All of them require at most a few dozens of lines of code, illustrating that DynaPyt enables the development of a range of dynamic analyses with little effort.

## 3.5 Runtime Overhead (RQ4)

We measure the runtime overhead of DynaPyt with three analyses. The first analysis, TraceAll, is the most expensive one, because it

[8]https://github.com/anxolerd/dvpwa

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class MLMemoryAnalysis(BaseAnalysis):
    def __init__(self) -> None:
        super().__init__()
        self.in_ctrl_flow = []
        self.threshold = 3
        self.memory_leak = defaultdict(lambda: 0)
        self.last_opr = None

    def enter_control_flow(self, dyn_ast, iid,
            cond):
        self.last_opr = None
        if ((len(self.in_ctrl_flow) > 0) and
                (self.in_ctrl_flow[-1] != iid)):
            self.in_ctrl_flow.append(iid)

    def exit_control_flow(self, dyn_ast, iid):
        self.last_opr = None
        self.in_ctrl_flow.pop()

    def binary_operation(self, dyn_ast, iid, opr,
            left, right, res):
        if ((len(self.in_ctrl_flow) > 0) and
                right.requires_grad):
            self.last_opr = iid
        else:
            self.last_opr = None

    def write(self, dyn_ast, iid, left, right):
        if ((len(self.in_ctrl_flow) > 0) and
                right.requires_grad and
                (self.last_opr is not None)):
            cur = (iid, self.in_ctrl_flow[-1])
            self.memory_leak[cur] += 1
            if (self.memory_leak[cur] >
                    self.threshold):
                print('Memory issue detected')
                exit(1)
        self.last_opr = None
```

**Listing 8: Analysis to detect a memory issue in PyTorch.**

incurs the overhead of an instrument hook and at least one analysis hook for each kind of runtime events supported by DynaPyt. The second analysis is the BranchCoverage analysis that traces the control flow entrances. This analysis only instruments the entry points of control flow statements. The third analysis traces all additions performed by a program, i.e., it instruments all binary + operations. We include it as an example of a low-overhead analysis. For each project and setting we report the average running time over five runs.

Figure 4 shows the results, where each bar is the overhead factor relative to an uninstrumented execution. As expected for a general-purpose framework aimed at heavyweight dynamic analysis, the overhead imposed by the TraceAll analysis is relatively high, namely between 1.2× and 16×. In contrast, the two more lightweight analysis, BranchCoverage and OnlyAdd, impose only a moderate overhead, namely between 1.0× and 2.0×. In practice, the overhead of an analysis, of course, depends not only on DynaPyt, but also on what computations an analysis performs when reacting to a particular runtime event.

*Comparison with* `sys.settrace` *and other frameworks.* To put the overhead results in context, we directly compare with the `sys.settrace` function offered by Python. To this end, we implement a simple analysis based on it that traces all executed opcodes and maintains a list for executed code segments. As shown in Figure 4, the analyses built with DynaPyt that track a subset of all runtime events are much faster than `sys.settrace`: between 5.6%-88.6% faster, depending on the analysis hooks of interest and project under analysis. For the *TraceAll* analysis, DynaPyt outperforms `sys.settrace` for some projects, and vice versa for some other projects.

Even though a direct comparison with frameworks for other languages is impossible, we mention some examples for an indirect comparison. For an analysis equivalent to *TraceAll*, the Jalangi framework for JavaScript imposes 26x overhead during record plus 30x overhead during replay [29]. Similarly, the RoadRunner framework for Java byte code imposes an average overhead of 52x without any analysis [10].

## 4 RELATED WORK

*Dynamic Analysis Frameworks.* Several dynamic analysis frameworks exist for languages other than Python. Jalangi [29] for JavaScript and Wasabi [17] for WebAssembly have inspired some of our design decisions, e.g., to perform source-to-source instrumentation and to selectively instrument only those code locations relevant for a specific analysis. Other related frameworks are DynamoRIO [4], Pin [19], and Valgrind [21], which all target x86 binaries. In contrast to them, DynaPyt instruments the source statically instead of at runtime, which avoids the overhead of runtime instrumentation. Two frameworks for Java are DiSL [20], which uses aspect-oriented programming to weave analysis behavior into a program, and RoadRunner [10], which specifically targets concurrency-related dynamic analyses. As also summarized in Table 1, DynaPyt stands out by offering a much larger number of analysis hooks, which we organize into an event hierarchy, and by being the first general-purpose dynamic analysis for Python.

*Built-in System Tracing API.* The closest existing approach for Python is the `sys.settrace` function provided by the CPython implementation of the language. The function allows for registering a hook that gets called at one of three granularity levels: at every executed Python opcode, at every executed line of code, or at every function call and return. A dynamic analysis framework for Python could, in principle, be built upon `sys.settrace`. However, that approach would lack the ability to track a finely selected subset of all runtime events, whereas DynaPyt offers selective instrumentation based on a fine-grained event hierarchy. Another option would be to implement a dynamic analysis directly on top of `sys.settrace`,

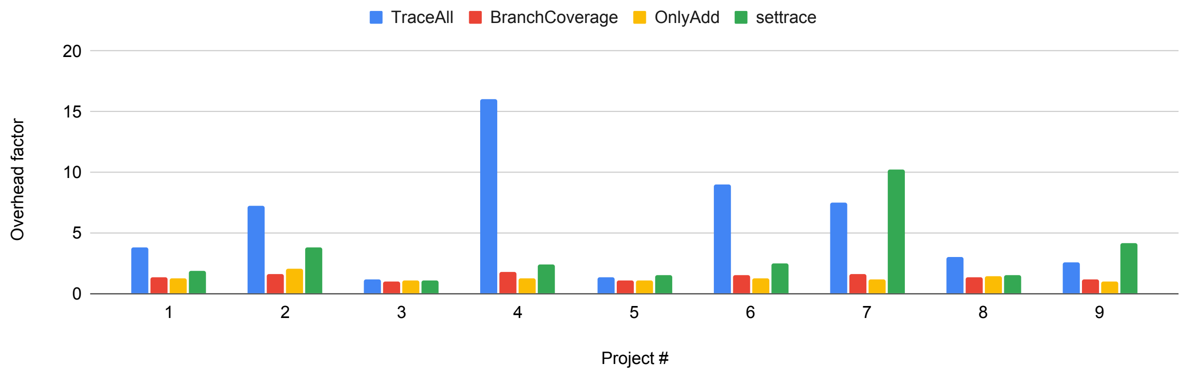

**Figure 4: Overhead of analyses as a factor of the original running time.**

as done, e.g., by Xu et al. [36]. We argue that writing an analysis on top of DynaPyt is easier, as its analysis hooks match the level of abstraction of the programming language, whereas opcode-level and line-level hooks do not. It can also be faster to run an analysis implemented with DynaPyt than with `sys.settrace`, if only a subset of runtime event are of interest. Our experiments show that DynaPyt analyses can be 5.6%-88.6% faster than `sys.settrace` ones.

*Program Analyses for Python.* Given the dynamic nature of Python, there currently are surprisingly few dynamic analyses for the language. Noteworthy analyses include work to detect bugs [36], to enforce differential privacy [1], or to slice programs [6]. Beyond dynamic analysis, there is range of static analyses for Python, e.g., a lightweight analysis to find programming issues in test cases [34], a type inference analysis [37], a constraint-based inconsistency checker [5], a call graph analysis [27], an abstract interpretation-based analysis to find runtime errors [11]. We attribute the relative scarcity of dynamic analyses for Python, at least partially, to the so far limited support for implementing them, which is the problem addressed by DynaPyt.

*Dynamic Analyses for Other Languages.* Also beyond Python, dynamic analysis is an effective means to address various problems related to software correctness, security, and performance, especially for dynamically typed languages, where static analysis tends to be more limited. Some analyses detect type-related problems [2, 25], concurrency bugs [9, 22, 28], and other common bug patterns [14]. Others infer API usage protocols [24, 38] and input grammars [15], or find optimization opportunities [32, 35]. Security-oriented analyses include taint analysis [7], dynamic detectors of similar functions [8], and binary-level security analyses [31]. Since DynaPyt is a general-purpose dynamic analysis framework, we envision it to enable a wide range of novel analyses to understand and improve Python programs.

*Studies of Python Software.* Several studies of code written in Python pinpoint problems that could be addressed through dynamic analysis. A study of bugs in deep learning code [39] reports a need for techniques to find and debug such bugs, e.g., by tracking how incorrect values propagate through a learning pipeline. A study of Jupyter notebooks, which are written in Python, shows that such notebooks often contain code of poor quality [33], and pinpoint a need for analysis tools targeting Jupyter notebooks. Another study reports that security issues are common in small Python code snippets shared between developers [26]. Our work provides a solid basis for addressing these and other issues through appropriate analyses.

## 5 CONCLUSION

This paper presents DynaPyt, the first general-purpose framework for dynamically analyzing Python programs. This framework allows developers to implement dynamic analyses with little effort and at their desired abstraction level. Moreover, it enhances the execution time of the analyses compared to built-in tracing methods.

*Artifact Availability.* DynaPyt is publicly available on GitHub at https://github.com/sola-st/DynaPyt and as a Python package on PyPi[9]. All evaluation scripts are available on the GitHub repository. The projects used as code to analyze are open-source projects available on GitHub.

## ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.

## REFERENCES

[1] Chike Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021.* IEEE, 1–15. https://doi.org/10.1109/CSF51468.2021.00043
[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby.. In *POPL*. 459–472.
[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* (2017).
[4] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.
[5] Lin Chen, Baowen Xu, Tianlin Zhou, and Xiaoyu Zhou. 2009. A Constraint Based Bug Checking Approach for Python. In *Computer Software and Applications Conference (COMPSAC)*. IEEE, 306–311.
[6] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *IEEE 38th Annual Computer*

[9]https://pypi.org/project/dynapyt/

*Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

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

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

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

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

[11] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static value analysis of Python programs by abstract interpretation. In *NASA Formal Methods Symposium*. Springer, 185–202.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223.

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

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

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

[16] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 897–906. https://doi.org/10.1145/3038912.3052674

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

[18] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 190–200.

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

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

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

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

[23] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications.. In *USENIX Security*. 829–844.

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

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

[26] Md. Rayhanur Rahman, Akond Rahman, and Laurie A. Williams. 2019. Share, But be Aware: Security Smells in Python Gists. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 536–540. https://doi.org/10.1109/ICSME.2019.00087

[27] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1646–1657.

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

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

[30] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 263–272.

[31] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International conference on information systems security*. Springer, 1–25.

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

[33] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better code, better sharing: on the need of analyzing jupyter notebooks. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 53–56. https://doi.org/10.1145/3377816.3381724

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

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

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

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

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

[39] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. https://doi.org/10.1145/3213846.3213866