DOI: 10.15514/ISPRAS-2025-37(6)-39

Devirtualization-Based Python Static Analysis

[1] *A.L. Galustov, ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,2] *K.I. Vihlyantsev, ORCID: 0009-0007-4292-0891 <vikhliantsev.ki@phystech.edu>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9, Institutskiy Pereulok, Dolgoprudny, Moscow Oblast, 141701, Russia.*

[3] *Lomonosov Moscow State University, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. In this paper we present an approach to static analysis of Python programs based on a low-level intermediate representation and devirtualization to provide interprocedural and intermodule analysis. This approach can be used to analyze Python programs without type annotations and find complex defects inaccessible to traditional AST-based analysis tools. Using CPython bytecode as a base, the representation suitable to static analysis is constructed and call resolution is performed via an interprocedural devirtualization algorithm. We implemented the proposed approach in a static analyzer for finding errors in C, C++, Java, and Go programs and achieved good results on open-source projects with minimal modifications to existing detectors. The detectors that are relevant to Python had a true positive rate from 60% up to 96%. This demonstrates that our approach allows to apply techniques used for analysis of statically typed languages to Python.

Key words: static analysis; Python; devirtualization.

For citation: Galustov A.L., Vihlyantsev K.I., Borodin A.E., Belevantsev A.A. Devirtualization-based Python static analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 3, 2025, pp. 109-120. DOI: 10.15514/ISPRAS-2025-37(6)-39.

Статический анализ языка Python с использованием девиртуализации

[1] *Галустов А.Л., ORCID: 0009-0001-9591-5873 <artemiy.galustov@ispras.ru>*
[1,2] *Вихлянцев К.И., ORCID: 0009-0007-4292-0891 <vikhliantsev.ki@phystech.edu>*
[1] *Бородин А.Е, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *Белеванцев А.А., ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский Физико-Технический Институт, 141701, Россия, г. Долгопрудный, ул. Первомайская, д. 5.*

[3] *Факультет ВМК МГУ им. М.В. Ломоносова, 119991, Россия, г. Москва, ул. Колмогорова, д. 1, стр. 52.*

Аннотация. В статье предлагается подход к статическому анализу программ на языке Python на основе низкоуровневого внутреннего представления и девиртуализации, который позволяет выполнять межпроцедурный и межмодульный анализ. Подход применим к программам, не содержащим ручных аннотаций типов, и может быть использован для поиска сложных ошибок, которые не ищутся популярными инструментами на основе анализа АСД. Представление для анализа строится по байткоду CPython, затем в результате работы межпроцедурного алгоритма девиртуализации разрешаются вызовы. Предлагаемый подход к девиртуализации уже реализован для языков C, C++, Java, Go и показал хорошие результаты без необходимости изменения существующих детекторов. После адаптации алгоритма для языка Python доля истинных срабатываний детекторов для Python составила от 60% до 96%. Таким образом, изначально предложенный для статически типизированных языков алгоритм оказался применимым к языку Python.

Ключевые слова: статический анализ; язык программирования Python; девиртуализация.

Для цитирования: Галустов А.Л., Вихлянцев К.И., Бородин А.Е., Белеванцев А.А. Статический анализ языка Python с использованием девиртуализации. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 109–120 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–39.

1. Introduction

The Python Programming language is consistently one of if not the most popular of the last decade. Popularity indices like TIOBE [1] place it in first by popularity among programmers in the last few years and it remained firmly in top 10 for almost the past two decades. Despite this and the fact that some studies have shown that Python programs may be more prone to software defects then software in other languages [2] there are relatively few static analysis tools for Python. Those that exist can be divided into two broad categories: utilizing AST-based analysis (tools like Pylint [3] and Bandit [4]) and utilizing type annotations (MyPy [5], Pyre [6]). AST based analysis tools may be more popular and easier to implement but they fail to detect more complex interprocedural and intermodular defects. Tools that utilize type annotations are better suited to find complex defects but are limited by the fact that vast majority of Python programs don't utilize type annotations as indicated by [7] which discovered that less than 4% of open-source projects on GitHub use type annotations in their projects.

In this paper we present our approach to static analysis of Python programs based on low-level intermediate representation using devirtualization algorithm to provide precise analysis. Our approach allows to leverage existing detectors implemented in Svace [8-9] – a static analysis tool aimed at finding complex defects in source code of programs on a variety of languages. Currently Svace supports C, C++, Java, Kotlin, Scala, and Go. All of these are statically typed compiled languages.

2. Building IR

To work with a wide variety of different languages Svace relies on a low-level intermediate representation Svace IR. This representation is represented as a Control Flow Graph (CFG) where instructions are in Static Single Assignment form (SSA) and branch conditions are denoted as assume instructions in target blocks. List of instructions relevant to Python analysis is presented in Table 1.

Python source files must be compiled into a suitable format that can then be transformed into Svace IR. *CPython bytecode* is well-suited for this purpose due to its similarity in structure and semantics to Svace IR. The main problem with this representation is lack of stability. For example, Python 3.11 introduced a new way of handling exceptions incompatible with the old one. Because of this we restrict Python version to *3.12* and focus on specifics of this version in this paper.

To obtain the bytecode, we utilize the standard Python library module **dis** [10], which provides a straightforward way to read and manipulate the bytecode. Moreover, by incorporating specific additional instructions into Svace IR, we can achieve full compliance with Python's semantics, further solidifying its utility for our analysis tasks.

Svace IR is constructed in two phases. First source directories are traversed recursively and for each Python source file the bytecode is saved to a file without any significant modification. Additionally directory structure that contains files found is also saved to facilitate import resolution (see section 3.5). Second when analysis starts bytecode is read from these files and transformed to Svace IR. This is where most of transformations outlined below take place.

Table 1. Main Svace IR instructions.

`a = alloca()`	memory allocation
`a = b`	SSA-assignment
`a = *b`	pointer dereference
`*a = b`	pointer assignment
`a = b.field`	read object attribute
`a.field = b`	write attribute
`a = makeclosure func, (`c_0`, `c_1`, …)`	closure/lambda creation
`a = ptr (`a_0`, `a_1`, …)`	virtual function call
`assume condition`	condition true on the execution path

2.1 SSA Form

Svace operates on an intermediate representation in Static Single Assignment (SSA) form, whereas Python bytecode does not conform to this format. Consequently, a translation process is necessary to reconcile this discrepancy. To achieve this, we employ a two-stage approach. Initially, we perform analysis of the string table within the bytecode, focusing on identifying variables that appear on the left-hand side of assignment operations with multiplicity greater than one. These variables are modeled via references to memory allocated via **alloca** instructions at the start of function. Single-instance variables are directly represented via variables in Svace IR. The outcome of this pass is a symbol table that serves as the foundation for constructing the intermediate representation. Subsequently, we execute a bytecode-to-Svace IR conversion process, where each instruction is individually modeled to accommodate its SSA-form requirements, effectively transforming the original bytecode into Svace IR. An example of source code, its Python bytecode and resulting Svace IR can be seen in fig. 1.

Additional consideration is global and non-local (captured) variable handling. At the level of Python bytecode all variables accessed using **STORE_NAME** and **LOAD_NAME** in module functions are

considered global and in normal functions special **LOAD_GLOBAL** and **STORE_GLOBAL** are used. Unlike Python bytecode Svace IR does not have special instruction for global variables handling and instead has special type of global symbols which are equal when used in different functions. Therefore, when converting to SSA form a special context is maintained per Python source file which allows tracking of all global variables across different functions. Non-local variables behave differently. Unlike globals, they are not just available to all inner functions. Instead, each **MAKE_FUNCTION** accepts a tuple argument with all captured variables. This information is saved in the **makeclosure** Svace IR instruction and is later used by devirtualization and main analysis.

Python source	Python bytecode	Svace IR

`def foo(n):`	`RESUME`	`n = alloca ()`
		`*n = arg_n`
		`s = alloca()`
		`return_value = alloca()`
		`*return_value = None`

`if n < 0:`	`LOAD_FAST (n)`
	`LOAD_CONST (0)`
	`COMPARE_OP (>)`
	`POP_JUMP_IF_FALSE (to 16)`
		`n_1 = *n`

`s = n`	`LOAD_FAST (n)`
	`STORE_FAST(s)`
		`assume n_1 > 0`
		`n_2 = *n`
		`*s = n_2`

`s += 2`	`LOAD_FAST_CHECK (s)`
	`LOAD_CONST (2)`
	`BINARY_OP (+=)`
	`STORE_FAST (s)`
		`s_1 = *s`
		`s_2 = s_1 + 2`
		
		`*s = s_2`

`return s`	`LOAD_FAST (s)` `RETURN_VALUE`	`s_3 = *s` `*return_value = s_3`

Fig. 1. Python source code translation example.

2.2 Exception handling

Exception handling in Python is done using an auxiliary structure called *exception table*. This structure stores the start and end offsets of the instruction blocks that can throw an exception and corresponding offset of block that handles the exception. During evaluation if an exception occurs and exception tables has a matching range, the code execution goes to the handler. However, this approach has several problems when converting to Svace IR: this creates implicit control flow not represented by any instructions and this potentially means that each instruction inside block defined in exception table can throw an exception. Solution to first problem is creating an explicit split in control flow graph (CFG) after each instruction and adding **assume** instructions in each execution path that denote whether this is normal or exception path (fig. 2) [11].

This approach is very convenient for static analysis but exacerbates the second problem, adding a split in CFG after each instruction in try block leads to extremely complex graph which will slow analysis down significantly. At the same time while many instructions in Python bytecode may throw exception this is not modelled the same way in Svace IR. Instead, we opted to assume that only call instructions may throw an exception (fig. 3).

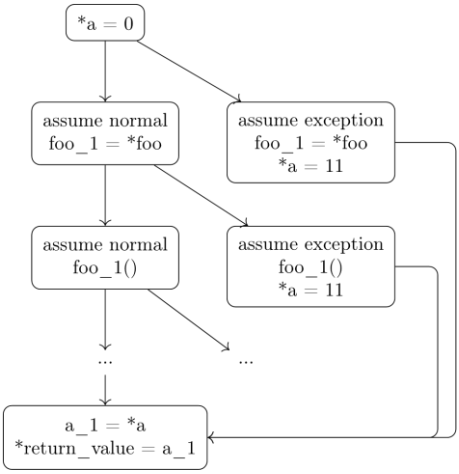

Fig. 2. CFG for try-except.

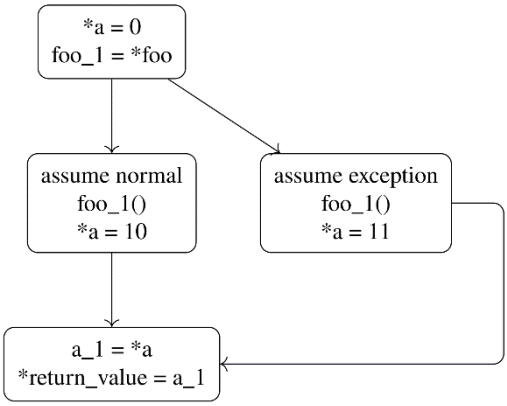

Fig. 3. Simplified CFG for try-except.

2.3 Classes and object instantiation

Classes in Python are special functions which when called produce an instance of the class [12, page 3.2.8.8]. Additionally behavior of instance creation can be modified with special methods like **__init__** and **__new__**. As specified in Python reference when class object is called first instance of class is created, possibly via the **__new__** method and then **__init__** method is invoked in instance. However, the **__call__** method of class object is not contained anywhere in CPython bytecode. Consider the following example:

```
class Example:
    a = 1
    def __init__ (self, b, c):
        self.b = a
        self.c = b
    def foo (self):
        print (self.a)
```

Bytecode generated for this example will contain a **<class Example>** block that initializes methods and fields, and functions **__init__** and **foo**. However, the factory function that corresponds to the

Example itself is not present (note that it is not the same as **<class Example>**, it does not invoke **__new__** or **__init__**). Actually, that function is generated using a special **__build_class__** built-in that is written in C[1] that accepts **<class Example>** function as input and produces class object. The **__call__** method of class is also a built-in function written in C (known as **type_call**) that is common for all classes (unless custom metaclass is used) [12, page 3.3.3.1].

This behavior is quite complex and will not work well with algorithms insensitive to caller context such as devirtualization algorithm used in Svace (see Section 3). Instead, our analysis models this by adding the missing **type_call** function for each class to Svace IR during build process and reproduce required behavior in it:

```
def Example (a, b, c):
        class = < class Example > ()
        instance = class.__new__ ()
        instance.__init__ (b, c)
        return instance
```

Adding such a function for each class (not just a common one as part of metaclass implementation) improves analysis accuracy by removing the need for flow and context sensitivity to analyze class creations.

3. Analyzing calls and imports

As previously mentioned, our solution does not construct any call graph while building IR. Other tools also use algorithms separate from construction of internal representation (be that some low level SSA form or AST) to perform two tasks: import resolution and qualified name resolution. This algorithms by themselves are quite difficult to implement due to intricacies of Python scoping rules and dynamic nature of language. However, they are only effective at working with calls to "basic" function not giving any insight into method calls or lambda invocations. Using typing annotation is more useful but as mentioned earlier small amount of Python programs use them.

We argue that more suitable approach is to use a virtual call resolution algorithm, i.e., devirtualizing for all functions. Svace already has a robust and extensible algorithm for multilingual devirtualization [13]. However, Python differs significantly from any languages that Svace already supported in two key ways: dynamic typing and no static linking. This warrants some additions to devirtualization algorithm.

Devirtualization algorithm in Svace is performed in two parts [13]:

1. Analysis on individual functions building *summaries*. Summaries are language agnostic representation of type aliases in functions and dependencies or updates to values external to the function. For example, assigning or reading global variables, using function arguments or calling other functions etc. When a summary is constructed each variable in a function is assigned to multiple alias classes of two possible types: direct and transitive. A direct alias represents that a variable contains a certain function, an instance of certain class etc., while a transitive alias represents that variable receives data from some external function.
2. Iterative algorithm that manipulates a dependency graph of function summaries to resolve global interprocedural dataflow. Summaries are processed to see which global values are updated by this summary and then algorithm proceeds to update function summaries affected and so on until algorithm eventually converges.

To add support for Python in this algorithm Python specific constructions need to be converted to this language-agnostic representation.

[1] This built-in function is so special that it even has a dedicated **LOAD_BUILD_CLASS** opcode.

3.1 Functions and lambdas

Python is often described as a functional programming language [14]. This is often attributed to first-class function support like lambdas. However, at Python bytecode level all functions, not only lambdas but also ones defined with **def** keywords are treated as values. This is in stark contrast to languages like C/C++, Go and others where distinction between regular calls and calls to function pointers, virtual methods etc. In fig. 4 CPython bytecode and Svace IR for simple function call is presented.

Python source	Python bytecode	Svace IR

`def foo:`
`  ...`
	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_FAST (foo)`
		`mkcls_res = makeclosure foo`
		`*foo = mkcls_res`

`foo()`	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_FAST (foo)`
		`foo_1 = *foo`
		
		`foo_1()`

Fig. 4. Simple function call IR.

MAKE_FUNCTION instruction produces an object corresponding to the function which is stored in variable **foo**. Any subsequent calls to this function will retrieve value from variable to stack and **CALL** uses this object to invoke function. This is the only way to create and invoke functions in Python[2]. This means that devirtualization in Python has one source of direct aliases **makeclosure** Svace IR instruction and one type of call virtual call of functional object, represented as pointer call Svace IR instruction. When **makeclosure** instruction is encountered result is assigned an alias of respective function. While this seemingly simplifies IR over languages like C/C++, Go and JVM-based languages where multiple types of calls can occur [13] the fact that even the simplest cases like the one above require dataflow analysis to resolve increases complexity of devirtualization for Python.

3.2 Captured variables

As mentioned above Python has a special way of passing data to a function in place where it is created via captured variables. This data is associated with created function object in **makeclosure** instruction. During execution the data stored in object is passed to function only at call site. However, in context- and flow-insensitive devirtualization algorithm this data can be passed directly to the **makeclosure** instruction. Just like an external dependency is established for arguments of **pcall** an external dependency is established for each captured variable passed to the created function. This removes the need to track captured variables along with the created function when modelling dataflow, but provides correct results due to context-insensitivity of analysis.

3.3 Attributes

Modelling composite objects in dataflow analysis can be done in many different ways. The most precise approach is to model both individual objects and their individual components. This is done during the main analysis in Svace but this is quite a heavy approach that does not scale well to global dataflow analysis algorithms. To simplify one can either treat all components of object as one (this is useful for modelling arrays, treating all elements of array as one) or treat all instances as the same object (this is useful for modelling classes, structures etc.). Devirtualization algorithm treats all fields of the same type as the same location. This works well in languages like C/C++, Java and Go where

[2] As mentioned earlier Python bytecode utilizes two instructions to denote calls: **CALL** and **CALL_KW**. In Svace both are translated to pointer call instruction.

type of object field of which is accessed is known due to static typing. In Python type is not known and needs to be inferred. Consider the following example:

```
class Data:
    pass
def assign (obj):
    obj.a = "string"
def read (obj):
    a = obj.a
    print (a.islower ())
obj = Data ()
assign (obj)
read (obj)
```

Here the algorithm first infers that **obj** argument of both **assign** and **read** functions is an alias to class **Data**. Then during analysis of **assign** function an update to all readers of field **a** of class **Data** is issued notifying them that field now aliases class **str** and finally **read** function is analyzed again and can resolve the call to **islower** method as type of **obj.a** is resolved. This also seamlessly handles that Python attributes (unlike fields in statically typed languages) can be assigned or read without any prior declaration. Note that if calls to **assign** and **read** receive objects of different types:

```
assign (Data())
read (OtherData())
```

then call to **a. islower ()** will not be resolved due to types of **obj** being different, as expected.

3.4 Classes

As mentioned earlier in Section 2.3 classes are initialized by a complex sequence of function invocations and more specifically methods of classes are initialized in a special **<class>** function by assigning function to respective attributes of class. The fact that this function is unique per class declared in program makes it a good identifier for class type alias used in devirtualization. Thus concrete aliases to class type reference respective **<class>** function and return values of special constructor functions generated for each class are assigned this alias. Any variable assigned in **<class>** function (using **STORE_NAME** instruction) are automatically assigned to respective attribute as illustrated in fig. 5.

Python source	Python bytecode	Svace IR
`class Example`	`RESUME`	
`a = 1`	`LOAD_CONST (1)` `STORE_NAME (a)`	`clazz.a = 1`

`def __init__():`
`  ...`
	`LOAD_CONST (<__init__>)`
	`MAKE_FUNCTION`
	`STORE_NAME (__init__)`
		`mkcls_res_1 =`
		`makeclosure __init__`
		`clazz.__init__ =`
		`mkcls_res_1`

`def foo(self)`	`LOAD_CONST (<foo>)`
	`MAKE_FUNCTION`
	`STORE_NAME (foo)`
		`mkcls_res_2 =`
		`makeclosure foo`
		`clazz.foo =`
		`mkcls_res_2`

Fig. 5. Class declaration translation example.

Consequently, method call **c. foo(args)** in Svace IR is represented as two instructions: **c_foo = c. foo** and **c_foo(args)**[3]. This means that previously described approach to modelling attributes models Python method semantics perfectly without the need to additionally model method tables like in C++, Java and Go where methods are a distinct entity [13]. This approach is also able to model some Python specific tricks like overwriting methods after object creation (a.k.a "monkey patching"):

```
class Example:
    def foo (self):
        pass

o = Example ()
o.foo = lambda: print ("Hello, world!")
```

In this example there are two assignments to **foo** attribute of **Example** class. One is a method that does nothing and one is a lambda that prints "Hello, world!". This is a valid Python code and is sometimes used in real Python project as an alternative to overriding methods. Our devirtualization algorithm can handle this situation correctly because it is no different semantically than method assignment in class creation process.

3.5 Module imports

The final piece to full Python analysis is intermodularity. In the most basic case the **import** statement receives a single argument, which is a path to the module, and binds it to the current function as a variable [12, page 5]. Imported modules in Python are also regular objects with attributes. Each module gets its own type based on **<module>** function for direct aliases encoding type information of corresponding object (just as each class does based on its **<class>** function). Each assignment to variable in module function is also modeled as an assignment to attribute of module with the same name. Then when encountering import instructions result must get an alias to corresponding module. As mentioned earlier information about directory structure is saved during the build phase. This information is stored as a filesystem tree and when an import instruction is processed, this tree is traversed according to rules outlined in Python documentation [12, page 5.5] and respective module is selected. Then the result of import call is assigned alias to appropriate module. More complex forms of imports such as **import a from b as c** then can be represented as a simple import and number of attribute reads for "import from" statements and assignments for "as" directives.

Similarly to class modelling this makes devirtualization able to deal with "monkey patching" and also other unusual applications of modules when they are treated as regular objects. For example, modules that are passed as function arguments:

```
def foo (obj):
    return obj.urlopen ("https://example.com")

def bar():
    import urllib.request
    return foo (urllib.request)
```

Here **urllib.request** argument is modeled as an alias to **<module request>** object and in **foo** function attribute **urlopen** is resolved to correct function.

[3] This corresponds to CPython opcodes **LOAD_ATTR** and **CALL**. Interestingly before Python 3.12 special **LOAD_METHOD** and **CALL_METHOD** opcodes that combined the two existed.

4. Results

To test the approach, we have analyzed a set of open source projects of various sizes ranging from tens of thousands to one and a half million lines of code. Projects, their versions and source code sizes are outlined in Table 2.

Table 2. Analyzed projects.

Project	Version	Size (KLOC)
home-assistant/core	2023.9.1	1537
pytorch	2.5.1	1304
tensorflow	2.18.0	926
plotly.py	5.14.1	747
cpython	3.11.5	709
django	4.2.5	372
jax	0.4.14	182
matplotlib	3.8.0	180
fastapi	0.101.1	83
ipython	8.12.2	54

To perform analysis a server with 16 CPU cores and 64 Gb of RAM was used. Warnings for a set of detectors were reviewed and results are presented in Table 3. Detectors chosen are the ones that do not depend on particular details of analyzed language and thus could be used for Python without any major modifications. Additionally, Table 4 contains devirtualization algorithm performance statistics for various projects.

Large number of emitted warnings and high rate of true positives demonstrates that approach to analysis of Python presented in this paper can effectively transform Python programs in a form suitable for complex static analysis using the same tools that are applied to statically typed languages like C, C++, Java etc. that are handled well by Svace.

Table 3. Results summary.

Warning	Total warnings	TP	TP%
DEREF_AFTER_NULL	47	23	82%
DEREF_OF_NULL	94	38	64%
DIVISION_BY_ZERO	235	133	78%
NULL_AFTER_DEREF	115	47	64%
REDUNDANT_COMPARISON	358	147	75%
TAINTED_PTR	110	91	96%
UNUSED_FUNC_RES	456	224	93%
UNUSED_VALUE	448	190	81%

5. Conclusion

This paper describes an approach to static analysis of Python programs. Using specific techniques during intermediate representation construction and applying a devirtualization algorithm allows to create representation similar to those generated for statically typed languages and is a critical

component for interprocedural and intermodular analysis. This representation can be efficiently used by existing static analysis tools such as Svace to detect many types of errors in programs without any additional modifications to existing detectors.

Table 4. Devirtualization statistics.

Project	Total time (sec)	Devirt time (sec)	Devirt time %	Virtual call number	Resolved	Resolved %
home-assistant/core	504	13.0	2.5%	805260	53330	6.6%
pytorch	644	15.9	2.4%	705661	46484	6.6%
tensorflow	370	9.1	2.5%	601024	68880	11.4%
plotly	404	6.9	1.7%	124635	14808	11.8%
cpython	626	11.5	1.8%	449269	24903	5.5%
django	282	7.1	2.5%	220678	15955	7.2%
jax	215	6.1	2.8%	120423	7717	6.4%
matplotlib	157	3.5	2.2%	82679	12503	15.1%
fastapi	68	1.3	1.9%	19359	1634	8.4%
ipython	90	1.4	1.5%	21897	1484	6.7%
scrapy	34	0.8	2.3%	27120	1232	4.5%

References

[1]. Tiobe index. Available at: https://www.tiobe.com/tiobe-index/ (accessed 23.02.2025).

[2]. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pp. 155-165, 2014.

[3]. Pylint documentation. Available at: https://pylint.readthedocs.io/en/latest/ (accessed 03.04.2025).

[4]. Bandit documentation. Available at: https://bandit.readthedocs.io/en/latest/ (accessed 03.04.2025).

[5]. Mypy - Optional Static Typing for Python. Available at: https://mypy-lang.org/ (accessed 03.04.2025).

[6]. Quickstart | Pyre – pyre-check.org. Available at: https://pyre-check.org/docs/pysa-quickstart/ (accessed 03.04.2025).

[7]. I. Rak-Amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pp. 57-70, 2020.

[8]. Ivannikov V., Belevantsev A., Borodin A., Ignatiev V., Zhurikhin D., and Avetisyan A. Static analyzer svace for finding defects in a source program code. *Programming and Computer Software*, 40(5). pp. 265-275, 2014.

[9]. A. Borodin and I. Dudina. Intraprocedural Analysis Based on Symbolic Execution for Bug Detection. *Programming and Computer Software*, 47(8), pp. 858-865, 2021.

[10]. Dis – disassembler for python bytecode. Available at: https://docs.python.org/3.12/library/dis.html (accessed 03.04.2025).

[11]. Афанасьев В.О., Дворцова В.В., and Бородин А.Е. Статический анализатор для языков с обработкой исключений. Труды Института системного программирования РАН, 34(6):7-28, 2022. / Afanasyev V.O., Dvortsova V.V., Borodin A.E. Static analysis for languages with exception handling. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 7-28 (in Russian). DOI: 10.15514/ISPRAS-2022-34(6)-1.

[12]. The python language reference. Available at: https://docs.python.org/3.12/reference/ (accessed 03.04.2025).

[13]. A. Galustov, A. Borodin, and A. Belevantsev. Devirtualization for static analysis with low level intermediate representation. In *2022 Ivannikov Ispras Open Conference (ISPRAS)*, pp. 18-23. IEEE, 2022.

[14]. G. Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41 of number 1, pp. 1–36. Santa Clara, CA, 2007.

Информация об авторах / Information about authors

Артемий Львович ГАЛУСТОВ – магистр, стажёр-исследователь ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Artemiy Lvovich GALUSTOV – masters graduate, researcher at ISP RAS. Research interests: static analysis for finding errors in source code.

Константин Игоревич ВИХЛЯНЦЕВ – бакалавр Московского физико-технического института (факультет радиотехники и кибернетики). Научные интересы включают статический анализ и профилирование динамических языков программирования.

Konstantin Igorevich VIHLYANTSEV – master’s student at the Moscow Institute of Physics and Technology (Faculty of Radio Engineering and Cybernetics). His research interests include static analysis and profiling of dynamic programming languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, член-корреспондент РАН, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., corresponding Member RAS, leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.