Evaluation of Static Analysis Methods of Python Programs

Gulabovska, Hristina; Porkoláb, Zoltán

Abstract: ***Static analysis is a method for detecting code smells and possible software bugs by examining the source code without executing the program. While we have considerable experiences for programming languages with static type system, especially for C, C++, and Java, languages with dynamic behavior requires different approaches. Python is an important programming language with a dynamic type system, used in many emerging areas, including data science, machine learning, and web applications. In this work we overview static analysis methods currently applied for Python, investigate their advantages and shortages, and highlight the restrictions of current tools and suggest further research directions to tackle these problems. We report our experiences applying static analysis methods on an open source Python software system where we found numerous issues confirmed by the developers. Based on these findings, we suggest refined configuration settings on static analysis tools.***

Index Terms: ***static analysis, symbolic execution, Python***

1. Introduction

PYTHON is one of the most rapidly emerging programming languages [38]. Being flexible and expressive, it is very popular to implement on Machine Learning and Cloud based systems among others. The popularity is partially derived from its dynamic behavior: Python is a dynamically typed programming language, i.e. a variable is just a value binded to a certain (variable) name, the value has a type but not the variable. Namely, one can assign a new value with a possibly different type to an existing variable. This would be a compile time error in statically typed languages such as Pascal, C, or Java, but allowed in Python. At the same time, Python is strongly typed, as operations may fail on an object when the operation is not defined on the actual type of the value held [39]. More dynamic features, such as calling methods dynamically, declaring dynamic attributes (using `getattr`), and others also increase the expressiveness and usability of the language.

Manuscript received ... This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

Hristina Gulabovska is a MSc Computer Science student in the Eötvös Loránd University (ELTE), Budapest, Hungary (e-mail: hristina@gulab.me).

Zoltán Porkoláb is an Associate Professor of the Eötvös Loránd University (ELTE), Budapest, Hungary (e-mail: gsd@elte.hu).

However, such dynamic behavior might be an obstacle when we try to validate the software systems written in Python. For languages with static type system, various static analysis methods exist and are used either as commercial [33], [36] or free, open source tools [6]–[8], [18]. Although there are promising projects for Python (see Section 3), they are significantly less expressive than their counterparts for C, C++ or Java languages.

In the recent years we experienced rapid development of static analysis tools and methods. Besides proprietary tools we found a fine number of open source projects with growing developers' communities [16], [43]. Static analysis is an important aspect in modern software development, addressed by various academic and industrial researches, and projects such as Intellectual outputs No. O1 and O2 of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No.2017-1-SK01-KA203-035402: "Focusing Education on Composability, Comprehensibility and Correctness of Working Software" [37], [40].

In this paper, we investigate the possible research directions towards more powerful static analysis tools for the Python programming language. In Section 2 we evaluate the applicable analysis methods based on their strength and weaknesses. In Section 3 we briefly overview the most important tools and research directions currently available for analyzing Python systems. We use two specific tools to compare the *Abstract Syntax Tree* (AST) based methods and the symbolic/concolic execution in Section 4. In Section 5 we evaluate static analysis on an open source software using the feedback from the developers of the software to confirm true and false positive results. Our paper concludes with Section 6.

2. Static Analysis Methods for Software Systems

To verify the correctness of a software system we can choose between methods. The most common solution is to write test cases, either using white box or black box approach. Although testing is essential for modern software development, it is a costly and slow approach which efficiency greatly depends on the test coverage we achieve. The earlier a bug is detected, the lower the cost of the fix is [5], testing is not ideal in this aspect.

Alternatively, we might turn to analyzer tools that apply various validation methods to find potential or actual misuses in the software. Dynamic analysis tools run the program in a special environment where they can detect incorrect, erroneous execution. However, tools such as *Valgrind* [27], or *Google Address sanitizer* [35] which work in run-time, evaluate the correctness of only those parts of the system which have actually been executed. Such *dynamic analysis methods* therefore require carefully selected input data, and they easily can miss certain corner cases.

In the case of static analysis, we do not run the software. Instead, the input of the static analyzers are the source code, and we apply various methods to find dangerous constructs without running the program. Static analysis is a popular method for finding bugs and code smells [4] as they do not depend on the selection of input data while they can (at least theoretically) provide full coverage of the code.

Most static methods apply heuristics, which means that sometimes they may *underestimate* or *overestimate* the program behavior. In practice, this means static analysis tools sometimes do not report existing issues what is called *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. Therefore, all reports need to be reviewed by a professional who has to decide whether the report stands. However, if the tool produces a large number of false positives, the necessary review process requires large efforts by the developers, meanwhile they also lose their trust in the analyser tool. Therefore, when in doubt, many tools rather choose to drop findings to minimize false alarms.

2.1. Pattern Matching

In this method the source code is first converted to a canonical format, and we match regular expression to every line in the source and reports each match. Although, this method seems to be very simple, its huge advantage is working on non-complete source, even if it cannot compile. Additional advantage is the low level of false positives, as well-written regular expressions have easy to predict results. Early versions of CppCheck [25] used pattern matching to find issues in C and C++ programs.

At the same time this method has several disadvantages too. As regular expressions are context free grammars, we are restricted to find issues based on information in the close locality of the problem. Thus, we could not use type information, name and overload resolution, and cannot follow function calls. As a summary, we can consider pattern matching based approaches as easy entry-level methods [26].

2.2. AST Matchers

To provide the necessary context information to the analysis, we can use the *Abstract Syntax Tree* (AST). AST is a usual internal data structure used by the front-end phase of the compilers [1]. Basically, the AST is a lossless representation of the program, frequently also decorated with type information and connections between declarations and their usage. This representation is suitable for catching errors that the simple pattern matching is unable to detect. Such AST based checks are usually relatively fast. Some rules can even be implemented using a single traversal of the AST. That makes it possible to implement such checks as editor plug-ins. Tools, such as the Clang Tidy [8] uses AST matching for most of its checks.

While the AST matcher method is more powerful than a simple pattern matching, it has some disadvantages too. To build up a valid AST requires a complete, syntactically correct source file. To resolve external module dependences we need some additional information not represented in the source itself, such as include path for C/C++ programs, CLASSPATH for Java or BASE_DIR in Python. That usually means, we have to integrate the static analysis tool into the build system which can be painful. Another shortage of the AST matcher method is that it cannot reason about the possible program states which can be dependent on input values, function parameters.

2.3. Symbolic execution

When executing *abstract interpretation* [11] the tool reasons about the possible values of variables at a certain program point. *Symbolic execution* [19], [20] is a path-sensitive abstract interpretation method. During symbolic execution we interpret the source code, but instead of using the exact (unknown) run-time values of the variables we use symbolic values and gradually build up constraints on their possible values. Memory locations and their connections are represented by a sophisticated hierarchical memory model [42]. A constraint solver can reason about these values and is used to exclude unviable execution paths. Most of the high-end proprietary analysis tools, such as CodeSonar [18], Klocwork [33], and Coverity [36], as well as open source products such as the Clang Static Analyzer [7], and Infer [6] use this method.

Symbolic execution is the most powerful method for static analysis as it makes profit from the program structure, the type system, the data flow information and is able to follow function calls. However, there is a price for this. To represent the internal state of the analysis, the analyzer uses a data structure called the *exploded graph* [31]. This graph is exponential in the number of control branches (conditions) of the program. This could be critical, especially for loops, which are represented as unrolled set of conditions and statements. This factor makes symbolic execution also the most resource expensive (in memory and execution time) method.

2.4. Concolic execution

An interesting mixture of symbolic and concrete execution is called a *concolic* execution [34] and it targets this problem. The main idea is that we use concrete values for execution driven by symbolic execution. We start the execution with an arbitrary input value, we maintain both the symbolic execution state and a storage for the concrete values. Whenever the concrete execution takes a branch, the symbolic execution is directed toward the same branch. Then the constraint solver is used to negate the path conditions thus to choose a new concrete value to cover the other branch. [2].

The advantage of concolic execution is that the operations of the program state can be executed on concrete values, thus it could be implemented in more simple way and using less resources, while the SAT solver still helps to follow all the possible execution paths. Especially for languages such as Python, where the interpreter could evaluate the analyzed program making possible to call unmodeled external methods or using third party modules this approach is seriously extending the power of symbolic execution.

3. Analysis tools for Python

Compiler relies on static analysis to generate its warnings during compilation time. However, its primary task is to translate source code, and not to execute a full scale and costly static analysis. The Python compiler misses catching a number of common bugs and errors, therefore already existing third-party Python static analysis tools are aiming to cover the catches missed by the actual compiler. Among the most common actual Python static analysis tools, the following could be listed as the most reliable: PyLint [22], Pyflakes [28], flake8 [10], Frosted [12], Pycodestyle [32], and Mypy [21]. These tools are open-sourced, and some of them are still explicitly said to be in an experimental stage. They are using the AST method in order to statically evaluate the potential bugs and errors of the source code.

Currently, PyLint is seen as the most popular Python static analysis tool, which is free and capable of not only catching logical errors, but also warns regarding the specific coding standards. In PyLint, there is a possibility to write custom rules, too. There are three types of possible custom rules: Raw checkers (analyzing each module as a raw file stream), Token checkers (using list of tokens representing the source code) and AST checkers. Most of the checkers are working on the Abstract Syntax Tree (AST) which is provided by the astroid [23] library. Adding to the reliability of PyLint, it is worth mentioning that it is trusted by many big companies, such as Google [17], which is mostly relying on PyLint for the static analysis of its Python code-base. There is also a number of popular IDEs and frameworks using PyLint for in-time static analysis of the Python code, some of which are: PyCharm, VSCode, Django with PyLint, Eclipse with PyDev etc.

Beside the present static analysis tools, there are several Python tools (mostly in experimental status) which are related to symbolic execution and SMT (Satisfiability Modulo Theories) such as: PyExZ3 [3], PySym [14], PySMT [24], and mini-mc [41], etc. Most of them are using the Z3-solver [15].

During the research and comparison of the AST and symbolic execution methods for Python static analysis in this paper, two tools were used. PyLint, as the currently most reliable representation of evaluating the AST, and, for the symbolic execution part, Z3-solver and mini-mc symbolic model checker, which help to explore the symbolic evaluation of the source code.

One of the more critical common bugs in Python that was not caught by PyLint during the research, nor the Python compiler itself, was the “Closure bug”. Closure in Python is an important concept that allows the function object to remember the values in enclosing scopes even if they are not present in memory. At the same time, it is prone to bugs which as shown in the code example on Listing 1 is very often hardly caught even during runtime.

```python
def greet(greet_word, name):
    print(greet_word, name)
greeters = list()
names = ["Kiki", "Riki", "Joe"]
for name in names:
    greeters.append(lambda x: greet(x,
    name))
for greeter in greeters:
    greeter("Hi ")
```

Listing 1: The closure bug

We may expect this code to print:

```
Hi Kiki
Hi Riki
Hi Joe
```

But instead it prints:

```
Hi Joe
Hi Joe
Hi Joe
```

The closure bug is one of the trickiest issue without actually causing a run time error. In our earlier researches we found that only PyLint is able to catch this problem, reporting a Warning “Cell variable name defined in loop” which is not necessary a clear message for the developers about the specific error they made.

4. Comparison of AST-based and symbolic execution methods

In this section, we compare the power of the AST-based method to the symbolic execution method. We selected two representative tools for the two methods, run tests with them, and analyzed the results. Our goal is not only to show which methods can report more real errors, *true positives*, but also which are better avoiding to report *false positives* – code snippets that are correct but falsely reported as a suspicious code segment.

Most of the present Python tools use the AST method for static analysis of the Python source code. We have selected PyLint [22] as one of the most widely used and powerful static analysis tool for Python, which is using the AST method based on the asteroid [23] library.

For the symbolic execution method, we have chosen mini-mc, an experimental symbolic execution implementation [41], using the Z3's Python interface. The mini-mc tool is implementing the fork-explore-check idea when evaluating symbolic values. The Python VM tries to convert them into boolean values at all branches to intercept the conversion and replace it with a `fork` statement. In practice, mini-mc processes all reachable program paths, forking new processes to evaluate the false branch. It also detects unreachable conditions where the evaluation stops. We selected mini-mc for its simplicity and demonstrative power.

Analyzing the static analysis methods, we noticed that symbolic execution might be better approach for static analyzing of Python considering its dynamically typed characteristics. Therefore, we composed a few examples to see step by step the symbolic evaluation and then compare if PyLint as an AST based static analyzer or mini-mc as a symbolic execution method could catch better the errors during the analysis, and exclude false positives in unreachable paths.

```
 1 #!/usr/bin/env python3
 2 from mc import *
 3 import os
 4 import time
 5
 6 def func(arg):
 7   if(1==arg):
 8     print("branch11",os.getpid())
 9     x=1
10   else:
11     print("branch12",os.getpid())
12     x=0
13   if(1==arg):
14     print("branch21",os.getpid())
15     y=5/x
16   else:
17     time.sleep(3)
18     print("branch22",os.getpid())
19     y=4/(x+1)
20 arg = BitVec("arg",32)
21 func(arg)
```

Listing 2: Usage example of mini-mc.

Listing 2 is the very first Python code example that we used to run a symbolic model checker, and as it is seen, this program should not report an error since both if-conditions (in line 7 and line 13) could be true at the same time and their bodies could be executed without errors. With this example we mostly demonstrate the execution of the mini-mc symbolic model checker. As it is seen on Listing 3 the program was executed in a quasi-parallel way. At every branch statement the program forks a new process, the process id and the logical assumption is written to the output. (The use of time.sleep(3) on line 17 is to emphasize the non-deterministic evaluation order of the branches). When the engine detects unsatisfied condition, that is also printed as `unreachable`.

```
[7088] assume (arg == 1)
[7090] assume ¬(arg == 1)
[7090] unreachable
[7088] assume (arg == 1)
[7088] assume (arg == 1)
branch11 7088
branch11 7088
branch21 7088
[7090] exit
[7089] assume ¬(arg == 1)
[7089] assume (arg == 1)
[7089] unreachable
branch12 7089
[7089] assume ¬(arg == 1)
[7091] assume ¬(arg == 1)
branch12 7089
branch22 7091
[7091] exit
[7089] exit
[7088] exit
```

Listing 3: Mini-mc result executing Listing 2.

The Python example on Listing 4 should point out the power of symbolic execution over the AST based approaches. The program defines the variable `z` in the true-branch of the first if-condition and uses it in the true-branch of the second if-condition. As the two if-conditions (line 2 and line 5) could not be true at the same time, if `1!=arg` then the first if-condition body would not be executed but the second if-condition body is executed. In this case, however, the program becomes faulty since the variable z was not introduced yet.

```
1 def func(arg):
2   if(1==arg):
3     print("branch11",os.getpid())
4     z=1
5   if(1!=arg):
6     print("branch21",os.getpid())
7     x=z
8 arg = BitVec("arg",32)
9 func(arg)
```

Listing 4: Local variable may be referenced before assignment. PyLint does not report.

PyLint, as an AST based static analyzer, does not execute a full path sensitive analysis, therefore it is unable to recognize that the assignment `x=z` will be executed only in those cases when statement `z=1` is not. In the same time, PyLint detects that the variable `z` is defined in one of the branches of the if-condition in line 2 and supposes that it will be used only in this case in line 7. At the end, conservatively, do not report error to minimize possible false positives.

When the mini-mc was run the *Unbound error* was detected (Listing 5) showing the benefits of symbolic execution over the AST based methods for Python as a dynamic language.

```
[6324] assume (arg == 1)
[6324] assume (arg != 1)
[6324] unreachable
branch11 6324
[6324] assume (arg == 1)
[6326] assume ¬(arg != 1)
branch11 6324
[6326] exit
[6325] assume ¬(arg == 1)
[6327] assume ¬(arg != 1)
[6327] unreachable
[6327] exit
[6325] assume ¬(arg == 1)
[6325] assume (arg != 1)
branch21 6325
Traceback (most recent call last):
  File "./example4.py", line 17, in <module>
    func(arg)
  File "./example4.py", line 14, in func
    x=z
UnboundLocalError: local variable 'z'
     referenced before assignment
[6325] exit
[6324] exit
```

Listing 5: Mini-mc output on the Listing 4 detecting the undefined variable error.

On Listing 6 we changed the code in order to check the behavior of symbolic execution when instead of concrete values, intervals are used in the if conditions.

```python
def func(arg):
  if(1<arg):
    print("branch11",os.getpid())
    z=1
  if(2<arg):
    print("branch21",os.getpid())
    x=z
```

Listing 6: Using intervals in conditions.

In order to enter the second if-condition one has to also enter the first if-condition and the unbounded error should not be reported.

```
[5243] assume (arg > 1)
[5243] assume (arg > 2)
[5243] assume (arg > 1)
[5245] assume ¬(arg > 2)
branch11 5243
branch21 5243
branch11 5243
[5245] exit
[5244] assume ¬(arg > 1)
[5244] assume (arg > 2)
[5244] unreachable
[5244] assume ¬(arg > 1)
[5246] assume ¬(arg > 2)
[5246] exit
[5244] exit
[5243] exit
```

Listing 7: Execution result of Listing 6.

The results on Listing 7 shows that the symbolic execution was working just fine when we used intervals.

There are certain situations, however, when symbolic/concolic execution may cause unreasonable false positives. This is derived from the nature of concolic execution we discussed in Section 2-D. The evaluation is partially driven by the SAT solver, that is the engine to encounter all execution paths, but the concrete execution of the statements inside the branches is using a concrete value chosen by the constraint.

In the example on Listing 8 we execute a function with an unknown argument `arg`. There is a very small possibility, that `arg` is 42, which could cause `ZeroDivisionError`. PyLint static analyzer does not report such an error, as this would be most likely a false positive.

```python
def func(arg):
  if arg == 41:
    print("branch21",os.getpid())
  else:
    print("branch22",os.getpid())
    z = arg − 42
    z = 99 / z
```

Listing 8: Concolic execution example

On Listing 9 it could be seen that the symbolic execution inaccurately reports the possibility of `ZeroDivisionError` for the `else` branch.

```
[15532] assume (arg == 41)
branch21 15532
[15533] assume ¬(arg == 41)
branch22 15533
42
Traceback (most recent call last):
  File "example11.py", line 23, in <module>
    func(arg)
  File "example11.py", line 17, in func
    z = 99 / z
ZeroDivisionError: division by zero
[15533] exit
[15532] exit
```

Listing 9: Symbolic execution with mini-mc produces false positive report on Listing 8.

The reason is, that the concolic execution accidentally takes 42 as the sample value for `arg` when `arg != 41`. This makes the otherwise unlikely situation of dividing by zero unavoidable. Although such unlucky situations may be infrequent in every day development, this false positive could be extremely annoying.

Nevertheless, at the moment concolic execution seems to be the most powerful static analysis method for dynamic languages such as Python, but this example shows that it is far from perfect and that there is room to improve it.

5. Empirical results

For the empirical results on Python static analysis in practice, PyLint was tested on an open source software system CodeChecker. CodeCheker is an analyzer tool, defect database and viewer extension for the Clang Static Analyzer and Clang-Tidy – written mainly in Python 3 as a cooperation between Eötvös Loránd University, Budapest and Ericsson Hungary Ltd. [9]. Two main reasons why CodeChecker was chosen is: first the expectation that it is high quality software as it is used daily by companies like Google, Mozilla and others, and second, the availability of the developers to confirm our findings. Since at this point of the test, we were specially focused to find directly logical errors and where not interested in the warnings or code styling reports and messages, PyLint was run over the CodeChecker source code with only error reports switched-on.

Several real and particularly interesting error reports were gathered as a result, and in this discussion, we will elaborate the reasons behind them, as well as, summarize the reasoning of their informative value. Moreover, we discuss the possibilities of the detected true positive errors being false positives in other cases (and vice versa), in order to notice if switching a specific error report off and further customizing the static analyzer setup might introduce possible improvements to the power of the static analyzer.

Apart from the empirical results being done for testing and research purposes, we are glad to hear that the developers found this immediately useful enough to start fixing the reported bugs. The pull request to fix most of these issues can be found at [13].

5.1. True positives

There were seven particularly interesting and discussion worthy true positive error reports found during the test run. To begin with, the report "Explicit return in `__init__`" with error code E0101 was confirmed as a true positive by the CodeChecker developers. The `__init__` method in Python is a special one, and is called when allocating memory for a new object. It is not used for creating new objects, only allocating and initializing, so it should not return any value. However, in Python very often it happens that the developers `return None` in the `__init__` method. This does not distinctly cause a run-time error, but the return value is meaningless.

The next one, "Class 'traceback' has no 'print stack' " member with error code `E1101`, is a usual runtime error which occurs when an object is accessed for a non-existent member. Although this was confirmed as a true positive error in the CodeChecker code base, it could be possible, in other cases, and especially if reported by an AST-based static analysis tool, that such an error report is false positive. Due to the dynamically typed nature of Python, this report could refer to object members which are created dynamically, and thus misleadingly predict a run-time error. In effect, this error can be seen reported in our case as well, in the false positive reports on CodeChecker with message Instance of 'BuildAction' has no 'original command' member. There was an object called 'BuildAction' in the code base, which members are dynamically set, and this same report there appears as a false positive.

Afterwards, the following three different true positive reports had the same error code. These are worth to analyze separately due to the fact that each of them being initiated by a slightly different reason. In addition, having the same error code for moderately divergent true positive reports, questions the clarity of the information in the error reports. As mentioned before, the format of the description of the error reports, as well as their clarity, is essential, since it can significantly influence the time and cost of evaluating the reports by the developers, which is presently an unavoidable final step in the complete process of the static analysis. One of these three issues report the message `"Undefined variable 'traceback' "`. It is a very typical error and happens simply because developers forget to import a specific module. In a similar way, it suggests that the `sys` module was not imported. However, other than the obvious case that this can happen, plainly by forgetting to import a specific module, this can also happen, for example, when developers actually do not forget to import the module, but they do it in a bad location in the source code. Namely, it occurs that several lines of imports are put in a try-catch block, including the crucial sys module, too. If for some reason, the try block fails in the middle of execution of the number of import statements, the rest of the statements are being left not executed. Thus, the program continues with the sys module not imported. Of course, this will be a run time error. The third reports of these has the error message "Undefined variable 'msg' ", and it is interesting to observe, because if very simple and naïve. Typing mistakes are really common errors in the everyday industry world. Again, the dynamic nature of Python makes such an error complex enough to get unnoticed by the compiler, and simple typing mistake which might seem trivial to us, leads to a real run-time error, by accessing an undefined variable.

Last true positive error under this discussion is the one with error code E0102 reporting "method is already defined in line 320". Redefining functions, classes and methods in Python is allowed since they are also values which can be stored as variable and you can rebind them and pass around as you would do in C++ when using function pointers. Functions in Python are first-class objects and they are defined dynamically, when we are defining a new function with a function name of an already previously defined function, basically that function name is now just bound to the new function object. Nevertheless, this is still very often confusing for the developers, therefore PyLint reports it as a possible logical error. By looking deeper into the nature of this problem, and due to many known programming conventions of writing unique function names, which is not a special exception while programming in Python, too, one can easily predict that this error report would usually appear as a true positive and definitely needs a very good reason to decide to switch it off while customizing the static analyzer.

5.2. False positives

In the six confirmed false positives, the PyLint error with code `E0611` which for example occurred with message: ″`No name 'spawn' in module 'distutils'` ″ is worth to begin the discussion with. Due to the improvements in Python 3.3+ over Python 2.7, this error is currently, by many developers, initially marked as a false positive, since the rule is written for a behavior in Python 2.7 which is now improved in Python 3.3+. In the simplest case, creating a simple empty `__init__.py` file in the directory of the module, solves the problem of this error. The `__init__.py` file is needed to be able to treat directories which contain it, as packages. According to the documentation, this prevents directories with common name, to unintentionally hide called modules that occur later on the module search path. In Python 2.7 these `__init__.py` files are a must to be able to import from packages, thus PyLint positively reports an error here. However, Python 3.3+ introduced namespace packages which do not need an ordinary list for their `__path__` attribute and there is no `parent/__init__.py` file. Since there might be multiple parent directories found during an import search, not necessarily physically located next to each other, Python creates namespace package for the top-level parent package whenever it or its sub packages are imported. A detailed description can be found in [29], [30]. Knowing this, we could notice that this particular error could be a possible true positive only if it is run on a Python 2.7 version, otherwise when Python 3.3+ is used, it would be the best to disable this error by customizing the setup of Pyint static analyzer. Another false positive error is the one with error code `E1101` which also occurred in the true positive findings. As mentioned in the discussion in 5-A True positives, due to the dynamic nature of the Python language, such an error report with message ″`Instance of 'BuildAction' has no 'original command' member`″ can appear as a false positive, since attributes of an object can be set dynamically during run-time. The number of true positive findings on this report is greater than the false positive, therefore we could not simply switch off this error report in order to lower the false positive cases. We experienced that altogether, Pylint reported slightly more true positives than false positives on the test run. This suggests that to find real issues in a well-designed and widely used open source software system, even the current AST based approaches are strong enough to contribute to the quality of the Python software systems.

6. Conclusion

Static analysis methods evaluate software systems without running them, applying various heuristics on the source code to detect possible code vulnerabilities. Current tools provide less support for programming languages with dynamically type system, such as Python. Although there exist some – mainly AST based – tools, their capacity to detect Python-specific errors are yet unsatisfactory. Symbolic execution techniques may open new directions for supporting Python static analysis. The dynamic behavior of the language is better covered by methods, where the change of the program state is emulated. With the help of SAT solvers, we can provide full coverage of the program path (under the resource constraints). We have shown that even the simplest symbolic execution tools can find issues otherwise not detected by AST based tools. Especially concolic execution is one of the most promising method directions.

Symbolic execution-based analyzer tools for Python can use powerful SAT solvers, such as Z3, but currently, they model neither the type of information nor the most important modules of the Python language. This is a serious restriction, and further research should target that area. Nevertheless, the otherwise powerful concolic execution can also cause unwanted false positives. Based on our experiments, we suggest using both an AST based tool and a symbolic execution tool to maximize the true positives and minimize the reported false positives.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[3] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, (MSR-TR-2015-95), January 2015.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[5] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[6] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[7] Clang SA. Clang Static Analyzer, 2019. https://clang-analyzer.llvm.org/.

[8] Clang Tidy. Clang-Tidy, 2019. https://clang.llvm.org/extra/clang-tidy/ (last accessed: 28-02-2019).

[9] CodeChecker. Codechecker home page, 2019. https://github.com/Ericsson/codechecker (last accessed: 28-08-2019).

[10] Ian Cordasco. Flake8, 2016. http://flake8.pycqa.org/en/latest/.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] Timothy Crosley. Frosted, 2014. https://pypi.org/project/frosted/.

[13] Márton Csordás. Fix pylint errors #2448 – pull request on github.com, 2019. https://github.com/Ericsson/codechecker/pull/2448.

[14] Björn I. Dahlgren. Pysym, 2016. https://pythonhosted.org/pysym/.

[15] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] Artem Dergachev. Clang Static Analyzer: A Checker Developer's Guide, 2016. https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 28-08-2019).

[17] Google. Google python style guide, 2018. http://google.github.io/styleguide/pyguide.html.

[18] GrammaTech. CodeSonar, 2019. https://www.grammatech.com/products/\codesonar (last accessed: 28-02-2019).

[19] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, September 2005.

[20] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[21] Jukka Lehtosalo. Mypy, 2016. https://mypy.readthedocs.io/en/latest/.

[22] Logilab. Pylint, 2003. http://pylint.pycqa.org/en/latest/.

[23] Logilab. Astroid, 2019. https://astroid.readthedocs.io/en/latest/.

[24] Yuri Malheiros. pysmt, 2016. https://pysmt.readthedocs.io/en/latest/tutorials.html.

[25] Daniel Marjamäki. CppCheck: a tool for static C/C++ code analysis, 2013.

[26] Martin Moene. Search with cppcheck. *Overload Journal*, 120, 2014.

[27] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[28] PyCQA. Pyflakes, 2014. https://pypi.org/project/pyflakes/.

[29] Python Software Foundation. Python 2 packages documentation, 2019. https://docs.python.org/2/tutorial/modules.html#packages.

[30] Python Software Foundation. Python 3 packages documentation, 2019. https://docs.python.org/3/tutorial/modules.html#packages.

[31] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[32] Johann Rocholl. Pycodestyle, 2006. http://pycodestyle.pycqa.org/en/latest/.

[33] Roguewave. Klocwork, 2019. https://www.roguewave.com/products-services/klocwork (last accessed: 28-02-2019).

[34] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[36] Synopsys. Coverity, 2019. https://scan.coverity.com/ (last accessed: 28-02-2019).

[37] Csaba Szabó. Programme of the winter school of project no.2017-1-sk01-ka203-035402: "focusing education on composability, comprehensibility and correctness of working software", 2018. accessed 02-July-2019.

[38] Tiobe. TIOBE programming community index, july 2019, 2019. accessed 02-July-2019.

[39] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[40] Š Korečko. Interactive approach to coloured petri nets teaching. Technical Report IK-TR3, Eötvös Loránd University, Faculty of Informatics, Budapest, May 2018.

[41] Xi Wang. A mini symbolic execution engine, 2015. http://kqueue.org/blog/2015/05/26/mini-mc/.

[42] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of C programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] Anna Zaks and Jordan Rose. Building a checker in 24 hours, 2012. https://www.youtube.com/watch?v=kdxlsP5QVPw.

Hristina Gulabovska is a MSc Computer Science student in the last semester of her studies at the Eötvös Loránd University (ELTE), Budapest, specializing in Software and Service Architectures.

Zoltán Porkoláb received his doctoral degree in Computer Science from the Eötvös Loránd University (ELTE), Budapest in 2004. He is an Associate Professor of the Department of Programming Languages and Compilers at the Faculty of Informatics, ELTE, Budapest, Hungary. At the same time, he holds Principal C++ Developer position at Ericsson Hungary Ltd.