Survey on Static Analysis Tools of Python Programs

HRISTINA GULABOVSKA and ZOLTÁN PORKOLÁB, Eötvös Loránd University

Static program analysis is a popular software technique performed by automated tools for verifying large scale software systems. It works without executing the program, just analyzing the source code and applying various heuristics to find possible errors, code smells and style discrepancies. Programming languages with strong static type system, like C, C++, Java are the usual targets of static analysis, as their type system provides additional information for the analyzer. However, there is a growing demand for applying static analysis for dynamically typed languages, like Python. In this paper we overview the current methods and tools available for static analysis on Python code base and describe some new research directions.

1. INTRODUCTION

Maintenance costs take a larger part of the estimated price of the software systems. Most of these expenses are spent for bug fixing. The earlier a bug is detected, the lower the cost of the fix [Boehm and Basili 2001], therefore various efforts are applied to speed up the *development–bug detection–bug fixing* cycle. The classical test based approach do not fit well to this pattern. Writing meaningful tests requires large code coverage and takes substantial development workload and certain time. Another approach, using tools, like *Valgrind* [Nethercote and Seward 2007], or *Google Address sanitizer* [Serebryany et al. 2012] which work run-time, evaluates the correctness only those parts of the system which have been actually executed. Such *dynamic analysis methods* therefore require carefully selected input data, and they easily can miss certain corner cases.

Contrary to testing and dynamic analysis methods *static analysis* works at compile time, based only on the source code of the system, and does not require any input data. It is a popular method for finding bugs and code smells [Bessey et al. 2010]. Compiler warnings are almost always based on various static analysis methods. As we will see in Section 2 many of the applied techniques are fast enough to be integrated into the Continuous Integration (CI) loop, therefore they have a positive impact on speed up the development–bug detection–bug fixing cycle. This makes static analysis a useful and cheap supplement to testing, built into the continuous integration loop during development.

The type system provides extra information for the compiler as well as the static analysis tools. As a consequence, static analysis is most popular for programming languages with static type system, like C, C++ and Java. A number of commercial [Synopsys 2019; Roguewave 2019] and open source tools [Clang SA 2019; Clang Tidy 2019; GrammaTech 2019; Calcagno and Distefano 2011] exist with considerable large developers' community to support these languages [Dergachev 2016; Zaks and Rose 2012]. However Python, one of the most rapidly emerging programming languages with dynamic type

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).
Author's addresses: Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1177, hristina@gulab.me, gsd@elte.hu

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

system is still not sufficiently supported by static analysis tools. Python is very attractive implementing Machine Learning and Cloud based applications among others, and in July 2019 it is already the third most popular language by the Tiobe index [Tiobe 2019].

In this paper we investigate how static analysis methods could be adapted for Python based software systems. The authors believe that in spite of the difficulties caused by the dynamic type system many of the methods applied for the classical languages could be utilized for Python too.

This paper is organized as follows. We discuss the most important analysis techniques in Section 2. We take a survey on static analysis tools used for the Python programming language in Section 3. In Section 4 we evaluate the tools according some typical use cases. This paper concludes in Section 5.

2. METHODS OF STATIC ANALYSIS

In this section we will overview the most frequently used methods of static analysis for modern programming languages. It is not our goal here to go to very deep technical details, but we intend to compare the advantages and the limitations of each methods. We will cover only methods which are (1) implemented at least for one programming language (2) based only on the source code, i.e. we will not discuss methods, where the user has to supply invariants or semantics of the program.

All 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 which situation is called as *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.

Here we face an interesting human challenge. Unlike the theoretical program verification methods, we do not strive to minimize the false negatives, i.e. try to find all possible issues, but we aspire to *minimize the false positives*. If the tool produces a large number of false positives, the necessary review process takes unreasonable large effort by the developers, meanwhile they also lose their trust in the analyser tool.

2.1 Pattern matching

In this method the source code is first converted to a canonical format, i.e. all branches are converted to simple one liner `if-else` format, loops to a simple `while-command` format, comparison expressions are reordered to `less-then` and `less-equal` format. The developers express the various issues to detect as regular expressions. The tool tries to match this regular expression to every line in the source and reports each matches.

Earlier versions of CppCheck [Marjamäki 2013] used pattern matching to find issues in C and C++ programs. Experiences with the tool report relatively low level of false positives, as well-written regular expressions have easy to predict results. Apart of the low false positive rate, additional advantage of pattern matching is working on non-complete source, even if we cannot recompile the software system. This means that the method can be applied to software under construction, or when we have only partial information on the system.

In 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. Having no knowledge about declarations we could not use type information. We cannot use name and overload resolution, cannot follow function calls and generally we have weak understanding on the overall program structure. As a summary, we can consider pattern matching based approaches as easy entry-level methods [Moene 2014].

2.2 AST matchers

To provide the necessary context information to the analysis we can use the *Abstract Syntax Tree* (AST). AST is the usual internal data structure used by the front-end phase of the compilers [Aho et al. 1986]. Basically, the AST is a lossless representation of the program; it encodes the structure of the program, the declarations, the variable usages, function calls, etc. Frequently, the AST is also decorated with type information and contains connections between declarations (types, functions, variables) and their usage.

This representation is suitable for catching errors that the simple pattern matching is unable to detect. These issues include heap allocation using the wrong size, some erroneous implicit conversions, inconsistent design rules, like hurting the rule of three/five in C++ and it proved to be strong enough to detect even some misuses of the STL API.

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. External tools, like the Clang Tidy [Clang Tidy 2019] uses AST matching for most of its checks.

While the AST matcher method is more powerful than the 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, like 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, that it cannot reason about the possible program states which can be dependant from input values, function parameters.

2.3 Symbolic execution

Executing *abstract interpretation* [Cousot and Cousot 1977] the tool reasons about the possible values of variables at a certain program point. *Symbolic execution* [Hampapuram et al. 2005; King 1976] 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 (e.g. a pointer pointing to some variable, structs and their fields) are represented by a sophisticated hierarchical memory model [Xu et al. 2010]. A constraint solver can reason about these values and is used to exclude unviable execution paths. Most of the high end proprietary analysis tools, like CodeSonar [GrammaTech 2019], Klocwork [Roguewave 2019], and Coverity [Synopsys 2019] as well as open source products like the Clang Static Analyzer [Clang SA 2019], and Infer [Calcagno and Distefano 2011] 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* [Reps et al. 1995]. Each vertex of this graph is a (symbolic state, program point) pair. The edges of the graph are transitions between vertices. 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.

3. THE CURRENT TOOL LANDSCAPE OF PYTHON STATIC ANALYSIS

Here we give a brief overview of the current most commonly used static analysis tools in Python.

3.1 Pylint

PyLint [Logilab 2003] is so far the most popular Python static analysis tool which is free and capable of not only catching logical errors, but also warns regarding specific coding standards, offers details about code complexity and suggests for simple refactoring. It is working by building an abstract syntax tree with the help of the Astroid [Logilab 2019]. Also, Pylint allows developers to write own plugins for specific checks which enables them easily to extend the tool. It is used by a several popular IDEs and frameworks mainly for in-time static analysis of the code, some of which: PyCharm, VSCode, Django, Eclipse with PyDev etc. PyLint popularity and reliability is shown through its use at Google as one of the leader tech companies. For the static analysis of its Python codebase, Google is mostly relying on PyLint, while having a common decision how to avoid some of its imperfections. [Google 2018].

3.2 Pyflakes

Pyflakes [PyCQA 2014] focuses only on the logical errors and does not complain about code style or standards. It works by parsing the source file, instead of importing and it is a bit faster than PyLint, since it examines the syntax tree of each file individually. As a consequence, it is more limited in the types of issues that it can check. Its main focus is not to emit false positives which has its advantages, but as a disadvantage of this strict focus, some typical errors are not reported at all.

3.3 flake8

Flake8 [Cordasco 2016] is a wrapper around Pyflakes, pycodestyle and mccabe [Cordasco 2017]. It is mostly used by those who like Pyflakes but also want to get stylistic checks. The code style is checked against PEP8 .

3.4 Frosted

Frosted [Crosley 2014] is a fork of pyflakes which focuses on more open contribution development from the outside public.

3.5 Pycodestyle

Pycodestyle [Rocholl 2006] strictly checks only the code style against the style conventions in PEP8. It is usually used in combination with the other tools which are looking for logical errors and warnings.

3.6 Mypy

Even though it is still considered experimental and optional static checker, Mypy [Lehtosalo 2016] proves itself to be highly effective, especially in type checking. It focuses on combining the benefits of dynamic typing and static typing. It has a powerful type system that in some cases, as discussed later in this paper, it catches type errors that are not caught by any of the common Python static analysis tools.

3.7 PySym

PySym [Dahlgren 2016] is an experimental Python package capable of a symbolic manipulation on a minimal scope. None of the previously mentioned tools are in contact with this package, nor with symbolic execution in general as a method of static analysis. Therefore, it is decided to be considered in this paper in the further investigation of symbolic execution in Python static analysis tools.

3.8 PyExZ3

PyExZ3 [Ball and Daniel 2015] is a tool which as part of the research on “Deconstructing Dynamic Symbolic Execution” [Ball and Daniel 2015], adds to the exploration of the dynamic symbolic execution (DSE) through a minimalist implementation of DSE for Python. With symbolic execution not being a common topic in the present static analysis field of Python, the authors of this paper believe they can rely on the PyExZ3 tool for further experimentation on dynamic symbolic execution in Python with the goal to improve the Python static analysis tool set.

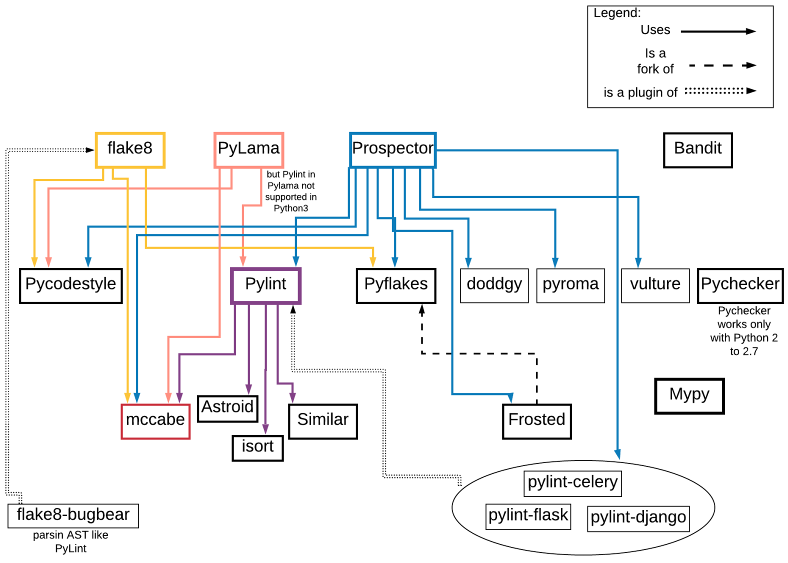

Fig. 1: Python static analysis tool relations.

On Figure 1 we summarize the connections between the Python static analysis tools.

4. EVALUATION

Here we evaluate the tools by (1) which methods they use, (2) how mature they are, (3) what kind of problems they detect.

4.1 Typical Python bugs and error tested

It is true that the compiler relies on static analysis, similar to these tools in order to generate its warnings during compilation time. However, its primary task is to translate a compilable code from high-level programming language to a lower level language, and the attention to its essential characteristic – the compilation time, makes it less capable of doing static analysis than the “third-party” static analysis tools. Python compiler misses all of the following discussed bugs and errors. Here is a short overview of the dummy code snippets used as a test examples of typical logical errors and bugs, used in this study to analyze and test the power of the static analysis tools in Python.

4.1.1 *Referencing undefined variables.* Very often in Python it happens that due to a typing mistake, an undefined variable is referenced. However, due to the dynamic nature of the language, such bugs don not get discovered up until runtime and when they are referenced. Example code for testing this bug with the static analysis tools of Python:

```
1 import sys
2
3 message = "Hello there!"
4 if "--greetMe" in sys.argv:
5   print(mesage)
6 print("This code is fine, no problems.")
```

It can be noticed that at the line 6 the variable “mesage” is mistyped, and does not refer to the string “Hello there!” even though it looks similar. It is possible that error like this might not appear even during runtime. For example if this Python code is run without the “–greetMe” as an argument, then the code will just print out “This code is fine, no problems.” without errors or warnings.

4.1.2 *Too many positional arguments.* The following dummy code beside the “too many positional arguments” error in the call of the method getAge, has 3 more issues. The getAge method itself does not have a “self” argument. Further, the platform variable is undefined, this is similar to the previous example. And the last issue the sys package is imported but never used. Unused imports might slow down the program.

```
import sys

class Person:

  def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age

  if "Windows" in platform.platform():
    print("You're using Windows!")

  self.age = self.getAge(1, 2, 3)

  def getAge(this):
    return "18"
```

4.1.3 *Passing parameter of a different type then intended.* Since Python is a dynamic language variables do not have types in Python, but rather values do, and the type of the values as well as the purpose for what an object can be used is decided during runtime. Python annotations were introduced to add hints for the type of a variable, however the type annotations issues are not caught by the compiler, so it is of a big importance for the other static analysis tools to discover that. The following code snippet is used to test the tools against and describes the importance of type annotations in Python:

```
def sum(x:int, y:int):
  return x + y
print(sum(3,4))
print(sum(3, '4'))
```

Considering the sum function here, Python compiler does not report any issue, but during runtime the program throws an error which explains function sum can not be called with string since int and str types can not be sum up together with the + operator in Python.

4.1.4 *Non-existing class member.* The following example is used to test the tools against a code which refers to an unexisting class member. As it can be seen, the program tries to print the PERSON1's height, but height does not exists as an attribute in the class Person.

```
class Person():
  def __init__(self, name, age):
    self.name = name
    self.age = age
PERSON1 = Person("Hristina", 23)
print(PERSON1.age)
print(PERSON1.height)
```

4.1.5 *Nested function call.* A call to a nested function is a big logical error, but still the Python compiler does not generate any warning about it. The following dummy code is used to test the static analysis tools against call of a nested function outside its scope:

```
def outer():
  x=3
  def inner():
    print(x)
  inner()
outer()
inner()
```

4.1.6 *Closure bugs.* In Python, Closure is a concept that allows function object to remember values in enclosing scopes even if they are not present in memory.

```
#Python Closure
def outer():
  x=3
  def inner():
    y=3
    result = x + y
    return result
  return inner

a = outer()
print(a()) #this is the same as a call to the inner()
print(a.__name__) # prints the name of the actual function
                  # which is called - inner.
```

The following is an example code used to test the Python static analysis tools against a closure bugs in Python which is not reported by the Python compiler and the program runs just fine, with an unexpected output:

```
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 ")
```

We may expect that this code would printout:

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

But instead it is printing:

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

4.2 Test results discussion

Table I. : Python static analyzer capabilities to detect typical errors

#	Typical error	PyLint	Pyflakes	Flake8	Mypy	Frosted
1.	Referencing undefined variables	X	X	X	X	X
2.	Too many positional arguments	X	-	-	-	-
3.	Passing a parameter of a different type then intended	-	-	-	X	-
4.	Nonexistent class members	X	-	-	X	-
5.	Nested function call	X	X	X	X	X
6.	Closure bugs	X	-	-	-	-

On Table I we summarize our test results. The accent of the discussed code tests is mostly put on the logical errors that are slipping into Python developers everyday work, thus at this point, code style focused static analysis tools (such as Pycodestyle) are not considered in the result table. This result discussion is based on catching-coverage of the logical errors, confirming if the tool's behavior is as it is described in their documentation as well as a general overview of spotted pros and cons of each of the tools used against the test code examples. Currently, just a short general overview of the output format of the tools is given, but since this is also an important aspect of the tool's functionality, especially when they are used against a bigger Python code based, more detailed analysis of this will take part further in the study. To clarify, all the tools are tested while in their default configuration. The results might differ if the tool's configuration is separately tuned according the code tested.

Upon initial inspection of the table results, Pylint outperformed the other tools when run against the described test examples. It gave the most accurate reports and it was also the most descriptive in the generated output. However, if we look at the output format and style (which is not the focus of the table results at present), it is apparent that in many cases Pylint is "too talkative", meaning the output can quickly get noisy and not so easily readable as the code base expands. Yet, the convenience of Pylint is again proven as it is the only tool that reports the errors "Too many positional arguments" and "Closure bugs", which are easily made mistakes and the first one is especially trivial, but left to be caught during runtime, which is much more expensive. As such, they are of a great importance to the static analysis and the tools clearly need a support in their improvement. The "Closure bug" is a bit more complex. Closure in Python is an important concept and it is prone to bugs which in many cases, shown in the dummy test example as well, is hardly caught even during runtime. Such errors can take even days of debugging and manual reviews by the developers. This is notably true for bugs hiding in the Python closure concept. The authors believe that with the implementations of symbolic execution, the Closure is one of the concepts in Python that might see great improvements regarding static analysis by moving it ahead from the current benefits of the AST approach.

There is only one test case where Pylint did not report any warning nor error, and to a positive surprise, Mypy was the only tool that caught it. This was a type annotation related bug where Mypy proved that even though it is still considered as an experimental tool and far from the commonly preferred Pylint, it justified its focus to perform well on type annotation errors. This makes Mypy an essential static analysis tool to be considered in a Python code that uses type annotations gain on the benefits of the predefined variable types, which is not in the core of Python as a dynamic programming language. The importance of type annotation static analysis is shown in the "Passing parameter of a different type then intended" example and its "sum" function. Seeing that analyzing type annotations is one important point of the static analysis tools that needs to be improved, and at the same time noticing that the tools lack the symbolic execution as a method technique behind the scenes which seems to be a great factor of this improvement, encourages author's further directions of the research to go towards implementing and testing symbolic execution in Python static analysis. On the brighter side of the results table, there are two tests that were covered with a warning report by all of the static analysis tools under test. These are "Referencing undefined variables" and "Nested function call", both of which are quite trivial errors. The example "Referencing undefined variables" in a strictly typed language such as C, C++ and Java, this will be immediately reported by the compiler, but since Python is a dynamically typed language, its compiler does not catch it. Thus, this is one of the crucial moments for the Python static analyzers to shine up above the Python compiler. Anyhow, the strong believe that symbolic execution can improve the performance even in the most crucial and trivial examples, stays alive and will be considered in the future of this research.

Since the focus now is in the logical errors caught by the static analysis tools, it is important to notice that Pyflakes and flake8 are expected to, and they do give, the same results in the table. This is mainly because flake8 just combines pyflakes, pycodestyle and mccabe, so they have the same approach in catching the logical errors. However, noticing that Pyflakes did not catch many of the logical errors that were caught by the other tools, knowing that as a tool it focuses mostly on the logical errors avoiding the style reports, it raises a lot of questions. Moreover, leaves the believes that its very strict avoidance of false negative error, adds more cons than pros to its convenience as a static analysis tool, since because of this, many errors were left not reported at all. Frosted as a tool during this test examples did not show that catches any different errors than what Pyflakes and flake8 caught, which is to a certain degree expectable, since it is a fork of pyflakes, with the main difference being that its open to a public development.

To sum up this brief introductory discussion and analysis of the existing Python static analysis tools, it can be said that there is no tool presently that can stand alone as the best of all. Best results at the moment would be achieved when using several of the existing tools at the same time, thanks to their manually configurable and customizable capabilities. Exploring Python symbolic execution further in the research, promises that would bring great improvements in the currently existing Python static analysis world.

5. CONCLUSION

Static analysis is a powerful technique to validate large software systems in the early phase of the Continuous Integration (CI) loop. There are sophisticated methods for analyzing mainstream languages with static type system, like C, C++ and Java. The current support for the Python programming language is way behind. Tools are in experimental phase and lacking large development communities. The main reason is the dynamic nature of the language which makes harder to apply the usual analyzer techniques. However, this can be improved as tools can step forward from the current AST based methods. The most promising direction is symbolic execution, which can handle the dynamic language features in a more straightforward way. The authors suggest future researches to implement further symbolic execution techniques and test them in industrial environment.

REFERENCES

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

Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering* MSR-TR-2015-95 (January 2015). https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. DOI:http://dx.doi.org/10.1145/1646353.1646374

Barry Boehm and Victor R. Basili. 2001. Software Defect Reduction Top 10 List. *Computer* 34, 1 (Jan. 2001), 135–137. DOI:http://dx.doi.org/10.1109/2.962984

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

Clang SA. 2019. Clang Static Analyzer. (2019). https://clang-analyzer.llvm.org/.

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

Ian Cordasco. 2016. Flake8 documentation. (2016). http://flake8.pycqa.org/en/latest/.

Ian Cordasco. 2017. Mccabe documentation. (2017). https://pypi.org/project/mccabe/.

Patrick Cousot and Radhia Cousot. 1977. 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*. ACM, 238–252.

Timothy Crosley. 2014. Frosted documentation. (2014). https://pypi.org/project/frosted/.

Björn I. Dahlgren. 2016. Pysym documentation. (2016). https://pythonhosted.org/pysym/.

Artem Dergachev. 2016. Clang Static Analyzer: A Checker Developer's Guide. (2016). https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 28-02-2019).

Google. 2018. Google Python Style Guide. (2018). http://google.github.io/styleguide/pyguide.html.

GrammaTech. 2019. CodeSonar. (2019). https://www.grammatech.com/products/codesonar (last accessed: 28-02-2019).

Hari Hampapuram, Yue Yang, and Manuvir Das. 2005. Symbolic Path Simulation in Path-sensitive Dataflow Analysis. *SIGSOFT Softw. Eng. Notes* 31, 1 (Sept. 2005), 52–58. DOI:http://dx.doi.org/10.1145/1108768.1108808

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. DOI:http://dx.doi.org/10.1145/360248.360252

Jukka Lehtosalo. 2016. Mypy documentation. (2016). https://mypy.readthedocs.io/en/latest/.

Logilab. 2003. Pylint documentation. (2003). http://pylint.pycqa.org/en/latest/.

Logilab. 2019. Astroid documentation. (2019). https://astroid.readthedocs.io/en/latest/.

Daniel Marjamäki. 2013. CppCheck: a tool for static C/C++ code analysis. (2013). http://cppcheck.sourceforge.net/

Martin Moene. 2014. Search with CppCheck. *Overload Journal* 120 (2014). https://accu.org/index.php/journals/1898

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. DOI:http://dx.doi.org/10.1145/1273442.1250746

PyCQA. 2014. Pyflakes documentation. (2014). https://pypi.org/project/pyflakes/.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. DOI:http://dx.doi.org/10.1145/199448.199462

Johann C. Rocholl. 2006. Pycodestyle documentation. (2006). http://pycodestyle.pycqa.org/en/latest/.

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

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

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

Tiobe. 2019. TIOBE Programming Community Index, July 2019. (2019). http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html accessed 02-July-2019.

Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. 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)*. Springer-Verlag, Berlin, Heidelberg, 535–548. http://dl.acm.org/citation.cfm?id=1939281.1939332

Anna Zaks and Jordan Rose. 2012. Building a Checker in 24 hours. (2012). https://www.youtube.com/watch?v=kdxlsP5QVPw.