Towards More Sophisticated Static Analysis Methods of Python Programs

Hristina Gulabovska
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
hristina@gulab.me

Zoltán Porkoláb
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
gsd@elte.hu

***Abstract*—Static analysis is a software verification method which is analyzing the source code without executing it for detecting code smells and possible software bugs. Various analysis methods have been successfully applied for languages with static type system, such as C, C++ and Java. Python is an important programming language with dynamic type system, used in many emerging areas, including data science, machine learning and web applications. The dynamic behavior of the Python language requires different static analysis approaches compared to the ones with static type system. In this paper we overview these methods and investigate their advantages and shortages. We compare the symbolic execution with the generally used Abstract Syntax Tree based approach and show its advantages based on concrete examples. We also highlight the restrictions of current tools and suggest further research directions to tackle these problems.**

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

I. Introduction

Python is one of the most rapidly emerging programming language [1]. Being flexible and expressive, it is very popular to implement 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 bind to a certain (variable) name, the value has a type but not the variable. One can assign to an existing variable a new value with a possibly different type. This would be a compile time error in statically typed languages such as Pascal, C or Java, but allowed in Python. In 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 hold [2]. More dynamic features, such as calling methods dynamically, declaring dynamic attributes (using `getattr`), and others also increase the expressiveness and usability of the language.

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 used either as commercial [3], [4] or free, open source tools [5]–[8]. Although there are promising projects for Python (see Section III), 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 [9], [10]. Static analysis is an important aspects in modern software development, addressed by various academic and industrial researches, and projects like by the 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" [11], [12].

In this paper we investigate the possible research directions towards more powerful static analysis tools for the Python programming language. In Section II we evaluate the applicable analysis methods evaluating their strength and weaknesses. In Section III we briefly overview the most important tools and research directions currently available for analyzing Python systems. We use two specific tools to compare the AST based methods and the symbolic/concolic execution in Section IV. Our paper concludes in Section V.

II. Static analysis methods for software systems

To verify the correctness of a software system we can choose between various 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. As the earlier a bug is detected, the lower the cost of the fix [13], testing is not ideal in this aspect.

Alternatively we might turn to analyzer tools which applies 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* [14], or *Google Address sanitizer* [15] which work run-time, evaluate the correctness of 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.

In the case of static analysis we do not run the software. Instead, the input of the static analysers are the source code, and we apply various methods to find dangerous constructs without running the program. Static analysis is a popular

978-1-7281-3180-1/19/$31.00 ©2019 IEEE

method for finding bugs and code smells [16] as they do not depend on the selection of input data while they can (at least theoretically) provide full coverage of the code. An other advantage of the static analysis method is that it is many cases applicable for only a part of the code. This is useful when we have no full control over the system, e.g. we use third party libraries, not all source is available, or we just have no resources to check the whole system.

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 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 only 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. A general approach for static analysis tools to try the balance between the large number of true positives and a relatively low number of false positives. When in doubt, many tools rather choose to drop real findings to minimize false alarms.

A. 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 matches. Although, this method seems to be very simple, its huge advantage is the low level of false positives, as well-written regular expressions have easy to predict results. Early versions of CppCheck [17] used pattern matching to find issues in C and C++ programs. 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 [18].

B. 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 [19]. 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, such as 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, such as the Clang Tidy [6] 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, 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, that it cannot reason about the possible program states which can be dependant from input values, function parameters.

C. Symbolic execution

Executing *abstract interpretation* [20] the tool reasons about the possible values of variables at a certain program point. *Symbolic execution* [21], [22] 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 [23]. 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 [7], Klocwork [4], and Coverity [3] as well as open source products such as the Clang Static Analyzer [5], and Infer [8] 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* [24]. 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.

D. Concolic execution

While the symbolic execution of a program can generate all possible control flow path that can occur during the execution, in practice it is unfeasible for many reasons. Most of the constraint solvers has some limitations on their modelling capabilities. Also, generic constraint solvers working e.g. on intervals may be resource (time and memory) intensive thus make full symbolic execution impossible.

An interesting mixture of symbolic and concrete execution is called as *concolic* execution [25] 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. [26].

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 keep follow all the possible execution paths. Especially for languages such as Python, where the interpreter could evaluate the analysed program making possible to call unmodeled external methods or using third party modules this approach is seriously extending the power of the symbolic execution.

III. Analysis tools for Python

Compiler relies on static analysis to generate its warnings during compilation time. However, since its primary task is to translate source code from a high-level programming language to a lower level language, it is less capable of doing static analysis than some of the third-party tools. 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 [27], Pyflakes [28], flake8 [29], Frosted [30], Pycodestyle [31], and Mypy [32]. 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 bug and errors of the source code. Pylint is seen as the most popular Python static analysis tool at the moment, 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 most of the checkers are working on the Abstract Syntax Tree (AST) which is provided by the astroid [33] library. Adding to the reliability of Pylint, it is worth mentioning that it is trusted by many big companies, such as Google [34], 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: 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 [35], PySym [36], PySMT [37], and mini-mc [38], etc. Most of them are using the Z3-solver [39].

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 helps 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.

```
1 def greet(greet_word, name):
2     print(greet_word, name)
3
4 greeters = list()
5 names = ["Kiki", "Riki", "Joe"]
6 for name in names:
7     greeters.append(lambda x: greet(x, name))
8 for greeter in greeters:
9     greeter("Hi ")
```

Listing 1: The closure bug

We may expect that this code would printout:

```
1 Hi Kiki
2 Hi Riki
3 Hi Joe
```

But instead it prints:

```
1 Hi Joe
2 Hi Joe
3 Hi Joe
```

The closure bug is one of the most tricky 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.

IV. Evaluation

In this section, we compare the power of the AST-based method with the symbolic execution. We were selecting two representative tools for the two methods, run tests with them, and analyze 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 a suspicious code segment.

Most of the currently existing Python tools use the AST method for static analysis of the Python source code. We have selected Pylint [27] as one of the most widely used and most powerful static analysis tool for Python which is using the AST method based on the asteroid library.

For the symbolic execution method we have chosen mini-mc, an experimental symbolic execution implementation [38], using the Z3's Python interface. The mini-mc tool is implementing the fork-explore-check idea, i.e. when evaluating symbolic values the Python VM will try to convert them into boolean values at all branches to intercept the conversion and replace it with a `fork` statement. In practice mini-mc will process all reachable program paths, forking new process to evaluate the false branch. It also detects unreachable conditions where the evaluation stops. We have chosen mini-mc for its simplicity and demonstrative power.

Analyzing the methods for static analysis and noticing that symbolic execution might be the best approach for static analyzing of Python considering its dynamically typed characteristics. In this section, we composed a few examples to see step by step the symbolic evaluation and then compare if Pylint as a AST based static analyzer or mini-mc as a symbolic execution method could catch 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 the 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. We used this example mostly to demonstrate the way of 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 show the non deterministic evaluation order of the branches). When the engine detects unsafisfiable condition, that is also printed as `unreachable`.

```
1  [7088] assume (arg == 1)
2  [7090] assume ¬(arg == 1)
3  [7090] unreachable
4  [7088] assume (arg == 1)
5  [7088] assume (arg == 1)
6  branch11 7088
7  branch11 7088
8  branch21 7088
9  [7090] exit
10 [7089] assume ¬(arg == 1)
11 [7089] assume (arg == 1)
12 [7089] unreachable
13 branch12 7089
14 [7089] assume ¬(arg == 1)
15 [7091] assume ¬(arg == 1)
16 branch12 7089
17 branch22 7091
18 [7091] exit
19 [7089] exit
20 [7088] exit
```

Listing 3: Execution result of Listing 2.

The next Python example on Listing 4 is faulty and should point out the power of symbolic execution over the AST based approaches. As the two if-conditions could not be true at the same time, if `1!=arg` then the first if-condition body would not be executed, and when the second if-condition body will be executed, the program gets 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
9  arg = BitVec("arg",32)
10 func(arg)
```

Listing 4: Local variable referenced before assignment.

The result shows that symbolic execution finds the possible error while the AST based approach will miss it. When PyLint as an AST based static analyzer was run, it did not detect and report any potential error in the program. PyLint was unable to recognize that `x=z` will be executed only after statement `z=1` is not. Pylint is assuming that variable `z` is defined in one of the executed branches and conservatively do not report error to minimize possible false positives.

When the mini-mc symbolic execution checker was run and the steps of the execution were provided it could be seen that the Unbound error at line 14 was detected (Listing 5). This shows the benefits of the 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: Execution result of Listing 4.

On Listing 6 we changed the code in order to check the behavior of symbolic evaluation when instead of concrete values, intervals are used in the if 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. The results on Listing 7 shows that the symbolic execution was working just fine when we used intervals.

```
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.

```
[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.

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 II-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.

```
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

The following result on Listing 9 shows that the symbolic execution method 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: False positive report of Listing 8.

The reason is, that the concolic execution accidently takes 42 as the sample value for `arg` when `arg != 41`. This makes the otherwise unlikely situation of dividing by zero unavoidable. Altough such unlucky situatuions may be unfrequent 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, this example shows that it is far from perfect and there is room to improve it.

V. Conclusion

Static analysis methods evaluate software systems without running them, applying various heuristics on the source code to detect possible code vulnerabilities. Since these do not require to choose specific input values, they are easier to integrate into the Continuous Integration loop. However, static analysis 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 not satisfactory.

Symbolic execution techniques may open new directions for supporting Python static analysis. The dynamic behavior of the language, such as the usage of variables without preceding declaration and changing its type during runtime 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 most simple 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 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.

Acknowledgment

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

References

[1] Tiobe. (2019) TIOBE programming community index, july 2019. TIOBE Software. Accessed 02-July-2019. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] G. van Rossum, "Python tutorial," Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Tech. Rep. CS-R9526, May 1995.

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

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

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

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

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

[8] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 459–465.

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

[10] A. Zaks and J. Rose, "Building a checker in 24 hours," 2012, https://www.youtube.com/watch?v=kdxlsP5QVPw.

[11] Š Korečko, "Interactive approach to coloured petri nets teaching," Eötvös Loránd University, Faculty of Informatics, Budapest, Tech. Rep. IK-TR3, May 2018.

[12] C. Szabó. (2018) Programme of the winter school of project no.2017-1-sk01-ka203-035402: "focusing education on composability, comprehensibility and correctness of working software". TUKE Kosice. Accessed 02-July-2019. [Online]. Available: https://kpi.fei.tuke.sk/sites/www2.kpi.fei.tuke.sk/files/personal/programme_of_the_first_intensive_programme_for_higher_education_learners_in_the_frame_of_the_project.pdf

[13] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1109/2.962984

[14] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342849

[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374

[17] D. Marjamäki, "CppCheck: a tool for static C/C++ code analysis," 2013. [Online]. Available: http://cppcheck.sourceforge.net/

[18] M. Moene, "Search with cppcheck," *Overload Journal*, vol. 120, 2014. [Online]. Available: https://accu.org/index.php/journals/1898

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

[20] P. Cousot and R. 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*. ACM, 1977, pp. 238–252.

[21] H. Hampapuram, Y. Yang, and M. Das, "Symbolic path simulation in path-sensitive dataflow analysis," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 52–58, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108768.1108808

[22] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[23] Z. Xu, T. Kremenek, and J. 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*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 535–548. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939281.1939332

[24] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[25] K. Sen, "Concolic testing," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 571–572. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321746

[26] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[27] Logilab. (2003) Pylint. http://pylint.pycqa.org/en/latest/.

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

[29] I. Cordasco. (2016) Flake8. http://flake8.pycqa.org/en/latest/.

[30] T. Crosley. (2014) Frosted. https://pypi.org/project/frosted/.

[31] J. Rocholl. (2006) Pycodestyle. http://pycodestyle.pycqa.org/en/latest/.

[32] J. Lehtosalo. (2016) Mypy. https://mypy.readthedocs.io/en/latest/.

[33] Logilab. (2019) Astroid. https://astroid.readthedocs.io/en/latest/.

[34] Google. (2018) Google python style guide. http://google.github.io/styleguide/pyguide.html.

[35] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution," *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, no. MSR-TR-2015-95, January 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/

[36] B. I. Dahlgren. (2016) Pysym. https://pythonhosted.org/pysym/.

[37] Y. Malheiros. (2016) pysmt. https://pysmt.readthedocs.io/en/latest/tutorials.html.

[38] X. Wang. (2015) A mini symbolic execution engine. http://kqueue.org/blog/2015/05/26/mini-mc/.

[39] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24