ESBMC-Python: A Bounded Model Checker for Python Programs

Bruno Farias
University of Manchester
Manchester, UK
bruno.farias@manchester.ac.uk

Rafael Menezes
University of Manchester
Manchester, UK
rafael.menezes@postgrad.manchester.ac.uk

Eddie B. de Lima Filho
TPV Technology
Manaus, Brazil
eddie.filho@tpv-tech.com

Youcheng Sun
University of Manchester
Manchester, UK
youcheng.sun@manchester.ac.uk

Lucas C. Cordeiro
University of Manchester
Manchester, UK
lucas.cordeiro@manchester.ac.uk

ABSTRACT

This paper introduces a tool for verifying Python programs, which, using type annotation and front-end processing, can harness the capabilities of a bounded model-checking (BMC) pipeline. It transforms an input program into an abstract syntax tree to infer and add type information. Then, it translates Python expressions and statements into an intermediate representation. Finally, it converts this description into formulae evaluated with satisfiability modulo theories (SMT) solvers. The proposed approach was realized with the efficient SMT-based bounded model checker (ESBMC), which resulted in a tool called ESBMC-Python, the first BMC-based Python-code verifier. Experimental results, with a test suite specifically developed for this purpose, showed its effectiveness, where successful and failed tests were correctly evaluated. Moreover, it found a real problem in the Ethereum Consensus Specification.

CCS CONCEPTS

• **Theory of computation → Verification by model checking**; • **Software and its engineering → Formal software verification**.

KEYWORDS

Formal Verification, Bounded Model Checking, Python

ACM Reference Format:
Bruno Farias, Rafael Menezes, Eddie B. de Lima Filho, Youcheng Sun, and Lucas C. Cordeiro. 2024. ESBMC-Python: A Bounded Model Checker for Python Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Python is an interpreted and multi-paradigm programming language to develop software systems, including general tasks, web applications, image processing, and artificial intelligence (AI) [25]. Regarding the latter, the presence of Python code is particularly significant due to its extensive libraries, such as TensorFlow [1], PyTorch [19], and Keras[13]. Indeed, its simple syntax, typing facilities, and resources made it popular, leading to its use in systems with critical security requirements. In contrast, its dynamic nature also hampers the development of static analyzers to ensure correctness.

A technique that can be used to check Python programs, often employed for software verification, is bounded model checking (BMC) [3]. Based on it, different languages can be tackled with specific tools or even front-ends for existing frameworks [17]. Moreover, the latter may harness the capacity of verification engines and then lead to more comprehensive and accurate results [16, 24].

However, the BMC's potential for verifying Python programs remains unexplored [21]. Again, it is mainly due to its dynamic nature, which lacks explicit type information, unlike languages such as C [15]. Indeed, in Python, concrete-type information is assigned during execution by its interpreter, which makes it harder for verifiers to evaluate correctness, given that they rely on such knowledge. To tackle this, some studies have converted languages without explicit type information into C code, making model checking possible [18].

Although this seems to lead to a dead-end, an aspect should be mentioned: the Python syntax allows annotation with type information on variables and functions. Consequently, this instrument, together with other resources, such as abstract syntax trees (ASTs) and satisfiability modulo theories (SMT) or Boolean satisfiability (SAT) solvers, could be used for reasoning about a program's states.

In other words, type annotations in Python code could favor its analysis by a BMC tool, such as the efficient SMT-based bounded model checker (ESBMC) [8]. This formal verifier has already been applied to many systems, including digital filters [2], controllers [7, 10], and unmanned aerial vehicles [6]. Such a track record assures a distinct level to it, which, with new languages and features, can expand its applicability and provide a functional approach.

The last paragraphs outline the inspiration for the present work, which proposes a scheme to make BMC tools capable of processing Python code. It converts the latter into an AST structure, which is then type-annotated and formatted to provide a description suitable to a BMC pipeline. Aiming at evaluation, we have implemented this approach using ESBMC due to its mature and well-proved engine.

The proposed approach includes a front-end to generate ASTs from Python programs. These elements serve as interfaces between Python source code and the ESBMC's internal model-checking structure, thus translating a program into an intermediate representation (IR) that it can analyze. Subsequently, the ESBMC's back-end

ISSTA 2024, 16-20 September, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

generates first-order logical formulae for a program's constraints and safety properties, aiming at formal verification. Ultimately, such formulae are submitted to an SMT solver to check for satisfiability.

The resulting tool was named ESBMC-Python and used to evaluate a benchmark suite. The latter is a collection of Python programs created to assess our tool and allow comparison with similar ones, which is another contribution of ours that can be used for evaluating Python verifiers. In this context, ESBMC-Python verified Python programs in a few milliseconds (30 ms on average), automatically identifying violations related to user-defined assertions and arithmetic and logical operations. We have also used ESBMC-Python to check the Ethereum consensus specification [5]. As a result, it found an issue confirmed and fixed by the respective maintainers.

2 TOOL DESCRIPTION

ESBMC receives source code as input and generates AST descriptions, which identify the different operations within a program and include relevant information, such as statement location and variable type and size. Next, program statements are translated into symbols and added to a structure representing a program's symbol table (ST) using the ESBMC's IR format (IRep). Indeed, the key aspect of using the BMC pipeline implemented in ESBMC is its ST, which must be the final result of a Python front-end.

Moreover, Python offers type annotation for variables, function parameters, and return values, which does not affect its runtime behavior or generate errors. Indeed, it should be used in static analysis to complement the information in a resulting AST description. Consequently, we have implemented ESBMC-Python using the libraries *ast* [22] and *ast2json* [20] together with ESBMC.

2.1 The Python Verification Architecture

The Python verification scheme proposed here results directly in a front-end for ESBMC, reusing its infrastructure and back-end components. This complete arrangement is called ESBMC-Python, whose architecture is shown in Fig. 1. The front-end includes code parsing, semantics analysis, and statement translation into an ESBMC's ST. The gray elements represent new front-end components, while the others are preexisting ones reused by ESBMC-Python.

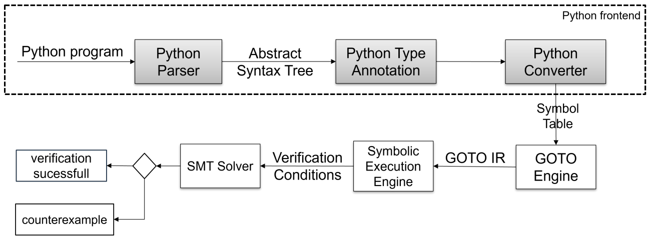

Figure 1: ESBMC-Python architecture.

Python Parser. The processing begins with *Python Parser*, which analyzes the input code structure and generates the corresponding AST, using the module *ast*. Then, the respective output is passed to *ast2json* for JSON conversion. Specifically, an instance of the Python interpreter is invoked to run a script that employs these libraries, ensuring that a program's behavior is correctly represented. Additionally, *Python parser* can handle user options from ESBMC to print AST content, i.e., *–parse-tree-too*, and isolate functions, i.e., *–function*. This last feature allows the verification of an entire file or its functions, which is useful to avoid the analyses of unsupported parts or reduce verification times. The output of this component is a file *ast.json* containing the input program's structure.

Python Type Annotation. Next, *Python Type Annotation* adds type annotations to the output AST using type inference [12], i.e., it inserts new nodes with typing information for a program's variables. This task involves processing an AST and adding nodes *AnnAssigned* containing the field *id* with specific type information [22].

Python Converter. Ending the front-end processing, *Python converter* turns the definition of classes, methods, and functions into an ST in IRep. Specifically, it iterates over expressions in functions, conditional blocks, and loops, converting each operation with the available ESBMC's application programming interface (API).

Subsequently, the existing ESBMC's pipeline retrieves expressions from the resulting ST and converts them into GOTO language, which is considered another IR and represents its control flow graph (CFG). It transforms the program logic into a simplified representation based on assignments, conditional and unconditional branches, assumes, and assertions. Then, a symbolic execution interprets a bounded execution of the GOTO program, resulting in its static single assignment (SSA) trace [9]. In SSA forms, all assignments over variables construct new symbols that can be combined using ϕ-functions, ultimately generating Boolean formulae. Such elements represent a verification condition (VC) $C \wedge \neg P$ submitted to a solver, where C means constraints and P denotes a safety property.

If a specific function F is verified, using *–function*, *Python Converter* converts only F and then adds its call, passing non-deterministic values as parameters. This process evaluates F with all possible values of a given type, helping identify issues that often go unnoticed.

Note that ESBMC was not designed to handle object-oriented programming (OOP)[8]. Therefore, we had to model OOP features with structured programming. For instance, *Python Converter* resolves calls to overloaded or inherited methods by searching for their definitions in base classes, respecting class ordering in inheritance lists. Moreover, when class attributes are not defined for a given class, it looks for them in base classes from the respective ST.

ESBMC-Python supports built-in types, i.e., *int*, *float*, and *boolean*, and basic structures. The latter include logical operations, comparisons, assignments, asserts, conditionals, loops, functions, module imports, classes, inheritance, and polymorphism. Finally, regarding verification properties, it can detect division by zero, arithmetic overflows, out-of-bounds array access, and user-defined assertions.

2.2 Illustrative Example

In this section, we explain tool usage for the program in Listing 1, which includes a recursive function for the factorial of an integer. Its input variable n is initialized with a non-deterministic value at line 7, whose range, with calls to *ESBMC_assume*, is constrained to $[1, 5]$, at line 8. Then, $factorial$ is called, at line 9, and, finally, an assertion checks that its return can not be 120, at line 10.

Listing 2 contains the AST in JSON format corresponding to the assignment to n that occurs at line 7 of Listing 1. Indeed, ESBMC-Python manipulates an AST by transforming simple assignments into annotated ones during its program parsing process explained in Section 2.1, using *AnnAssign* nodes. In this case, the variable *result* is initialized with the value returned by *nondet_int*.

ESBMC-Python verifies whether the negation of a property is satisfied, which becomes a satisfiability problem. Using SSA, it creates C with assignments, interval restrictions, and the factorial computation itself, leading to $C = [n = nondet() \wedge n > 0 \wedge n < 6 \wedge result = \phi(n = 1 \longrightarrow 1, n = 2 \longrightarrow 2, n = 3 \longrightarrow 6, n = 4 \longrightarrow 24, n = 5 \longrightarrow 120)]$, while P uses line 10, resulting in $P = [result \neq 120]$. Then, the corresponding VC is submitted to a solver, which tries to find a value combination that satisfies $C \wedge \neg P$.

This program can be verified by running the binary *esbmc* [1] with our Python front-end already integrated, passing its Python file name as a parameter. Assuming a file *main.py*, our tool can be executed using the command "*$esbmc main.py –unwind 5*", where *–unwind* limits the recursion depth during symbolic execution.

Fig. 2 contains the verification output for the same program in Listing 1. As one may notice, the solver within the ESBMC's pipeline detects a value of 120 when n is equal to 5.

Listing 1: A Python program verifiable by ESBMC-Python.

```python
1  def factorial(n:int) -> int:
2      if n == 0 or n == 1:
3          return 1
4      else:
5          return n * factorial(n - 1)
6
7  n:int = nondet_int()
8  __ESBMC_assume(n > 0 and n < 6)
9  result:int = factorial(n)
10 assert(result != 120)
```

Listing 2: AST in JSON for an annotated assignment.

```json
{
  "_type": "AnnAssign",
  "annotation": {
      "_type": "Name",
      "id": "int",
  },
  "target": {
    "_type": "Name",
    "id": "n"
  },
  "value": {
    "_type": "Call",
    "args": [],
    "func": {
      "_type": "Name",
      "id": "nondet_int",
    },
  }
}
```

3 EXPERIMENTAL EVALUATION

Here, we present ESBMC-Python's verification results, which intend to answer two experimental questions (EQ):

- **EQ1 (soundness)** - can our approach report known wrong programs and preserve the correct ones, presenting soundness?
- **EQ2 (performance)** - what are the time and memory performances associated to our approach?

In this context, soundness refers to the capacity to ensure that no correct program is considered wrong.

```
ESBMC version 7.6.1 64-bit x86_64 linux
Parsing main.py
Converting
Generating GOTO Program
GOTO program creation time: 0.023s
...
Building error trace
[Counterexample]
State 1 file main.py line 7 column 0 thread 0
---------------------------------------------
  n = 5 (00000000 00000000 00000000 00000101)

State 4  thread 0
---------------------------------------------
Violated property:
  assertion
  result != 120

VERIFICATION FAILED
```

Figure 2: ESBMC-Python output.

To answer these questions, we created a benchmark suite of 85 programs[2], each named for its target feature, split across 15 categories. There are at least two tests: one with a failing assertion and the other with one or more passing assertions. We also created extra elements for sensitive features such as imports and functions.

Moreover, we assess ESBMC-Python's performance in handling different Python expressions by measuring memory usage and verification times using the Linux *time* tool. The obtained results report total computer processing unit (CPU) times, including user and system portions, thus gathering the real CPU occupation.

Our benchmark suite encompasses features usually found in real-world Python programs: arithmetic operations, conditionals, loops, user assertions, bit-wise operations, classes, objects, class attributes, instance attributes, inheritance, polymorphism, function definitions, function calls, recursive functions, module imports, non-determinism, and assume directives. This way, it should not be considered only suitable for simple validation or a set of toy examples, which often happens in initial studies [23]. Indeed, the absence of benchmarks for Python verification underscores its importance as a possible baseline for future investigations.

We checked our benchmark suite on a 64-bit Intel i7-12700H processor with 16 GB of RAM, running Ubuntu 22.04. Moreover, we used version 7.6.1 of ESBMC, following the compilation instructions in its project documentation [3]. Specifically regarding the ESBMC's verification pipeline, we employed version 3.2.3 of the solver Boolector [4].

All verification processes were successful. This way, ESBMC-Python identified programs with property violations and validated the ones with correct behavior, which answers **EQ1**.

EQ1: ESBMC-Python only detected property violations for wrong program elements, which included types, conditionals, loops, functions, user assertions, and OOP aspects.

Table 1 summarizes average results for memory usage and execution time, per test category. The highest and lowest average

[1]https://github.com/esbmc/esbmc/releases/tag/v7.6.1

[2]https://github.com/esbmc/esbmc/tree/master/regression/python

[3]https://github.com/esbmc/esbmc/blob/master/BUILDING.md

verification times were 49.1 ms and 24.5 ms, respectively, which, compared to what is obtained with BMC tools for similar programs in other languages, can be regarded as satisfactory [14]. It also means that large project repositories or extensive program sets could be verified in relatively short periods, automatically [11]. Regarding memory consumption, the highest and lowest amounts were 26.4 MB and 14.5 MB, respectively, which is also usual [14]. Moreover, the highest memory usage for an isolated test was 27 MB, which occurred in category *Classes*. It seems to be due to the representation of instance attributes and the necessary search for base classes, when inheritance is involved. Nevertheless, these figures are still considered low for modern personal computers. Finally, such results provide a summarized view of the ESBMC-Python's performance, thereby addressing **EQ2**.

Category	Test Cases	Mem. Usage	Exec. Time
Arith operations	2	26.4 MB	33.5 ms
Assignments	5	18.5 MB	38 ms
Assume	4	16.5 MB	28.2 ms
Binary operations	2	20.5 MB	29.5 ms
Binary types	4	20.4 MB	28.5 ms
Built-in functions	7	19.9 MB	28.1 ms
Classes	9	19 MB	27.1 ms
Conditionals	4	17.8 MB	25.5 ms
Functions	11	21.8 MB	30 ms
Imports	8	15.3 MB	49.1 ms
Logical operations	6	20.4 MB	24.5 ms
Loops	10	20.7 MB	35.4 ms
Non-determinism	4	21.4 MB	29.2 ms
Numeric types	6	20.9 MB	29.1 ms
Type annotation	3	14.5 MB	27.3 ms

Table 1: Results for ESBMC-Python regarding our test suite.

EQ2: ESBMC-Python presented execution time and memory consumption figures that are similar to what is noticed for BMC tools targeting other languages.

As far as we know, only one tool is similar to ESBMC-Python: modeling, simulation, and verification (MSV) [23]. However, there is no test set, reproducible results, or repository for retrieving its source code (see Section 4), which impedes a direct comparison.

3.1 Experimental Results for the Ethereum Consensus Specification

We also used ESBMC-Python to check the Ethereum blockchain consensus protocol, which comprises elements that control the Ethereum network's node inclusion, validation, and validator penalty processes. It is described with a set of markdown files, in a GitHub repository [4], containing functions that compose a reference API used to generate a Python library called *eth2spec*.

All functions in this specification include parameters and return values with annotated typing. This way, one could submit *eth2spec* for evaluation by ESBMC-Python. However, it also contains elements not initially handled by ESBMC-Python, which led to extensions for custom types such as *uint64*, *uint128*, and *uint256*.

We used ESBMC-Python to verify Python files generated for *eth2spec*, which involved testing each function individually (see Section 2.1). Specifically, we employed the parameter *–function* followed by the name of the function to be verified, thus checking specific elements with non-determinism in their invocations.

As a result, our evaluation revealed a division-by-zero in function *integer_squareroot*. This error occurred because an unsigned integer overflowed to zero, after being incremented, and was later used as the denominator of a division operation. Indeed, division-by-zero events in blockchains can lead to service interruptions, compromising their availability and facilitating attacks.

4 RELATED WORK

Shu *et al.* [23] proposed using the MSV language (MSVL) to describe and check Python programs with the MSV tool, utilizing rules to express Python's semantics in MSVL.

Although this work proposes a technique for automatic verification, its examples and functionalities are still basic. Hence, we can state that the MSV tool can not verify complex Python programs as found in industrial applications. Moreover, it is not available for download, preventing its evaluation, comparison, and further development. We have indeed attempted to contact its authors by email, but we did not receive a response while writing this paper.

5 TOOL AVAILABILITY

A video demonstration is available at https://t.ly/QTSdp, and tool artifacts and documentation can be found at https://t.ly/7PSFv.

6 CONCLUSIONS AND FUTURE WORK

This paper introduces ESBMC-Python, a novel tool to detect errors in Python programs. It builds a front-end for the ESBMC framework that translates Python expressions into symbols, which are converted into a GOTO program, symbolically executed, and checked for satisfiability. To the best of our knowledge, there is only one tool [23] to formally verify Python code with model checking, which highlights the importance of our work. However, a direct comparison was not possible due to code and result unavailability.

To evaluate ESBMC-Python's effectiveness, we conducted experiments using a test suite. On average, simple programs could be verified in 31.56ms. Moreover, such a suite can also serve as a reference for evaluating other Python verification tools. Additionally, ESBMC-Python was able to identify a real bug in the specification of the Ethereum blockchain consensus protocol.

In future work, we intend to extend ESBMC-Python's capabilities to include more features. Additionally, we aim to enhance our type inference algorithm to deal with more complex program flows. The potential of large language models will also be explored to address type inference for complex expressions and execution paths. Finally, we plan to develop operational models for verifying AI libraries.

ACKNOWLEDGMENTS

This project was supported by the Ethereum Foundation under Grant FY22-0751.

[4]https://github.com/ethereum/consensus-specs

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Renato B. Abreu, Mikhail R. Gadelha, Lucas C. Cordeiro, Eddie Batista de Lima Filho, and Waldir Sabino da Silva Jr. 2016. Bounded Model Checking For Fixed-point Digital Filters. *Journal of the Brazilian Computer Society* 22, 1 (2016), 1:1–1:20.

[3] Armin Biere. 2021. Bounded model checking. In *Handbook of satisfiability*. IOS press, 739–764.

[4] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer, 174–177.

[5] Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. 2022. Formal verification of the ethereum 2.0 beacon chain. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 167–182.

[6] Lennon C. Chaves, Iury Bessa, Hussama Ismail, Adriano Bruno dos Santos Frutuoso, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. 2018. DSVerifier-Aided Verification Applied To Attitude Control Software In Unmanned Aerial Vehicles. *IEEE Transactions on Reliability* 67, 4 (2018), 1420–1441.

[7] Lennon C. Chaves, Hussama Ibrahim Ismail, Iury Valente de Bessa, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. 2019. Verifying fragility in digital systems with uncertainties using DSVerifier *v*2.0. *J. Syst. Softw.* 153 (2019), 22–43. https://doi.org/10.1016/j.jss.2019.03.015

[8] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2011. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering* 38, 4 (2011), 957–974.

[9] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[10] Iury Valente de Bessa, Hussama Ibrahim Ismail, Lucas Carvalho Cordeiro, and Joao Edgar Chaves Filho. 2014. Verification of Delta Form Realization in Fixed-Point Digital Controllers Using Bounded Model Checking. In *Brazilian Symposium on Computing Systems Engineering*. 49–54.

[11] Janislley Oliveira de Sousa, Bruno Carvalho de Farias, Thales Araujo da Silva, Lucas C Cordeiro, et al. 2023. Finding Software Vulnerabilities in Open-Source C Projects via Bounded Model Checking. *arXiv preprint arXiv:2311.05281* (2023).

[12] Dominic Duggan and Frederick Bent. 1996. Explaining type inference. *Science of Computer Programming* 27, 1 (1996), 37–83.

[13] Nikhil Ketkar and Eder Santana. 2017. *Deep learning with Python*. Vol. 1. Springer.

[14] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 389–391.

[15] Magnus Madsen. 2015. Static analysis of dynamic languages. (2015).

[16] Rafael Menezes, Daniel Moura, Helena Cavalcante, Rosiane de Freitas, and Lucas C Cordeiro. 2022. ESBMC-Jimple: verifying Kotlin programs via jimple intermediate representation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 777–780.

[17] Felipe R Monteiro, Mikhail R Gadelha, and Lucas C Cordeiro. 2022. Model checking C++ programs. *Software Testing, Verification and Reliability* 32, 1 (2022), e1793.

[18] Felipe R Monteiro, Francisco AP Januário, Lucas C Cordeiro, and Eddie B de Lima Filho. 2017. BMCLua: A translator for model checking Lua programs. *ACM SIGSOFT Software Engineering Notes* 42, 3 (2017), 1–10.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[20] Laurent Peuch. 2024. ast2json. https://pypi.org/project/ast2json/ Accessed: 2024-06-03.

[21] Quoc-Sang Phan, Pasquale Malacaria, and Corina S Păsăreanu. 2015. Concurrent bounded model checking. *ACM SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–5.

[22] Python Software Foundation. 2024. ast - Abstract Syntax Trees. https://docs.python.org/3/library/ast.html Accessed: 2024-06-03.

[23] Xinfeng Shu, Fengyun Gao, Weiran Gao, Lili Zhang, Xiaobing Wang, and Liang Zhao. 2019. Model Checking Python Programs with MSVL. In *International Workshop on Structured Object-Oriented Formal Language and Method*. Springer, 205–224.

[24] Kunjian Song, Nedas Matulevicius, Eddie B de Lima Filho, and Lucas C Cordeiro. 2022. Esbmc-solidity: An smt-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 65–69.

[25] Guido Van Rossum and Fred L Drake Jr. 1995. Python tutorial.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009