Department of Computer Science
Bachelor Project Report

Verifying Python Programs in Robotic Applications

Author: Wenda Lu

Supervisor: Dr. Lucas Cordeiro

April 2022

Abstract

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics suited for most programs. Due to its properties, Python is frequently applied in the creation of robotic applications for associated industries. However, during the development process of these programmes, various faults may be encountered; for example, erroneous logical operators are operated, resulting in a count-by-one error in the loop, or even dividing by zero. The Efficient SMT-Based Context-Bounded Model Checker (ESBMC) is used as a backend in this research to provide an innovative verification technique for Python applications. This methodology, entitled bounded model checking - Python (BMCPython), implies converting Python applications to ANSI-C source code and then verifying them utilizing ESBMC. Research performed on Python programmes demonstrates that the proposed verification approach works productive and is efficient. Experiments prove that BMCPython generates an ANSI-C code that is fast to compile and verify. As far as we can tell, this study is the first to deploy bounded model checking to verify Python programs.

Acknowledgements

First and foremost, I would like to express my deepest appreciation to Dr Lucas Cordeiro, my project supervisor, for his unwavering support, patience, and guidance during the duration of my research project. In addition, he has provided me with valuable resources that have encouraged me to come up with a novel and creative design.

Second, I also want to express my gratitude to my family and friends for their encouragement and unwavering trust in my qualities.

Project Planning

I visualized my project planning timeline(as follows). During weekly tutorial meetings, I was able to summarize the work done this week and propose the project goals for the next week.

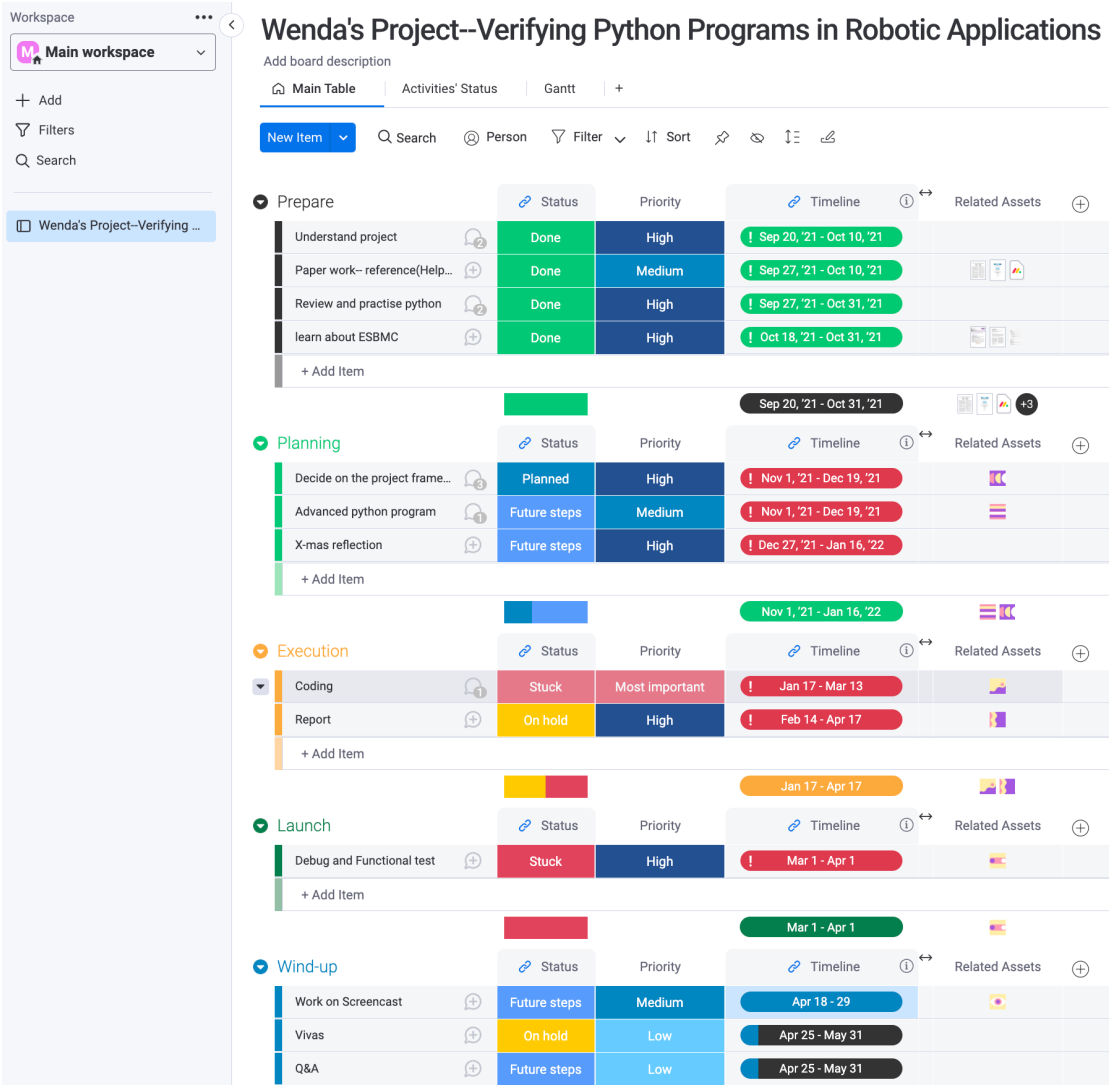

Figure 0. Project Planning Timeline

List of Figures

Figure 0. Project Planning Timeline

Figure 1. Arithmetic logic of SMT

Figure 2. ESBMC architectural overview

Figure 3. The architecture of the project

Figure 4. Python code compiled process

Figure 5. illustrates the AST structure of *func_def*

Figure 6. *ast.NodeVisitor* case study

Figure 7. *ast.NodeTransformer* case study

Figure 8a. AST API test code

Figure 8b. AST API partial test result

Figure 9. Compile C code to GOTO program

Figure 10. compiled variable and type contrast

Figure 11. compiled class contrast

Figure 12. correspondence between python types and c types

Figure 13. compiled defined Extension contrast

Figure 14. compiled declaration syntax contrast

Figure 15. compiled non-Python-aware function contrast

Figure 16. A complete compile process (Python → C)

Figure 17. pseudocode of Floyd-warshall algorithm

Figure 18. Python Floyd Warshall Algorithm compile to ANSI-C code

List of Abbreviations

1. SAT: Boolean Satisfiability
2. SMT: Satisfiability Modulo Theories
3. BMC: Bounded Model Checking
4. BDD: Binary Decision Diagrams
5. ESBMC: Efficient SMT-based Bounded Model Checker
6. CFG: Control-Flow Graph
7. VC: Verification Conditions
8. AST: Application Programming Interface
9. API: Intermediate Representation
10. IR: Intermediate Representation
11. BINT: Boolean Values in C
12. PY SIZE T: Signed Size Values in Python

Contents

Abstract **2**

Acknowledgements **3**

Project Planning **4**

List of Figures **5**

List of Abbreviations **6**

1. Introduction **8**

1.1 Motivation and Related Work 8

1.2 Aims and Objectives 8

1.3 Report Structure 9

1.4 Impact of COVID-19 9

2. Background **10**

2.1 Boolean Satisfiability (SAT) 10

2.2 Satisfiability Modulo Theories (SMT) 10

2.3 Model Checking 11

2.4 Bounded model checking 12

2.5 Efficient SMT-Based Context-Bounded Model Checker 12

2.6 Python program vulnerabilities 14

3. Proposed Method **15**

3.1 Architecture 15

3.2 Front-end 15

3.3 Python Abstract Syntax Tree analysis 16

3.4 Intermediate Representation (GOTO) 20

3.5 Cython 20

4. Evaluation **27**

4.1 Experimental evaluation 27

4.2 Experimental Setup and Benchmarks 28

4.3 Experimental Results 29

5. Conclusion and Future Work **30**

5.1 Conclusion 30

5.2 Reflection 31

5.3 Future work 32

Bibliography **33**

1 Introduction

In this chapter, we will start with the motivation behind choosing this project and introduce the main concepts of Efficient SMT-Based Bounded Model Checking (ESBMC). A problem description of python programs vulnerabilities and the aim and objectives for this project are also provided. This chapter also highlights the impact of COVID-19 as well as the structure of my report.

1.1 Motivation and Problem description

Python has sparked much interest, especially among computer experts, over the last decade, and much python-based software has been developed. At the same time, robotic applications will also continue to be a prominent topic due to rising consumption and industrial demand. Therefore, it is not difficult to imagine a Python framework robotic application that generates possible uncertainty for users because of Python vulnerabilities.

People proposed the model checking approach as a supplement, which can automatically verify whether the software being produced has issues more effectively in all the feasible states since flaws and defects hardly reached in the programme are challenging to identify entirely using standard modelling and testing techniques. We will develop BMCPython, a novel verification approach based on a bounded model checking tool for software written in Python, to find vulnerabilities in Python scripts in this project [22]. When BMCPython identifies a vulnerability in a programme, it creates a counterexample. However, the Benchmark results of BMCPython [28] yield several inaccurate findings, including wrong TRUE and FALSE. As a result, we must validate the findings after running BMCPython, and we may potentially implement some optimisations using ESBMC[20].

1.2 Aims and Objectives

This project will produce BMCPython, which will be based on the Efficient SMT-Based Bounded Model Checker tool (ESBMC). By understanding software model checking, readers will be able to identify Python vulnerabilities[22], assess current verification methodologies, and design appropriate improvements.

The specific aims and objectives are as follows.

- Understand the background and logic of ESBMC
- Develop BMCPython by calling the library of ESBMC
- Determine the ideas and concepts underlying vulnerability discovery of BMCPython in Python bytecode
- Reproduce the extension described in the initial research and ascertain its limits and disadvantages
- Verifying and analysing the BMCPython after the developments

1.3 Report Structure

The chapters of this report cover:

- **Chapter 2: Background** presents the necessary background concepts for comprehending this project and evaluating this report. Contains details about the Satisfiability Modulo Theories (SMT) used by the tool and intermediate representation (IR) supported by ESBMC, as well as an explanation of the technique used for bounded model checking. Additionally, the chapter concludes with a synopsis of the relevant work.
- **Chapter 3: Proposed Method** describes the components of the BMCPython application and details the encodings used to transform restrictions and characteristics, along with the Abstract Syntax Tree (AST) analysis of Python, into the background concepts mentioned in Chapter 2. It also introduces the methods for converting Python to ANSI-C code, also known as the GOTO section.
- **Chapter 4: Evaluation** covers the outcomes of the testing, as well as the benefits and drawbacks of the implementation and probable causes.
- **Chapter 5: Conclusions and Future Work** emphasises the project's accomplishments and limitations, including student reflections on the strategy and critical review of future work.

1.4 Impact of COVID-19

Six semesters comprised my undergraduate courses, and the COVID-19 outbreak lasted five semesters. It was quite frustrating. Spend more than half of my undergraduate time studying online, making it difficult to have a good experience. When it comes to selecting my final year project, we can just browse our supervisor's project profile and communicate with them via email. If we teach offline, we could discuss it in advance with our tutor. For instance, my project involves robotic applications, and I am only limited by my creativity. Online tutorial sessions and resource searches are insufficient. The robotic application attracted me to this project. I could have consulted the technical advisor about robotic applications and visiting the robot prototypes at our department. Due to lack of knowledge on robotic applications, I was unable to improve on this section. This is also my biggest regret regarding my project. Additionally, I was vaccinated against COVID-19 during the first semester of my final year. Due to my preceding pneumothorax surgery on my lungs, I suffered a serious adverse reaction that lasted more than 10 days. With the government's COVID-19 restriction removed, in early March, for more than two years it eventually came to me, I affected the virus. I had a slight fever, felt exhausted, was concerned about my project, and was just miserable. After I recovered, I spent the Easter vacation concentrating on completing the project report. When the epidemic spread to Shanghai, my entire family live there, experiencing another round of lockdown with scarce supplies and even no fresh veggies or food to eat. I had serious mental health problems recent days because of what my family is suffering due to the local government's lack of control. I wrote this report in a depressed state, which have impacted the quality of my report to a certain extent.

2 Background

2.1 Boolean Satisfiability (SAT)

The Boolean satisfiability problem[14] (sometimes referred to as the propositional satisfiability problem) is a problem in logic and computer science that involves assessing if an explanation exists that satisfies a particular Boolean formula[14]. In other words, it determines whether a variable specified by a Boolean formula can be consistently substituted with a TRUE or FALSE value such that the formula evaluates to TRUE. If this is TRUE, the formula is satisfiable. If such assignments do not exist, the function described by the formula is FALSE for all conceivable variable terms, rendering the formula unsatisfiable.

2.2 Satisfiability Modulo Theories (SMT)

SMT[12, 19] solver has established itself as the primary engine in the fields of software engineering, programming languages, and information security; its application scenarios are numerous. Because this project is based on the ESBMC library call and SMT[18] solvers available in ESBMC verification, we will primarily introduce and demonstrate SMT's software analysis and verification functions. Software deductive verification reduces to solving the implication problem of two logical formulae, which can then be expressed as the satisfaction problem of SMT.

More SAT solvers have been used for formal hardware and software verification in recent decades. Determining the satisfiability of logical expressions, first order, has been instrumental in determining the correctness of a system. Nonetheless, many applications need to determine satisfiability that is not first-order but defined in some grounding theory, the interpretation of which uses certain symbols of predicates and functions. For example, formula 2.1 uses integer arithmetic symbols, which cannot be interpreted alone by propositional logic[1, 14].

$$F :: = y \cup z \wedge (y+6) < z \wedge \neg (z < y+6) \quad (2.1)$$

SMT is a subset of the problems of formula satisfiability determination. This sort of formula possesses the following two characteristics: the propositional logic formula contains several first-order logical formulations[3]; possesses an arbitrarily complex Boolean structure. All are familiar with propositional logic formulations, such as:

$$P \wedge Q \ \wedge (R \rightarrow S \vee \neg T) \quad (2.2)$$

That is, variables, negatives, and logical conjunctions are all permitted. And the formula for SMT is as follows:

$$g(a) = c \wedge \big(f\big(g(a)\big) \neq \ f(c) \vee g(a) = d\big) \wedge c \neq d \quad (2.3)$$

In comparison to the propositional logic formula, there are a few additional features, mainly the two given at the outset: To begin, there are more non-logical symbols, such as functions g(a),f(g(a)), and constants c, d. [13]Only the characters of P, Q, R, $\vee$ and

→ instead of the signs of logic, constitute the contents of first-order logic in propositional logic. Second, the Boolean structure has been enlarged or become a macroscopic Boolean structure. For instance, $g(a) = c$, $f\big(g(a)\big) \neq f(c)$ this equation substitutes the original Boolean expression definition of the atom. The second point above, extended Boolean structures, becomes apparent with this example. Again, we employ formula 2.3 to make a first-order substitution[9]:

$$g(a) = c \mapsto P_1,\ f\big(g(a)\big) = f(c)\ \mapsto P_2,\ g(a) = d \mapsto P_3,\ c = d \mapsto P_4$$

So, the formula 2.3 takes on the following form:

$$B(F): P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

This is the most often used Boolean expression. As a result, the SMT formula has an arbitrary abstract Boolean structure and is only concerned with the logical connective. B(F) created in this manner is an abstraction of F and can also be thought of as a macro-Boolean structure.

Figure 1 illustrates the typical notation for explaining the SMT syntax. In this notation, F symbolises a Boolean expression, T specifies terms composed of integers, reals, and bit vectors, and op denotes binary operators. [30]Conjunction (∧), disjunction (∨), exclusive-or (⊕), implication (⇒), and equivalence (⇔) are the annotated logical connectives. The relational and nonlinear operators (∗, /, rem) analyse arguments from arrays of bits, integers, and reals. In bit vector manipulation, the shift operators (« and »), and (&), or (|), or exclusive (⊕), complement (∼), concatenation (@), Bitstract(T, i) FullSignExt(T,k), and UnsignExt(T, i) are utilised. It evaluates a Boolean formula F, picking the first argument if the formula is true and setting the second argument if the formula is false. The select operator is used to identify the value stored at the vector's position i. The store operator replaces the value at class i with the new value v contained within the vector.

F ::= *F con F* |¬*F* |*A*
con ::= ∧ | ∨ | ⊕ | → | ⇒
A ::= *T rel T*|*Id*|*true*|*false*
Rel ::= < | ≤ | ≥ | > | = |≠
T ::= *TopT*| ∼*T*|*ite(F,T,T)*|*Const*|*Id*|
Bitstract(T,i,j)|*FullSignExt(T,k)*|*UnsignExt(T,k)*
Op ::= + | − | ∗ |/|*rem*| << | >> |&| | | ⊕ | @

Figure 1. Arithmetic logic of SMT

2.3 Model Checking

The most extensively used approach for automated formal verification of finite-state transition systems is model checking[27]. It entails modelling the intended design as a finite state machine and defining temporal logic characteristics[5] to formalise the specification. The correctness[7, 8] property can be guaranteed in theory by testing exhaustively inside all attainable configurations. A counterexample[2] will be produced if the condition does not match and is connected to a prohibited state.

Algorithms are used in model checking to ensure that the model is valid. Program models are composed of states, transitions, and a specification or attribute that is logically defined. Stack and heap settings are also assessed in a state, as is the programme counter. The way a programme moves from one state to the next is described in terms of transitions. [26]Checking all potential states in a programme is done using model checking approaches. This technique has a built-in assurance that it will terminate if the state space is finite. In the event that a state is detected that breaks a soundness property, an execution trace identifying the fault is created as a counterexample (to demonstrate the error). Model Checking approaches are used to verify properties that are only partially stated, such as security or liveliness. The deficiency of vulnerable positions such as assertion violations, null pointer dereferences or buffer overflow, or API[25] utilisation contracts, including the sequence of function calls, is described by safety characteristics. It's a sign that something beneficial will ultimately happen, such as the requirement for requests to be serviced or for a programme to end.

2.4 Bounded model checking

Bound Model Checking (BMC) was designed as a technique for substituting symbolic analysis methods for the usage of Binary Decision Diagrams (BDD) in symbolic model checking. BMC is a notion used in computer science. It is based on the idea of traversing a finite prefix of states constrained by some constraint k, such that there may exist a trace that fails the critical condition. [25]The presence of a back loop from the most recent state of the prefix to any of its initial conditions leads to an endless route. To solve the original planning issue and the bound, the k-bounded model is translated into a polynomial SAT[6] or SMT instance using BMC.

Bound Model Checking also considers liveness and nested temporal features, in contrast to deterministic planning, which only considers essential safety aspects (i.e., if and how the desired state can be attained). To conclude, BMC verifies the satisfiability of verification criteria, which is the outcome of the issue translated into a syntax that the verifier accepts. It is called bounded because it examines only states that may be reached in a limited number of steps, for example, k. Unwinding the model under verification k times and associating it with a property generates a propositional formula presented to a SAT solver[10]. The procedure is fulfilled if and only if there exists a sign of length k that contradicts the property. BMC has detected several flaws that would have received little attention otherwise.

2.5 Efficient SMT-Based Context-Bounded Model Checker

ESBMC is a context-bounded model checker for incorporated ANSI-C[11] system that utilises SMT solvers to verify single- and multi-threaded software with private objects and keys. ESBMC is entirely compatible with ANSI-C and capable of ascertaining programmes that implement bit-level data structures, arrays, pointers, structs, unions,

memory allocation, and fixed-point arithmetic. It is possible to carry out the validation of single or multi-threaded programs and check for deadlocks, arithmetic overflow, division by zero, vector limits and other types of violations.

The examined programme is built in ESBMC as a state transition system M = (S, R, s_0), which is retrieved from the control-flow graph (CFG). S denotes the set of states, $R \subseteq S \times S$ denotes the set of transitions (i.e., pairs of states indicating how the system might transition from one state to another), and $s_0 \subseteq S$ is the set of beginning states. A state $s \in S$ is made up of the quality of the programme counter *pc* and the importance of all programme variables combined into a single object. A CFG's first programme location is assigned to *pc* by the starting state s0 of the CFG. A logical formula $r = (s_i, s_{i+1}) \in R$ identifies each transition from one state to another in the *R*-state between the two states s_i and s_{i+1}. This logical formula contains the restrictions placed on the values obtained from the programme counter and the corresponding outputs of the programme variables.

Additionally, ESBMC enables users to expand this method to property violation verification to complicated programmes with several iterations. The architecture of the ESBMC[20, 18] is depicted in Figure 2.

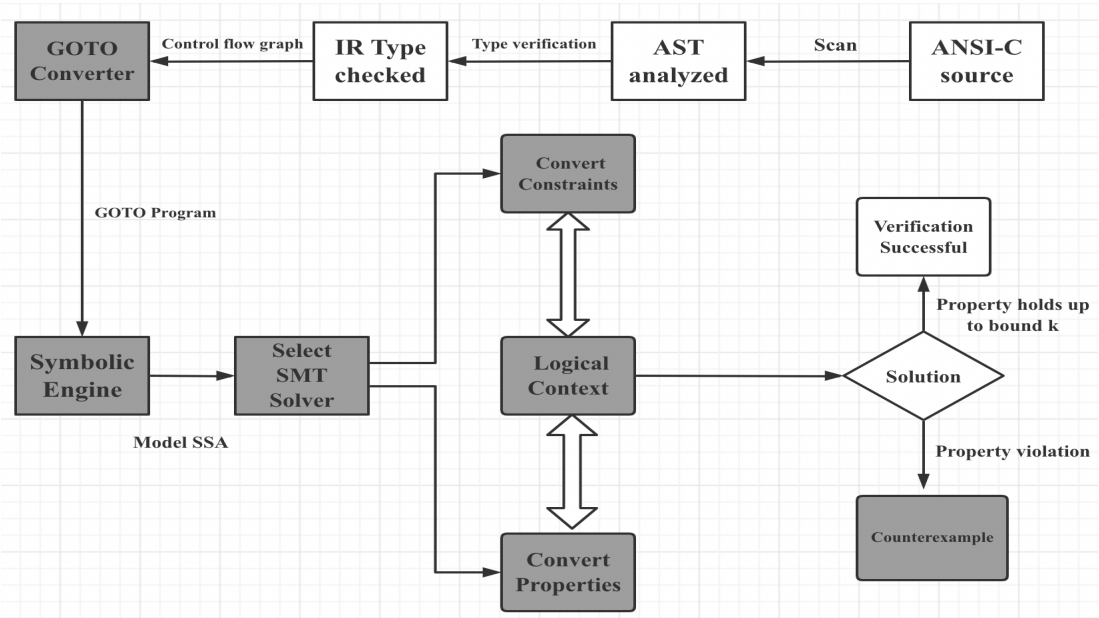

Figure 2. ESBMC architectural overview

This technique enables the generation of verification conditions (VC) for checking arithmetic overflow, doing an analysis of a program's CFG, determining the optimal solution for a given situation, and simplifying the unfolding of a formula. In brief, ESBMC turns an ANSI-C/C++ programme into a GOTO programme[14]; that is, it changes switch and while expressions to GOTO statements, which are subsequently symbolically replicated by the GOTO symbol. Following that, an SSA model is built with static values given to the properties, which can be checked using an appropriate SMT solver. If there is a violation in the property, the counterexample is interpreted

and the fault discovered is reported; otherwise, the property fulfils the k iteration limit capacity.

The automated nature of the ESBMC verification method makes it perfect for fast real-time embedded software testing[18]. For the purposes of this project, the capabilities of the ESBMC tool are generalised to the verification of Python applications. This is the first attempt in the existing literature to utilise ESBMC to verify Python programmes.

2.6 Python programs vulnerabilities

When developing an application or coding, it is possible to make errors or introduce loopholes. These faults result in weaknesses, also known as vulnerabilities. When exploited, these defects may be highly damaging for enterprises since they endanger the security and availability of data stored in the system. The nature of the coding fault determines the severity of the vulnerability. Developers may provide user input directly into the system command, for example. If this is the case, an attacker may be able to gain complete control of the server through remote code execution. As a result, it's critical to know how these flaws arise and to prevent making mistakes that might lead to exploitation. Almost all of Python's problems stem from faulty input validation, which allows users to exploit security weaknesses by inserting arbitrary data. The most frequent Python security flaws will be examined now.

- Python code injection[22] and arbitrary command execution: Server-side code injection vulnerabilities occur when a code interpreter dynamically analyses a string containing user-controllable data. An attacker can modify the code that will be executed and insert arbitrary code that will be executed by the server using manipulated input. Vulnerabilities in server-side code injection can reveal a program's capabilities as well as the server on which it is running. The server might be exploited as a jumping-off point for attacks on other systems. The standard Python function, which is in charge of executing commands on the system, takes direct user input. As a result, the attacker has the ability to seize control of the target system.
- Directory traversal tools in Python: This well-known vulnerability also arises due to improper sanitisation of user inputs when a file is viewed. This approach enables an attacker to upload files to the server. This might result in sophisticated data leaking and remote code execution.
- Outdated Dependencies (e.g., Unnecessary, old, and overlapping transitive dependencies): In other words, dependencies are the fundamental aspects upon which all of these different components are constructed. While modules are being created, unintentional vulnerabilities[22] may arise. As a result, programmers update these dependencies regularly to address these vulnerabilities. When a developer continues to use an out-of-date dependency with a vulnerability, the programme becomes vulnerable.

- Flawed logic in Python Assert Statement: Strong assertions make a claim or represent a fact in a programme. For instance, when a division function is written, it asserts that the divisor cannot be 0. Assertion is built into Python. Assertions, such as Boolean expressions, are used to assess the condition. Execution advances to the next line if the target is satisfied. Otherwise, an error message will appear.

3 Proposed Method

This chapter discusses the project's general architecture, beginning with the front-end framework and progressing to an introduction to Python Abstract Syntax Tree (AST) analysis, as well as a detailed explanation of the method used in the Intermediate Representation process, also known as the GOTO part, which describes how to convert Python to ANSI-C code. We compile into the anticipated ANSI-C code using an existing Python optimised execution library (CPython) and are compatible with ESBMC.

3.1 Architecture

The process of designing and implementing this project is depicted succinctly in Figure 3. The project's base is built on the Python flask frontend; benefit from this interface, python scripts control and execute the process of AST analysis are efficient and transparent.

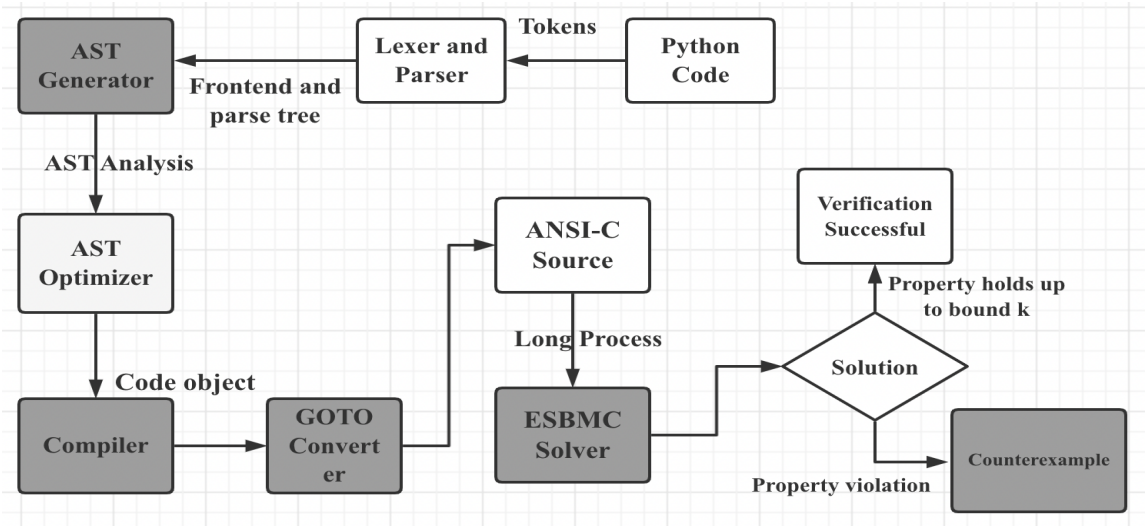

Figure 3. The architecture of the project

3.2 Front-end

The Python parser (front end) allows users to construct their unique Python compilers, analysis tools, and source transformation tools. This project is mainly related to the Python flask front-end analysis tools. PyFlask[21] is a Python web framework with a small core and a strong emphasis on extensibility. Unlike Django's batteries-included approach, flask does not contain serialisers, user administration, or built-in internationalisation. Other flask extensions provide these and many more features,

forming a diverse ecosystem. Flask is very straightforward to learn as a newbie due to the lack of boilerplate code required to get a small app setting and running.

3.3 Python Abstract Syntax Tree analysis

Abstract Syntax Tree (Ast) is an intermediate for Python that sits between source code and bytecode. The Ast module enables users to perform a syntax tree analysis on the source code. Additionally, programmers can modify and execute the syntax tree, as well as unparsed the syntax tree generated by Source into Python Source. As a result, the AST provides ample opportunity for Python source examination, parsing, code modification, and debugging. The AST and overlay types that users want to utilise for type inference come from two sources: operations-specific types and user-supplied types. The conventional method of constraint-based unification will be used to recreate the styles. To address the massive constraint problem that arises when given a collection of input arguments of varying types, we'll traverse our AST and generate a constraint set of equality relations between types (expressed as a b). If we do not know the kind of an expression, we shall use free type variables. Several outcomes could occur, including the types that have been accurately identified; we don't know what the classes are yet, polymorphism abounds in the styles, and there is a wide range of varieties. When creating a polymorphic function, it still has free type variables at the top-level type. For instance, we might have a subcategory like:

```
1. [Array a, Array a] -> a
```

This means that the reasoning is not limited to a particular array element type and may be applied to any element type regardless of its nature. When we suppose that our compiler understands how to decrease *a*, we receive a whole family of functions. This is advantageous for code reuse since we now have access to an entire family of functions. The varieties are not entirely determined. This means that the constraints imposed by usage are excessively permissive in determining the entirety of each subexpression's meaning. In this case, an explicit annotation is necessary. The typeface has a confused look. This will occur if no solution exists that satisfies all of the constraints.

When the AST concept is applied, it is necessary to introduce the Control-Flow Graph (CFG) Generator. When it comes to constructing GOTO programmes, the CFG generator is in charge. These are derived from the input programmes' abstract syntax trees (AST) and are a condensed form of the source code, consisting of conditional and unconditional branches, assertions, assignments, and assumptions, as well as conditional and unconditional branching. Technically, it eliminates all loops (for, while, and do-while) and switches expressions, which is a significant improvement.
The CPython interpreter officially provided by Python processes the python source code as follows:

- Create a parse tree from the source code
- Convert the parse tree to an Abstract Syntax Tree (AST)
- Convert AST to a Control Flow Graph (CFG)

- Produce bytecode from the Control Flow Graph

Precisely, Figure 4 illustrates the procedure of processing the original Python code.

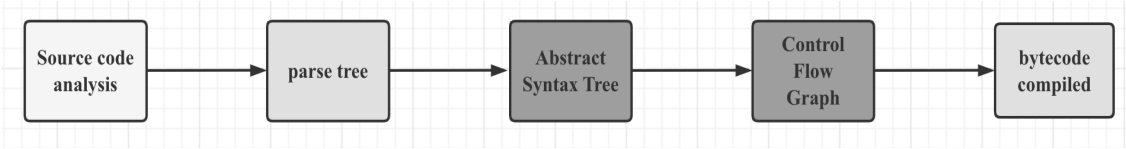

Figure 4. Python code compiled process

Python source code is parsed into two trees: a syntax tree and an abstract syntax tree. The abstract syntax tree illustrates the Python source file's syntactic structure. While users probably don't need an abstract syntax tree to programme most of the time, the AST offers certain unique advantages under specific circumstances and conditions. This section describes in detail the creation and traversal of the AST in order to properly apply AST analysis methods in this project. The section that creates the AST is divided into two sections: compile function and generate the AST:

- Compile Function in Python

compile(source, filename, mode[, flags[, dont_inherit]])

Source -- String or AST (Abstract Syntax Trees) object. Users can normally pass in the entire contents of the *py* file through *file. read()*
Filename -- The filename of the source code or a recognised value if the source code cannot be read from the file
Mode -- Defines the type of code that will be compiled. It can be chosen from one of the following: *exec, eval,* or *single*
Flags -- A variable scope, local namespace, or any mapping object, if specified
Flags and *dont_inherit* are indicators used to govern how source code is compiled

An AST compile example is shown below. And compile and execute this code fragment using *compile* function.

```
2.  func_def = \
3.  """
4.  def add(x, y):
5.  return x + y
6.  print add(3, 5)
7.  """
8.  >>> cm = compile(func_def, '<string>', 'exec')
9.  >>> exec cm
10. >>> 8
```

The above *func_def* is compiled to get bytecode via *compile,* and *cm* is the code object.

- AST generation

Use the *func_def* above to generate the AST.

```
r_node = ast.parse(func_def)
print astunparse.dump(r_node)
print ast.dump(r_node)
```

```
Module(body=[
    FunctionDef(
        name='add',
        args=arguments(
            args=[Name(id='x',ctx=Param()),Name(id='y',ctx=Param())],
            vararg=None,
            kwarg=None,
            defaults=[]),
        body=[Return(value=BinOp(
            left=Name(id='x',ctx=Load()),
            op=Add(),
            right=Name(id='y',ctx=Load())))],
        decorator_list=[]),
    Print(
        dest=None,
        values=[Call(
                func=Name(id='add',ctx=Load()),
                args=[Num(n=3),Num(n=5)],
                keywords=[],
                starargs=None,
                kwargs=None)],
        nl=True)
  ])
```

Figure 5. illustrates the AST structure of *func_def*:

- Traversal of Python AST

Python provides two solutions to traverse the entire Abstract Syntax Tree. The first one is *ast.NodeVisitor* which is used primarily to change the AST structure by modifying nodes in the syntax tree. The other is *ast.NodeTransformer,* it achieves this mainly by replacing nodes in the AST. Both approaches are applied to the AST analyser for this project.

--*ast.NodeVisitor:* Change the addition operation in the *add()* function in *func_def* to subtraction and add the call log for the function implementation.

```
class CodeVisitor(ast.NodeVisitor):
    def visit_BinOp(self, node):
        if isinstance(node.op, ast.Add):
            node.op = ast.Sub()
        self.generic_visit(node)

    def visit_FunctionDef(self, node):
        print 'Function Name:%s'% node.name
        self.generic_visit(node)
        func_log_stmt = ast.Print(
            dest = None,
            values = [ast.Str(s = 'calling func: %s' % node.name, lineno = 0, col_offset = 0)],
            nl = True,
            lineno = 0,
            col_offset = 0,
        )
        node.body.insert(0, func_log_stmt)

r_node = ast.parse(func_def)
visitor = CodeVisitor()
visitor.visit(r_node)
# print astunparse.dump(r_node)
print astunparse.unparse(r_node)
exec compile(r_node, '<string>', 'exec')

#Output
Function Name:add
def add(x, y):
    print 'calling func: add'
    return (x - y)
print add(3, 5)
calling func: add
-2
```

Figure 6. *ast.NodeVisitor* case study

-- *ast.NodeTransformer:* After changing the *add*() function specified in *func_def* to a subtraction function, it is preferred to be thorough and modify the function name and parameters, as well as the function called in the AST, and complicate the additional function call log.

```
01. 1 class CodeTransformer(ast.NodeTransformer):
02.  2     def visit_BinOp(self, node):
03.  3         if isinstance(node.op, ast.Add):
04.  4             node.op = ast.Sub()
05.  5         self.generic_visit(node)
06.  6         return node
07.  7
08.  8     def visit_FunctionDef(self, node):
09.  9         self.generic_visit(node)
10. 10         if node.name == 'add':
11. 11             node.name = 'sub'
12. 12         args_num = len(node.args.args)
13. 13         args = tuple([arg.id for arg in node.args.args])
14. 14         func_log_stmt = ''.join(["print 'calling func: %s', " % node.name, "'args:'", ", %s" * args_num % args]
    )
15. 15         node.body.insert(0, ast.parse(func_log_stmt))
16. 16         return node
17. 17
18. 18     def visit_Name(self, node):
19. 19         replace = {'add': 'sub', 'x': 'a', 'y': 'b'}
20. 20         re_id = replace.get(node.id, None)
21. 21         node.id = re_id or node.id22         self.generic_visit(node)
22. 23         return node
23. 24
24. 25 r_node = ast.parse(func_def)
25. 26 transformer = CodeTransformer()
26. 27 r_node = transformer.visit(r_node)
27. 28 # print astunparse.dump(r_node)
28. 29 source = astunparse.unparse(r_node)
29. 30 print source
30. 31 # exec compile(r_node, '<string>', 'exec')
31. 32 exec compile(source, '<string>', 'exec')
32. 33 exec compile(ast.parse(source), '<string>', 'exec')
33.
34. #output
35. def sub(a, b):
36.     print 'calling func: sub', 'args:', a, b
37.     return (a - b)
38. print sub(3, 5)
39. calling func: sub args: 3 5
40. -2
41. calling func: sub args: 3 5
42. -2
```

Figure 7. *ast.NodeTransformer* case study

The divergence is readily apparent in the code shown in Figure 6 and Figure 7. As a source code analysis tool, AST is effective. Checking syntax, debugging faults, detecting particular fields, etc. While the above is a technique for debugging Python source code by including call log information in a function, in practice, we parse the complete Python file to loop over the updated source code.

We designed a rather extensive AST analysis Application Programming Interface (API) based on the flask framework through Python flask front-end learning and AST parser demonstration imitation. Figure 8a and 8b illustrate the project's AST parser's original test code and AST structure findings, respectively.

Figure 8a. AST API test code

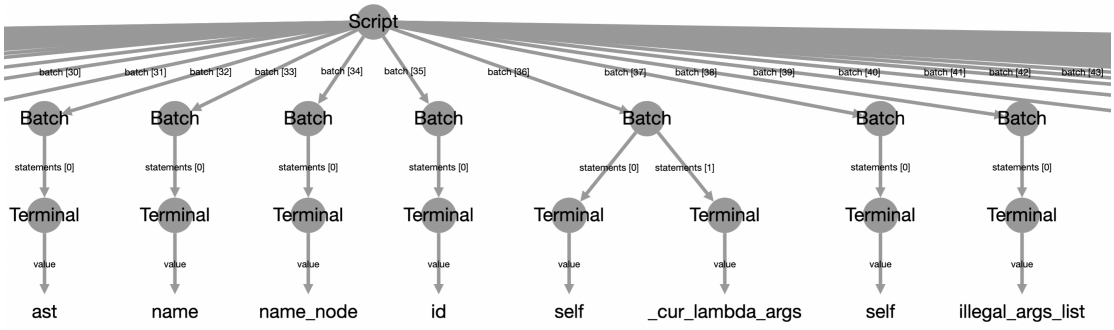

Figure 8b. AST API partial test result

3.4 Intermediate Representation (GOTO)

This section illustrates how to transform code into quantifier-free equations and representations of various portions to increase one's understanding of ESBMC capabilities. As described in the Project design, Figure 9 illustrates the translation of a C programme to a GOTO programme[23], provides guidance for subsequent python GOTO programming and has a basic understanding of GOTO.

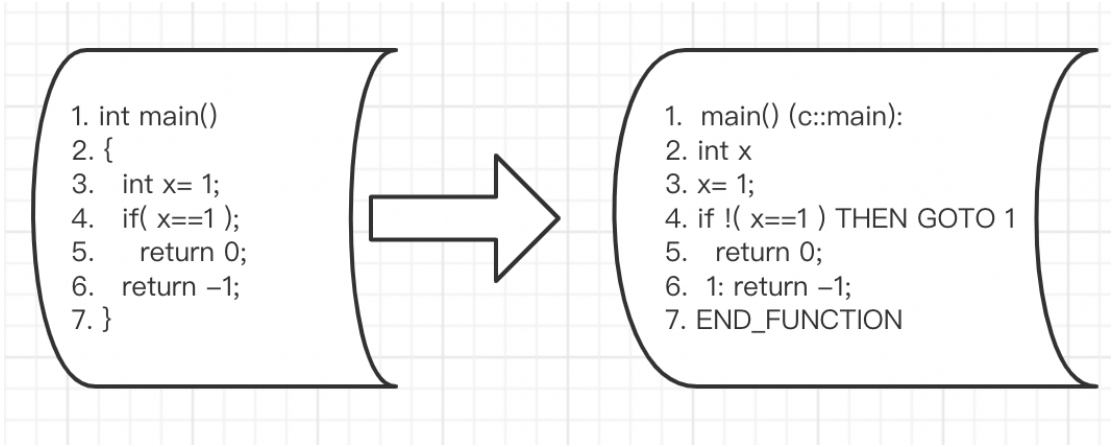

Figure 9. Compile C code to GOTO program

3.5 Cython

Cython's *cdef* syntax was created to make type descriptions straightforward and understandable from a C/C++ perspective[15]. Due to the PEP 484[22] type hints and PEP 526[31] variable annotations in pure Python syntax, static Cython type declarations are permitted in pure Python programmes. Importing a custom Cython module into the Python module user is creating enables the use of C data types. e.g.

```
1. import cython
```

Python programming encourages users to consider about classes and objects in terms of their methods and properties rather than their position in the class structure. The 'red

tape' of data type handling is delegated to the interpreter, which may result in Python being a more relaxed and pleasant language for rapid development.

Throughout execution, searching for namespaces, obtaining attributes, and decoding argument and adjective tuples all consume considerable time. In comparison to other "early coupling" languages such as C, Python has a 'late binding' difficulty during runtime. However, with Cython[33], 'early binding' programming techniques can result in significant speedups. If users wish to improve the performance of their code, users can use static typing for arguments and variables, although this is not required. When and if necessary, do optimisations. If users are unable to optimise their code due to typing and Cython still needs to verify that an object's type matches the provided one, typing may actually slow down their application.

The following strategies can be used to define C variables and types. Annotations can be typed in PEP-484/526 or cython.declare, or they can be operated using the Cython-specific cdef statement. Declare() and the cdef statement can be used to declare variables and attributes at the function and module levels. However, type annotations have no effect on variables or attributes at the module level. Instead of generating module-specific C variables, Cython[33] is prevented from doing so by type annotations, and therefore variables are saved in the package dict (as Python values). Declare() can be used explicitly to declare C variables in the Python program.
All global variables that have been declared in the C language are instantly reset to either 0, NULL or nil. For a local variable, as is well known in Python and C, merely stating it is not sufficient to initialise it. If declaring a local variable but does not assign any data to it, the developer will get an error from both Cython and the C compilers. Before users can use the variable, a value must be given to it. In most circumstances, users may do this directly in the declaration. Figure 10 gives an example of this variable type compiled.

```
a_global_variable = declare(cython.int, 42)

def func():
    i: cython.int = 10
    f: cython.float = 2.5
    g: cython.int[4] = [1, 2, 3, 4]
    h: cython.p_float = cython.address(f)
```

```
cdef int a_global_variable

def func():
    cdef int i = 10, j, k
    cdef float f = 2.5
    # cdef float g[4] = [1,2,3,4]  # currently not supported
    cdef float *g = [1, 2, 3, 4]
    cdef float *h = &f
```

Figure 10. compiled variable and type contrast

Implication types can be provided for classes:

```
01.  from __future__ import print_function
02.
03.  @cython.cclass
04.  class Shrubbery:
05.      width: cython.int
06.      height: cython.int
07.
08.      def __init__(self, w, h):
09.          self.width = w
10.          self.height = h
11.
12.      def describe(self):
13.          print("This shrubbery is", self.width,
14.                "by", self.height, "cubits.")
```

```
01.  from __future__ import print_function
02.
03.
04.  cdef class Shrubbery:
05.      cdef int width
06.      cdef int height
07.
08.      def __init__(self, w, h):
09.          self.width = w
10.          self.height = h
11.
12.      def describe(self):
13.          print("This shrubbery is", self.width,
14.                "by", self.height, "cubits.")
```

Figure 11. compiled class contrast

When it comes to the variable types, it is convenient to visualize the type difference between Python and C (see figure 12), after all, around a third of the work in this section is to imagine how covert can be a C type of variable in Python.

Python type(s)	C type(s)
bool	bint
int long	[unsigned] char [unsigned] short [unsigned] int [unsigned] long [unsigned] long long
float	float double long double
complex	float complex double complex
bytes str unicode	char * std::string (C++)
dict	struct

Figure 12. correspondence between python types and c types

The C-based Cython[33] programming language makes use of pointers. All C types, including char, short, int, long, and long-long, are accessible in unsigned variants (cython.uint in Python code). BINT (boolean values in C) and PY SIZE T(signed size values in Python containers) are the two forms of bint. The pointer types in Cython follow C traditions, such as appending a * to the underlying type they refer to, such as int. In Pure Python mode, pointer types are named with "p"s instead of "p"s followed by an underscore (cython.pp int for C int pointer).

Other pointer types, such as cython.pointer(), may be constructed using the cython.pointer() function (cython.int). When stack-allocated arrays are used, the array's size must be established prior to formation. For instance, int[10] Cython does not allow variable-length arrays, which were introduced in C99. Because Python's *x syntax is incorrect, Cython uses x[0] as a pointer. Any Extension Types specified by the user may also benefit from the use of static typing. Consider figure 13:

```
def main():
    foo: list = []
def main():
    bar: (cython.double, cython.int)
```

```
cdef list foo = []

cdef (double, int) bar
```

Figure 13. compiled defined Extension contrast

In Cython, there are two ways to define a function. The def statement is used to describe Python functions, much like in Python. A Python object is sent in as an argument, and a Python object is returned. In Cython syntax, the cdef statement or the @cfunc modifier are used to declare C functions. As input, they accept either Python objects or C values[33] and may output either Python objects or C values.

Only Python functions may be invoked from outside the package by construed Python code, while both Python and C procedures can reach each other freely inside a Cython module. Def must be used to define all the Cython procedures that users would like to export from this module. Users may use the *@ccall* decorator, or the *cpdef* function, to define a combination function.

Following the example below(figure 14), such procedures may be accessed from anywhere; however, when invoked from other Cython programmes, they utilise the quicker C calling convention. Additionally[33], they may be overwritten by a Python function on a subtype of an object feature, even when invoked from Cython. If this occurs, the majority of performance improvements are erased, and even if it does not, there is a negligible expense associated with invoking such a function from Cython as opposed to calling a C method. Using standard C declaration syntax, parameters of any kind of function may be defined to have C data types.

```
def spam(i: cython.int, s: cython.p_char):
    ...

@cython.cfunc
def eggs(l: cython.ulong, f: cython.float) -> cython.int:
    ...
@cython.cfunc
def chips(t: (cython.long, cython.long, cython.double)) -> (cython.int, cython.float):
    ...
def spam(python_i, python_s):
    i: cython.int = python_i
    s: cython.p_char = python_s
    ...
```

```
def spam(int i, char *s):
    ...


cdef int eggs(unsigned long l, float f):
    ...
cdef (int, float) chips((long, long, double) t):
    ...
def spam(python_i, python_s):
    cdef int i = python_i
    cdef char* s = python_s
    ...
```

Figure 14. compiled declaration syntax contrast

At the moment, automatic conversion is only available for numeric types, text types, and structs (constructed recursively of any of these kinds); using any other type as an argument to a Python function would lead to a compile-time exception. Strings must be handled carefully to guarantee that they include a value if the pointer is to be utilised after the call. Types of data may be retrieved using Python mappings, and once again, caution should be used when using string attributes after the method returns.

Because the call is done via a regular C function call, it is possible to provide any kind of argument into a C function. For example, C functions that are defined using Cdef or a @cfunc decorator that does not have an explicitly returned object will return an object return type similar to that of Python. This is an exception to the rule that the lambda function may be left undefined in C/C++. Null is returned for reference types and False for ints; 0 is returned for bints and other non-Python objects, and 0 is returned for all other objects. Python (and specifically the Cython runtime) makes use of specified error return values to communicate exceptions up the callback function and inform the caller when they occur inside a process, as explained here.

Error returns are always the NULL pointer for functions yielding Python objects; hence every function that returns a Python object has an error return value. Unlike code written as C functions or cpdef/@ccall functions, which must always produce a specific C type in case of a failure, functions declared as C functions or cpdef/@ccall functions may return any C type. It is, therefore, possible to write a function that returns instantly without transmitting an exception to its operator if an exception is thrown in such a function.
Exception output values for C functions must be declared as connections with the library if the user wishes such a C function[33] to be capable of propagating

exceptions. Here is an illustration(figure 15).

```
from cython.cimports.libc.stdio import FILE, fopen
from cython.cimports.libc.stdlib import malloc, free
from cython.cimports.cpython.exc import PyErr_SetFromErrnoWithFilenameObject

def open_file():
    p = fopen("spam.txt", "r")   # The type of "p" is "FILE*", as returned by fopen().

    if p is cython.NULL:
        PyErr_SetFromErrnoWithFilenameObject(OSError, "spam.txt")
    ...


def allocating_memory(number=10):
    # Note that the type of the variable "my_array" is automatically inferred from the assignment.

    my_array = cython.cast(p_double, malloc(number * cython.sizeof(double)))
    if not my_array:  # same as 'is NULL' above
        raise MemoryError()
    ...
    free(my_array)
```

```
from libc.stdio cimport FILE, fopen
from libc.stdlib cimport malloc, free
from cpython.exc cimport PyErr_SetFromErrnoWithFilenameObject

def open_file():
    cdef FILE* p
    p = fopen("spam.txt", "r")
    if p is NULL:
        PyErr_SetFromErrnoWithFilenameObject(OSError, "spam.txt")
    ...


def allocating_memory(number=10):
    cdef double *my_array = <double *> malloc(number * sizeof(double))
    if not my_array:  # same as 'is NULL' above
        raise MemoryError()
    ...
    free(my_array)
```

Figure 15. compiled non-Python-aware function contrast

As the name indicates, the execution module includes the implementation of developers' functions, classes, extension types, etc. This module supports almost all of the python syntax. When a.py file is changed to.pyx without altering any code, Cython will preserve the python functionality. This is the case most of the time. It is feasible for Cython[33] to build both .py and .pyx files. Cython doesn't care about the suffix in the filename if you're simply going to use Python syntax; it won't alter the produced code in any way.

The .pyx file is required if one wants to utilise Cython syntax. C values and functions may be declared using Cython syntax (such as cdef) as well as Python syntax (such as cimport), both of which can be used to import C definitions. This article and the remainder of the Cython[33] instructions cover a plethora of additional functional Cython capabilities that may be included in the implementation code. The construction of certain Extension Types is constrained by those defined in the associated definition file.

Characterization files are used to specify a variety of different things. It is possible to make any C declaration, and it is also possible to declare a C object or function that has been performed in a C file. This may be accomplished using the command cdef extern from. Cython can comprehend a .c files since they are used to translate C data types

into a syntax that Cython[33] can recognize then enables the C variables and functions to be used directly in execution files, eliminating the need for a separate import statement(. cimport). Read can read more about it in Interfacing with External C Code and Using C++ in Cython.

Also included are the definitions for an additional category and the assertions of procedures for an auxiliary package. It cannot include any C or Python procedure definitions, nor can it contain any Python class declarations, nor can it contain any interactive representations. It is required when attempting to access cdef properties and methods, as well as when attempting to derive from cdef classes specified in this module. Ultimately, we wanted to create a highly modular system that could conduct simulation, data processing, and visualisation—and that could often accomplish all of these activities at the same time.

There is a temptation to achieve this by creating a streamlined atomic library (for example, by utilising a well-structured C++ set of categories), which is not ideal. This seems to be, in our perspective, too formal and limiting. We intended to make it possible to support modules that were only weakly tied to one another. For instance, there is no need for a graphics library to rely on the same concepts as a simulation code or even that they are written in the same language in order to function correctly. A similar approach was used with third-party packages, where no hypotheses could be formed about the underlying structure of such libraries.

With efficient analysis and utilization of Cython[33], the core aspects of our proposed methods can be effectively implemented. To conclude this chapter, two code snippets(figure 16) illustrate the outcomes of Cython. This is a commendable and accomplished process of converting Python source to ANSI-C code.

```
01.  # A function written in Python
02.  from SPaSM import *
03.  def run(nsteps, Dt, freq):
04.      for i in xrange(0, nsteps):
05.           integrate_adv_coord(Dt)
06.           boundary_periodic()
07.           redistribute()
08.           force_eam()
09.           integrate_adv_velocity(Dt)
10.           if (i % freq) == 0) :
11.              output_particles('Dat'+str(i))
```

```
/* A simple function written in C */
#include "SPaSM.h"
void run(int nsteps, double Dt, int freq) {
    int i;
    char filename[64];
    for (i = 0; i < nsteps; i++) {
        integrate_adv_coord(Dt);
        boundary_periodic();
        redistribute();
        force_eam();
        integrate_adv_velocity(Dt);
        if ((i % freq) == 0) {
            sprintf(filename,"Dat%d",i);
            output_particles(filename);
        }
    }
}
```

Figure 16. A complete compile process (Python → C)

4 Evaluation

This chapter presents how BMCPython was tested in order to prove the absence of memory safety and undefined behaviour issues in Python programs. Moreover, an evaluation of the BMCPython veracity of generating GOTO programs as well as generating ANSI-C is presented to demonstrate the efficiency and effectiveness of this verification technique.

4.1 Experimental evaluation

In this section, we focus on the evaluation of the IR program. The evaluation approach has been discussed particularly in sections 3.4&5 along with some compiled C code snippets. To further demonstrate the reliability of the compiler, we chose a representative mathematical algorithm(Floyd Warshall Algorithm) to make Cython convert according to its Python source code, see figure 16.

Floyd-warshall algorithm[16] is an algorithm to solve the shortest path between any two points. It can correctly deal with the shortest path problem of a directed graph or negative weight and is also used to calculate the transitive closure of a directed graph. The time complexity of the Floyd-Warshall algorithm is $O(N^3)$ and the space complexity is $O(N^2)$. Floyd's algorithm is a classic example of dynamic programming. In brief, our initial objective is to determine the shortest path between points *I* and *J*.

We provide a more professional interpretation from the perspective of dynamic programming. There are only two possible paths from each node I to any node J: one straight from I to J, and another via numerous nodes K to J. Assume that Dis(I, J) is the shortest path between nodes U and V. We verify whether *Dis(I, K) + Dis(K, J) < Dis(I, J)* is true for each node K. If this is the case, it is demonstrated that the journey from I to k and then to j is shorter than the straight path from I to J. We define *Dis(I, J)* as *Dis(I, K) + Dis(K, J)*, such that after all nodes k are traversed, *Dis(I, K)* is the shortest path between I and j. It is necessary to briefly introduce this algorithm in the

experimental evaluation section, only in this way, the reader can better evaluate the accuracy of the compiled c code. The following figure(figure 1888) shows the logical structure of the Floyd-warshall algorithm in pseudocode.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

Figure 17. pseudocode of Floyd-warshall algorithm

```python
# Python Program for Floyd Warshall Algorithm
V = 4
INF = 99999
def floydWarshall(graph):
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    for k in range(V):
        for i in range(V):
            for j in range(V):
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j]
                                                    )
    printSolution(dist)
def printSolution(dist):
    print ("Following matrix shows the shortest distances\
between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print ("%7s" % ("INF"),end=" ")
            else:
                print ("%7d\t" % (dist[i][j]),end=' ')
            if j == V-1:
                print ()
graph = [[0, 5, INF, 10],
         [INF, 0, 3, INF],
         [INF, INF, 0, 1],
         [INF, INF, INF, 0]
         ]
floydWarshall(graph)
```

```c
// C Program for Floyd Warshall Algorithm
#include<stdio.h>
void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    printSolution(dist);
}
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

int main()
{

    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };
    floydWarshall(graph);
    return 0;
}
```

Figure 18. Python Floyd Warshall Algorithm compile to ANSI-C code

By observing the compiled C code, we can see that functions are correctly replaced, variable names are preserved, variable types have a good mapping according to Figure 11, and the overall algorithmic logic structure is preserved.

4.2 Experimental Setup and Benchmarks

Platform – Laptop, MacBook Pro 2021, 16-inch.

All the experiments were conducted on a 3 GHz Apple M1 Pro processor with 16GB of RAM, running macOS Monterey 12.1 with 10 cores.

For each experiment, the execution time, which is the average of three executions, is measured in seconds based on CPU time.

The maximum execution time for ESBMC and BMCPython is two hours and three hours respectively.

As for software, ESBMC v6.7 is selected as verification tool and boolector is chosen as SMT solver.

Compiler: Visual Studio Code Version 1.65, with Python 3.8 for most of coding work, Cython vision 3.0.

For AST master implementation, additional library is required:
gunicorn~=19.7.1
gevent~=1.2.1
Flask~=1.0.2
whitenoise~=4.1
PyYAML~=5.1

antlr4-python3-runtime~=4.7.2
antlr-ast~=0.8.1

shellwhat==1.2.0
antlr-plsql==0.9.1
antlr-tsql==0.12.6

4.3 Experimental Results

In this section we will verify BMCPython is implemented correctly by testing it on some of the test packages from the SV-benchmarks GitLab repository and we create some tests according to the benchmarks of BMCLua, those algorithm functions are specially developed in python. We will also compare the speed of the original ESBMC and BMCPython.

Firstly, we compared the speed of the implementation of BMCPython on 3 different algorithms taken example by BMCLua benchmarks. The results are as follows:

BMCPython Input	Total Lines of Python	Loop iterations	Total Processing Time (seconds)
Sum of Prime number	7	1000	1.2
	7	1000000	127
Fibonacci	14	5001	6.3
	14	50001	49
Bellman-Ford	31	6	1.3
	31	21	1.9

Unsurprisingly, for these tasks, both versions achieve the same perfect result. This result suggests to us that the implementation of BMCPython is correct. We now will compare the speed of BMCPython and ESBMC on a variety of problems from SV-Benchmarks.

Test file name	ESBMC	BMCPython
sum01-1.c/.py	0.51s	2.21s
sum01-2.c/ .py	1.07s	3.32s
matrix-1.c/ .py	0.10s	1.54s
sum-array-2.c/ .py	1.02s	3.63s

From these results, we can observe that BMCPython spent more time verifying a Python program over the original ESBMC. This difference indicates BMCPython is more complex than ESBMC. It needs to analyze the Python AST and covert the source code to ANSI-C code, which can be checked via ESBMC.

5 Conclusion and Future Work

5.1 Conclusion

This report discussed bounded model checking and how it can be used to verify Python programmes. It commenced by conducting a literature review of the fundamental logic algorithm used in software verification. It provided an intensive explanation of how a bounded model checking tool works. Bounded model checking and ESBMC's specifications were then discussed in detail, with an ESBMC structure diagram provided as a demonstration. A significant challenge in this research was comprehending the current ESBMC and intermediate representation code compile. This subject required a tremendous amount of work, but the results were well worth the effort. It was assessed on standard tests for Efficient SMT-Based bounded model checking tools after the development of this GOTO programme, and an illustration of how it operates was presented in chapter 3. The main challenges were ultimately combined as a form of BMCPython and then utilised to verify python applications.

The following methods were used to achieve the five objectives listed in the introduction:

- Understand the background and logic of ESBMC-- The student was able to provide a clear description of ESBMC logic theory via an intensive literature search and citations in the background theory portion.
- Develop BMCPython by calling the library of ESBMC-- This was accomplished by Cython implemented, after converting the Python source code to ANSI-C, BMCPython will automatically execute the specified ESBMC test library to verify the transformed code.
- Determine the ideas and concepts underlying vulnerability discovery of BMCPython in Python bytecode-- By receiving failed validation data from BMCPython, users can readily detect faults in python source code.
- Reproduce the extension described in the initial research and ascertain its limits and disadvantages-- When the technique was presented, the majority of its flaws and limitations were highlighted. The editor will also address the verification scope and faults of BMCPython at the subsequent reflection session.
- Verifying and analysing the BMCPython after the developments-- This was accomplished by thousands of tests and a detailed BMCPython analysis was given in chapter 3.

To sum up, the student has developed an effective and convenient python verification tool and a complete demonstration system has been created for programme verification and model checking. Capabilities for project development and report writing have also increased.

5.2 Reflection

Every coin has two sides, and BMCPython is no exception. Several Python features were not supported during the development of Intermediate Representation due to the limitations of the compile tool. Cython is not a superset of the Python programming language in its entirety. Restrictions are applicable if the following conditions are met: Regardless of whether they are expressed in *def* or *cdef* syntax, function definitions cannot be nested inside other function definitions. A class can be defined only at the module's top-level, not within a function. The 'import *' form of the reference library is not permitted in any region (other formats of the import statement are acceptable, though). In Cython, it is not possible to implement generators. Additionally, user cannot utilise the *globals()* function or the *locals()* function. The foregoing constraints are likely to stay, as they would be hard to remove, and Cython proposed applications do not necessitate their removal. Class and function declarations cannot currently be inserted inside control structures; however, this may change in the future. In the meanwhile, list conditional expressions are not available. Unicode is not supported. The docstrings for extension types of unique methods are not functional. It is not suggested at this time to utilise string literals as comments in areas where executable declarations are not permitted. Given the above limitations, there is still too much functionality that

needs to be optimized for BMCPython, otherwise, a significant portion of python code will fail due to compile problems rather than validation problems.

There is one more point to reflect on, due to lack of knowledge of robotic applications, I was unable to improve on this section, so only BMCPython has been created without practical application yet. It is also my biggest regret regarding my project.

5.3 Future work

After submitting this project, I will monitor if the compiler's libraries have been updated and optimized to support more Python features, and if time allows, I will even attempt to construct one to two compiler-callable Python features libraries this summer.

For the robotic application, I think Unmanned Aerial Vehicles[24](UAVs or drones) are also a robotics application. This BMCPython method can be applied to the verification of UAV path[17], UAV GPS analysis and Python Fuzzer in the computer vision field. The most important thing is that I will test my program in the robot application to optimize its compatibility to the greatest extent.

Bibliography

[1] Alglave, J., Kroening, D., and Tautschnig, M. (2013). *Partial orders for efficient bounded model checking of concurrent software*. In: Computer Aided Verification, volume 8044 of LNCS, pages 141–157.

[2] Armando, A., Mantovani, J., and Platania, L. (2009). *Bounded model checking of software using SMT solvers instead of SAT solvers*. In: Software Tools for Technology Transfer, 11(1):69–83.

[3] Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. (2009). Handbook of Satisfiability, *chapter Satisfiability Modulo Theories*, In: pages 825–885. IOS Press.

[4] Beyer, D. (2019a). *Automatic verification of c and java programs*: Sv-comp 2019. In Beyer, D., Huisman, M., Kordon, F., and Steffen, B., editors, Tools and Algorithms for the Construction and Analysis of Systems. In: pages 133–155, Cham. Springer International Publishing.

[5] Beyer, D. and Lemberger, T. (2017). *Software verification: Testing vs. model checking*. In: Haifa Verification Conference, pages 99–114. Springer.

[6] Bradley, A. R. (2011a). *SAT-based model checking without unrolling*. In Verification, Model Checking, And Abstract Interpretation, In: volume 6538 of LNCS, pages 70–87.

[7] Clarke E.M., Klieber W., Zuliani P. , *Model Checking and the State Explosion Problem*. In: Meyer B., Nordio M. (eds) Tools for Practical Software Verification, 2012.

[8] Clarke EM, Henzinger TA, Veith H. Handbook Of Model Checking, chap. *Introduction To Model Checking.* In: Springer International Publishing, 2018; 1–26.

[9] Clarke, E. and Emerson, E., 1981. *Design and synthesis of synchronization skeletons using branching time temporal logic.* In: Logics of Program*s*, volume 131 of Lecture Notes in Computer Science, pp.52-71.

[10] Clarke, E., Biere, A., Raimi, R. et al. *Bounded Model Checking Using Satisfiability Solving.* In: Formal Methods in System Design 19, 7–34, 2001.

[11] Clarke, E., Kroening, D., Lerda, F.: *A tool for checking ANSI-C programs.* In: TACAS, LNCS 2988, pp. 168–176, 2004.

[12] Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P. and Trtik, M., 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. *Computer Aided Verification - 30th International Conference*, In: volume 10981 of Lecture Notes in Computer Science, pp.183-190.

[13] Cordeiro, L., Kroening, D. and Schrammel, P., 2019. JBMC: Bounded Model Checking for Java Bytecode - (competition contribution). *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics*, In: volume 11429 of Lecture Notes in Computer Science, pp.219-223.

[14] Cordeiro, L.C., Fischer, B., Marques-Silva, J.: *SMT-based bounded model checking for embedded ANSI-C software*. In: IEEE TSE 38(4), pp. 957–974, 2012.

[15] Cython.org., n.d. *About Cython*. [online] Available at: https://cython.org/

[16] Dasgupta, Sanjoy; Papadimitriou, Christos; Vazirani, Umesh. Algorithms (PDF) 1. In: McGraw-Hill Science/Engineering/Math. 2006-09-13: 176 [2015-04-11]. ISBN 978-0073523408

[17] Dey, V., Pudi, V., Chattopadhyay, A., and Elovici, Y. (2018). *Security vulnerabilities of unmanned aerial vehicles and countermeasures: An experimental study.* In: VLSID, pages 398–403. IEEE Computer Society.

[18] Gadelha MR, Monteiro F, Cordeiro L, Nicole D. *ESBMC v6.0: Verifying C programs using k-induction and invariant inference.* Tools And Algorithms For The Construction And Analysis Of Systems, In: LNCS, vol. 11429, 2019; 209–213.

[19] Gadelha, M. Y. R., Cordeiro, L. C., Nicole, D., *Encoding floating-points using the SMT theory in ESBMC: An empirical evaluation over the SV-COMP benchmarks.* In: 20th Brazilian Symposium on Formal Methods (SBMF), LNCS 10623, pp. 91-106, 2017.

[20] Gadelha, M., Monteiro, F. R., Morse, J., Cordeiro, L., Fischer, B., Nicole, D., *ESBMC 5.0: An Industrial-Strength C Model Checker.* In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 888-891, 2018.

[21] GitHub. n.d. GitHub - pallets/flask: *The Python micro framework for building web applications.*. [online] Available at: <https://github.com/pallets/flask> .

[22] Guido van Rossum (2015). Python Enhancement Proposals. [online] Peps.Python.org. Available at: https://peps.python.org/pep-0484/

[23] *IEEE Guide for Software Verification and Validation Plans,* In: IEEE Std 1059-1993, vol., no., pp.1-87, 28 April 1994, doi: 10.1109/IEEESTD.1994.121430

[24] Kerns, A. J., Shepard, D. P., Bhatti, J. A., and Humphreys, T. E. (2014). *Unmanned aircraft capture and control via gps spoofing*. In: Journal of Field Robotics, 31(4):617–636.

[25] Merz, F., Falke, S., Sinz, C.: *LLBMC: Bounded model checking of C and C++ programs using a compiler IR*. In: VSTTE, LNCS 7152, pp. 146–161, 2012.

[26] Monteiro FR, Alves EHdS, Silva IS, Ismail HI, Cordeiro LC, Filho EBdL. *ESBMC-GPU a context-bounded model checking tool to verify CUDA programs.* In: Science of Computer Programming 2018; 152:63-69.

[27] Morse, J., Cordeiro, L.C., Nicole, D., Fischer, B.: *Model checking LTL properties over ANSI-C programs with bounded traces*. In: SoSym 14(1), pp. 65–81, 2015.

[28] Monteiro, F. R. , FAP Januário, Cordeiro, L. C. , & Filho, E. . (2017). *BMCLua: a translator for model checking Lua programs.* ACM SIGSOFT Software Engineering Notes, 42(3).

[29] Professionalqa.com. n.d. *Software Verification* |Professionalqa.com. [online] Available at: <https://www.professionalqa.com/software-verification> .

[30] Ramalho M, Freitas M, Sousa F, Marques H, CordeiroL C, Fischer B. *SMT-based bounded model checking of C++ programs.* In: Engineering of Computer Based System, 2013; 147–156.

[31] Rocha, H., Barreto, R.S., Cordeiro, L.C.: *Memory management test-case generation of C programs using bounded model checking*. In: SEFM, LNCS 9276, pp. 251–267, 2015.

[32] Ryan Gonzalez (2016). Python Enhancement Proposals. [online] Peps.Python.org. Available at: https://peps.python.org/pep-0526/

[33] Skiena, Steven. The Algorithm Design Manual (PDF) 2. In: Springer. 2008-07-26: 212 [2015-04-11]. ISBN 978-0073523408. doi:10.1007/978-1-84800-070-4

[34] Stefan Behnel, Robert Bradshaw, William Stein, Gabriel Gellner, et al..(2022). Cython documentation, *Users Guide of Language Basics.* [online]cython.readthedocs.io. Available at: https://cython.readthedocs.io/en/latest/src/userguide/language_basics.html