Program Analysis: Towards the Analysis of CPython Bytecode

André Theuma

Supervisor: Dr. Neville Grech

May, 2022

Submitted in partial fulfilment of the requirements for the degree of B.Sc. Computer Science.

L-Università ta' Malta
Faculty of Information & Communication Technology

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Neville Grech, for his patience and guidance throughout this project.
A sincere thank-you also goes to my close friends and my family for their constant support, without whom this dissertation would not have been possible.

Abstract

Program analysis methods offer static compile-time techniques to predict approximations to a set of values or dynamic behaviours which arise during a program's run-time. These methods generate useful observations and characteristics about the underlying program, in an automated way.
PATH (Python Analysis Tooling Helper) is a static analysis tool created in this project, which generates a standardized Intermediary Representation for given functions, allowing analysis metrics to be generated from the facts produced by the tool. The goal of this project was to create a framework that generates facts from a function, in addition to an IR that is amenable for further analysis. The framework created should simplify the engineering complexity of fact analysis for future use. PATH would disassemble CPython bytecode into a more straightforward representation, making any further possible analyses a simpler task, as analysis can be conduced on the generated IR.
The final findings of the project indicate that performing analysis on the IR generated by PATH is indeed a simpler task than generating facts manually and conducting block analysis without such a framework. These results are satisfactory and hold up to the aims of this project.

Contents

List of Figures

List of Tables

List of Abbreviations

AIL Abstract Interpreter Loop
PATH Python Analysis Tooling Helper
IR Intermediary Representation
MD5 Message-Digest 5 Algorithm
DFA Data Flow Analysis
CFA Control Flow Analysis
CFG Control Flow Graph
III Iterative Instruction Inspection
I/O Inputs & Outputs
PVM Python Virtual Machine
PA Program Analysis
VM Virtual Machine

Chapter 1

Introduction

1.1 Motivation

As an increasing number of people are becoming reliant on complex software systems, the lack of security and analysis tooling frameworks is unacceptable in this day and age. Such frameworks provide vital insight into software systems; allowing further development and refinement of said systems. Core vulnerabilities are exposed, and optimizations are possible. Tools such as Pylint (Logilab, 2022), and Bandit (Developers, 2022) already exist in the industry.

1.2 Preliminary Overview

1.2.1 Python & CPython

Python is a high-level, object-oriented scripting language (Lutz, 2001), suited for a wide range of development domains; from text processing (Bonta and Janardhan, 2019) to machine learning (Goldsborough, 2016) to game development (Sweigart, 2012). The language's wide adoption (TIOBE's Language of the Year: 2007, 2010, 2018, 2020, 2021; (Index, 2022)) may be attributed to the fact that it is based on the English language (Saabith et al., 2019), making it easy to learn; aiding in the production of relatively complex programs. It is used extensively for rapid prototyping and the development of fully-fledged real-world applications.

The predominant and most comprehensive implementation of Python is known as CPython (van Rossum); a bytecode interpreter for Python, written in C.

1.2.2 Program Analysis

Complex programs imply complex behaviours. Such behaviours have to be analysed, as they might highlight potential vulnerabilities, and possibly indicate where optimisations can be carried out in the program. This area of interest is known as Program Analysis. Program Analysis provides answers to the following questions;

- Can the provided code be optimized?
- Is the provided code free of errors?
- How does data flow through the program & in what order do instructions get executed (Control-Flow)?

Naturally, as an increasing amount of modern-day systems and frameworks are developed in Python, the need for conducting program analysis on these systems is ever-growing. There are two main approaches to program analysis; Dynamic Program Analysis & Static Program Analysis. Dynamic analysis is the testing and evaluation of an application during runtime, whilst static analysis is the testing and evaluation of an application by examining the code, producing facts and deducing possible errors in the program from the facts produced; without code execution (Intel, 2013). Since all (significant) properties of the behaviour of programs written in today's programming languages are mathematically undecidable (Rice, 1953), one must involve approximation for an accurate analysis of programs. This kind of analysis cannot be carried out by a Dynamic analysis as carrying out a runtime analysis only reveals errors, but does not show the absence of errors (Møller and Schwartzbach, 2012); being the primary motivation behind Static analysis. With the right kind of approximations, Static analysis provides guarantees regarding the properties of all the possible execution paths a program can take, giving it a clear advantage over Dynamic analysis; thus will be the main topic of interest in this paper.

1.3 Proposed Solution

PATH provides general metrics for functions; known as facts, along with a standardized IR (Intermediary Representation) for external analysis. PATH also creates a Control Flow Graph for flow analysis.

1.3.1 Aims & Objectives

The aim of this dissertation is the production of an easily implementable analytical tool (PATH) for existing software systems. The tool is to be used on Python V3.10 (van Rossum)

systems which are interpreted with the CPython (van Rossum) interpreter. It needs to scale up to larger systems and still provide accurate metrics. The analytical tool must also generate a standardized IR of the functions present in the system that are analysed. The standardized generated IR has to summarize the block analysis done by PATH.

The creation of this tool is vital as there is a lack of security and analysis tooling, for what is the world's most used programming language (Index, 2022); giving developers increased freedom of choice when having to choose an analysis framework for their projects.

1.4 Document Structure

This paper is composed of five chapters; broken down in the following manner:

Chapter 1 contains the introductory content, along with a brief overview of the technologies and ideas explored further along in this paper. The introduction also gives a run-down of the main objectives met in this project.

Chapter 2 gives an in-depth literature review of the content briefly touched upon in the introductory chapter [Chapter 1]. The literature review consists of (but is not limited to) the following works; Python, the uses of Python, Python's bytecode & CPython's *ceval.c* interpreter, and Static Analysis Tooling.

Chapter 3 delves into both the methodology and implementation of the disassembler (PATH). The design of choice is further discussed together with the reasoning behind such a design.

Chapter 4 evaluates the results produced by PATH and answers specific research questions, ensuring that the Analytical toolkit works as intended. A couple of case studies are included, testing the scalability and ease-of-use of PATH.

Chapter 5 presents the conclusions of the project and any suggestions for any possible further work.

1.5 Contributions

We carried out the reverse engineering of the individual CPython bytecodes; observing their effect on the value stack. These operations are concisely noted in the Opcode Table.

We automated the production of facts from Python functions, expediting different analyses of functions.

We standardized the generation of an IR; exposing to the user for better understanding of function behaviours. Alongside IR generation, control flow with functions is indicated by a Control-Flow graph.

Chapter 2

Background & Literature Overview

2.1 Python

2.1.1 History

Python being a platform-independent (Srinath, 2017), high-level programming language (Van Rossum and Drake Jr, 2020, pp.2–4), makes the development of complex software systems a relatively non-trivial task in comparison to the production complexity that comes along with other comparable programming languages, such as C (Summerfield, 2007).

The language was developed to be a successor of the ABC programming language (Geurts et al., 1990) and initially released as Python V0.9.0 in 1991. Similarly to ABC, Python has an English-esque syntax but differs in its application domain; Python being a tool intended for use in a more professional environment, whilst ABC is geared towards non-expert computer users (van Rossum and de Boer, 1991, pp.285–288).

2.1.2 Features & Philosophy

The simplicity of Python enables it to be an extremely popular language to use in the industry of software development (Index, 2022). It was designed to be highly readable, thus removing the 'boilerplate' style used in the more traditional languages, such as Pascal. Python uses indentation for block delimiters, (off-side rule (van Rossum, pp.4–5)) which is unusual among other popular programming languages (van Rossum, pp.2–3); new blocks increase in indentation, whilst a decrease in indentation signifies the end of a block. It supports a dynamic type system, enabling the compiler to run faster and the CPython interpreter to dynamically load new code. Dynamic type systems such as Python offer more

flexibility; allowing for simpler language syntax, which in turn leads to a smaller source code size (Thomas, 2013). Although dynamically typed, Python is also strongly-typed; disallowing operations which are not well-defined. Objects are typed albeit variable names are untyped.

One of Python's most attractive features is that it offers the freedom to allow the developer to use multiple programming paradigms (Van Rossum et al., 2007); appealing to a wider audience. Python also includes a cycle-detecting garbage collector (Van Rossum et al., 2007), freeing up objects as soon as they become unreachable (Van Rossum and Drake Jr, 1994, pp.9–10). Objects are not explicitly freed up as the collector requires a significant processing overhead (Zorn, 1990, pp.27-30), and re-allocating memory to objects every time an object is required is resource consuming. Python has module support in its design philosophy, formulating a highly extensible language. Modules can be written in C/C++ (Srinath, 2017) and imported as libraries in any Python project; highlighting the extensibility of the language. There are plenty [1] of readily available third-party libraries suited for many tasks, ranging from Web Development (Forcier et al., 2008) to complex Machine Learning frameworks (Pedregosa et al., 2011), further increasing ease-of-use, and supporting the quick and simple development philosophy of Python.

2.1.3 Implementations

There are several environments which support the Python language, known as implementations. The default, most feature comprehensive Python implementation is CPython (Van Rossum, 1995); written and maintained by Guido van Rossum. Other popular re-implementations include PyPy (Bolz et al., 2009), Jython (Juneau et al., 2010) and IronPython (Mueller, 2010). This paper will focus on the CPython implementation of the Python language and will not cover any of the other alternate implementations.

2.2 CPython

CPython is the predominant implementation of the Python language written in C. It has a thin compilation layer from source code to CPython bytecode (See C.2); simplifying the design of the interpreter. Unlike the typically structured program representations, bytecode is easier to parse and has a standardized notation. A comprehensive reference has been compiled for this dissertation, available in Appendix C.

[1] Over 329,000 packages as of September 2021 (Van Rossum et al., 2007)

2.3 Analysis

Program analysis is constituted of four main approaches; Control Flow Analysis, Abstract Interpretation, Type and Effect Systems, and finally Dataflow Analysis (Nielson et al., 2004, pp.1–2). Typically, these approaches are practised in conjunction with each other to provide the most accurate approximate answers. Program analysis techniques should be semantics-based and not semantics directed; the latter is the process by which the information obtained from the analysis conducted is proved to be safe concerning the semantics of the programming language, whilst the former is the process by which the structure of the analysis conducted reflects the structure of the semantics of the language; a process which is not recommended (Nielson et al., 2004, pp.2–3). These approaches also have two main methodologies driving them; Statically analysing a program or Dynamically analysing a program.

Program analysis is conducted before program input, rendering any analysis undecidable. In complexity theory, this is known as Rice's Theorem (Rice, 1953). The analysis would need to compute results that are valid for all possible inputs into the program. Seeing as such a statement is near impossible to back up, the aforesaid analysis approximates; producing a *safe* answer (Andersen, 1994, pp.9–11). *Safe* answers are decidedly *safe* based upon the aim of the analysis and the information provided to the analysis. A result which might be considered *safe* in a certain analysis, may not be in other analyses.

2.3.1 Control Flow Analysis

Control Flow analysis (*Constraint Based analysis*) is the act of determining information about what elementary blocks lead to other blocks, whereby seeing the flow of program control. More formally, such an analysis for each function application gives us which functions may be applied. Control Flow analysis makes use of the constraint system. The essence of this method is to extract several inclusions out of a program. This system creates relations which can be constituted from three different classes (Nielson et al., 2004, pp.10–13);

1. The relation between the values of the function abstraction and their labels.
2. The relation between the values of variables and their labels.
3. The interrelations of application points and their relative function mappings:

   Application Point 1: The constraint representing the formal parameter of the function bounded with the actual parameter value.

Application Point 2: The constraint representing the value outputted by said function.

There are multiple types of CFA analyses (Nielson et al., 2004, pp.139–195):

- Abstract 0-CFA Analysis,
- Syntax Directed 0-CFA Analysis,
- Constrain Based 0-CFA Analysis,
- Uniform *k*-CFA Analysis,

2.3.2 Dataflow Analysis

In Data Flow Analysis, the program is subdivided into sections via elementary blocks, connected by edges, describing the delegation of control in the program. There are two primary methodologies of approaching Data Flow analysis; The Equational Approach, and the Constraint Based approach (as mentioned in Section 2.3.1)

The equational approach extracts a number of equations from a program; belonging to the following classes:

- The relation between the exit information of a node to the entry information of the same node (flow of data).
- The relation between the entry information of a node to exit information of nodes from which control could have possibly come from.

There are multiple types of Intra-procedural Data Flow analyses as may be seen below (Nielson et al., 2004, pp.33–51):

- Available Expression Analysis,
- Reaching Definition Analysis,
- Very Busy Expression Analysis,
- Live Variable Analysis.

An important form of analysis in this subsection is the Reaching Definition Analysis. It is made use of in other analyses, such as Abstract Interpretation. This type of analysis relates distinct labels to allow the identification of the primitive constructs of a program without the need to construct a flow graph.

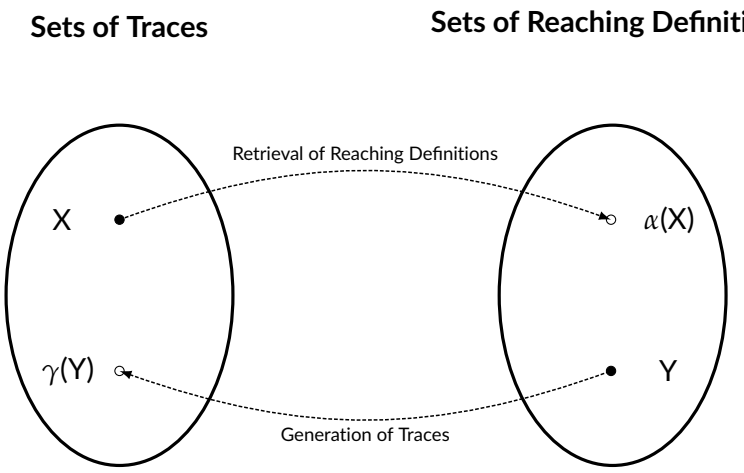


Figure 2.1: Realization of Galois Connection

2.3.3 Abstract Interpretation

This form of analysis is the way Analyses are calculated rather than how their specification is constructed. Thus, it is independent of the specification style. The analysis maps an initial state and a fixed-point from a concrete domain onto an abstract domain; enabling program properties to be decidable (2.2). Abstract Interpretation is a three-step procedure (Nielson et al., 2004, pp.13–17);

Collection of Semantics

This is the preliminary step which records a set of traces that can possibly reach a program point. A trace records the origins of a variables value. From a trace, a set of semantically reaching definitions can be extracted; pairs of variables and labels.

Galois Connections

A Galois connection is the joining of the *trace* sets and a *reaching definition* sets; creating a relation. The construction of this joint set is realized by an abstract function α and a concretisation function γ (seen in Figure 2.1), forming the set (α, λ). The abstraction function extracts reachability information present in a set of traces, whilst the concretisation function produces all traces which are consistent with the given reachability information (Nielson et al., 2004, pp.14–15).

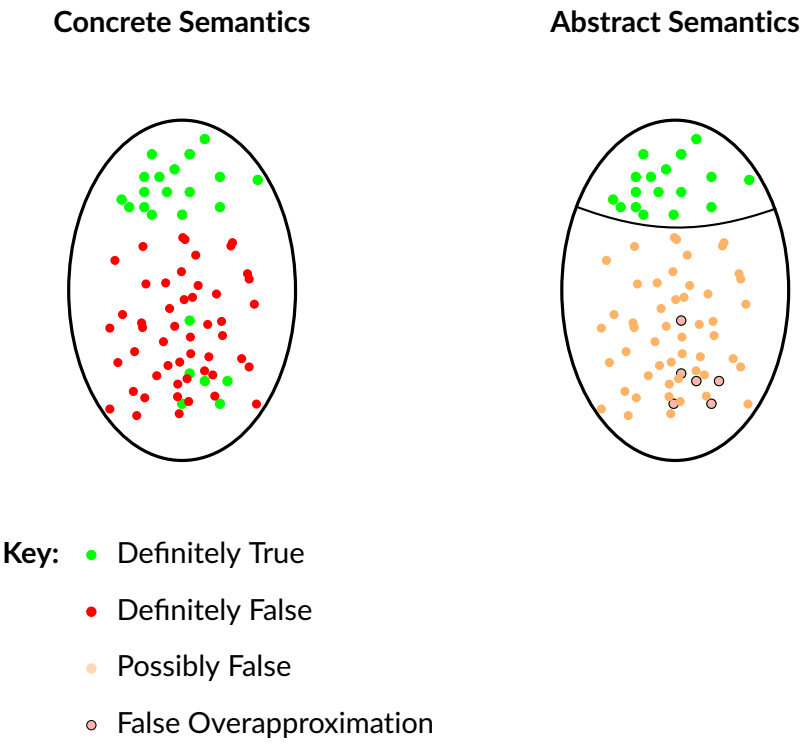


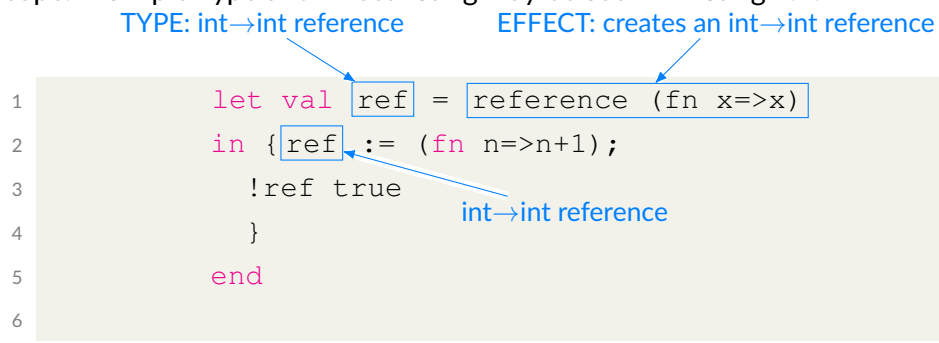
Figure 2.2: Efficiency-Precision trade-off presented by Abstraction Interpretation

Induced Analysis

Finally, an induced analysis is performed on the information obtained, providing a calculated analysis of the previously undecidable properties, as shown in Figure 2.2. This type of analysis provides a result which is produced efficiently, and relatively precisely.

2.3.4 Type and Effect Systems

Type and effect systems are the amalgamation of both an Effect System and an Annotated Type System (Nielson et al., 2004, pp.17–18). In an Effect System information about what happens when an execution occurs, rendering a change the current state, is produced (ex: what exception might be raised if this execution occurs). In an Annotated Type system the judgements that occur describe certain properties of states, such as a variables signum. Further detail into this method of analysis will not be delved into, as it is out of this papers scope. A simple Type and Effect listing may be seen in Listing 2.1.




```
1           let val ref = reference (fn x=>x)
2           in (ref := (fn n=>n+1);
3             !ref true
4             )
5           end
6
```

Listing 2.1: Type & Effect System example on SML Code Snippet

Chapter 3

Methodology

This chapter describes how the Proposed Solution was implemented, and why certain decisions were taken regarding design implementation. This chapter also explores the problems encountered whilst formulating such a design.

3.1 Initial Challenges

The effects that CPython Bytecodes have on the CPython stack are not explicitly defined; posing as an initial challenge. CPython also implements certain bytecode optimizations, initially hindering the ability to fully understand the operations taking place in relation to the source code. Official Python documentation is very vague, intertwining terminologies making it difficult to understand the inner workings of the PVM.

3.2 Abstract Design

PATH is a three-phase bytecode inspection tool (Sections 3.2.1 & 3.3 respectively). The high-level implementation is as follows; PATH iteratively inspects each bytecode instruction produced by the Python Compiler, generating facts as instructions are iterated through.

3.2.1 Phase Overview

1. The preliminary phase is the setup phase. This phase tackles the initialization of all the global variables required for bytecode inspection along with the statement relations.

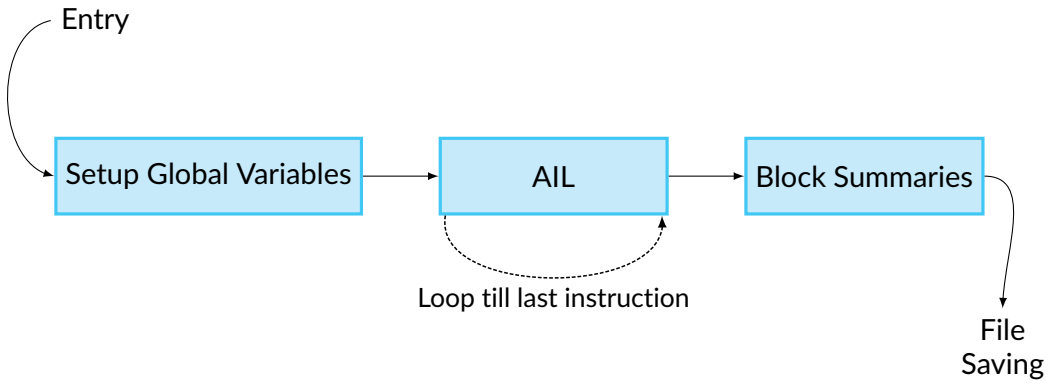


Figure 3.1: Phases of PATH

2. The second phase is the iteration phase; AIL. This phase is where the main logic of the framework is found. The manipulation of local and global variables, pertaining to the IR and fact generation also occurs in this phase.

3. The third and final phase generates a block summary, by retrieving information from the generated facts in phase 2.

3.3 Concrete implementation

3.3.1 Setup

The setup stage is the entry point to the framework. This stage is simple; setting up of the basic variables, which provide a basis for the upcoming logic of PATH.

Relation Initialization

The first stage deals with the initialization of the *Statement Relations*; imperative for fact generation. Most of the relations are initially realized as sets. Lists are only used for the relations which are required to be stored in an ordered fashion. This requirement is motivated by the relations dependency, so as to achieve proper internal fact generation. These relations are `simple_statement_ir`, `statement_uses_local`, and `statement_uses_global`. By design, as one reaches the termination of PATH, these relations are consolidated into a Python dictionary; returned as `fact_dict`. An in-depth review of all relations is found in Section 3.4

Global Variable Initialization

Following the initialization of the relations, the *Global Variables* are conceived. These variables are used throughout PATH, namely for proper block and instruction variable handling, thus are divided into two main distinct groups; Global Instruction Variables, and Global Block Variables. Any other variables are classified as Miscellaneous Variables. All Global Variables are initialized as empty variables, unless said otherwise.

3.3.2 AIL

An abundant amount of the framework's logic is packed into the AIL. This loop iterates through every bytecode instruction, recording attributes of the instruction, and how each instruction interacts with the previous instruction, and will interact with the next instruction. The iterative nature (known as III) of the AIL was inspired by the Python VM Innards workings (See Figure C.3). Inspecting the bytecodes in such a fashion (i.e. generating all the facts iteratively) is arguably faster than having a phase per fact needed to be generated. In addition to fact generation, we also incorporated CFA and IR generation in the III fashion; taking a different approach to how current inspection tools operate (such as Pylint), whereby first the IR is generated, following an analysis conducted on the IR itself. This is a multiphase process, in comparison to the single phase design created by AIL.

Bytecode Validity

Firstly, bytecode instruction validity is ensured via the specified opcode dictionary (supported Bytecode Instructions can be found in Opcode Table).

Local Variable Initialization

Following verification, local instruction variables are set. These variables are the first to be set as they are the basis for fact generation and block generation. These variables are also the foundation for the IR generated, namely for setting the instruction line number and the instruction identifier. The former is dynamically set, dependent on the amount of opcodes that need to be processed, whilst the latter is generated by a specially designed MD5 algorithm, creating parameter dependent unique hashes for every instruction.

Block Handling

Block handling is the subsequent step, modifying the local block variables. In PATH the notion of a [elementary] block is a primitive from which a program is constructed by. Entry

and exit points of blocks depend on the current program flow (as seen in Section 2.3.2); they form part of the basis of both DFA and CFA. New blocks are uniquely identified (similarly to the instruction identifier) and instructions are bound to their respective block (as is seen in Section 3.4). By design, each block has its own stack and unique properties (see Section 3.5) that are used as a basis for both fact generation and accurate block analysis.

An interesting design feature of AIL is the incorporation of both Block Handling and Control Flow Analysis in one sub-phase. Moreover, this sub-phase is only executed if an instruction forms part of a new block; avoiding redundant execution stages.

Opcode Handling

This stage handles general opcode tasks (i.e: general relation generation) and opcode specific tasks (i.e: recursive `MAKE_FUNCTION` dictionary nesting). There exist opcodes, such as `MAKE_FUNCTION` which require additional parameters to be considered for accurate fact generation. The incorporation of the opcode specific IR generation in this stage, expedites the process of fact and IR generation, compared to having the IR generated in a separate stage.

IR Handling

The penultimate process is the IR Generation stage. This stage handles the IR generation according to the current opcode. IR generation is further delved into in Section 3.6.

Updating Global Variables

The final stage in the AIL simply updates the global variables that are to be used in the next iteration.

3.3.3 Block Summary

The termination of PATH is brought about by the block summaries, conducted on the facts generated in the AIL. A block summary is generated for every block, showing where blocks start and end; useful for external debugging purposes and instruction grouping. The generation of a block summary is a form of block analysis.

3.4 Fact Generation

In PATH a fact is information which is generated from the program that is being inspected. These facts are useful for end users as they provide deeper insight in the operations that occur in the program, reducing the overall engineering complexity of program analysis. PATH produces three types of facts: Statement facts, Block facts, and Program facts. Facts are generated in III fashion (per bytecode instruction) and outputted to the user as *.fact* files. A table of all the facts and their descriptions is found in the Facts Table.

3.5 Control Flow

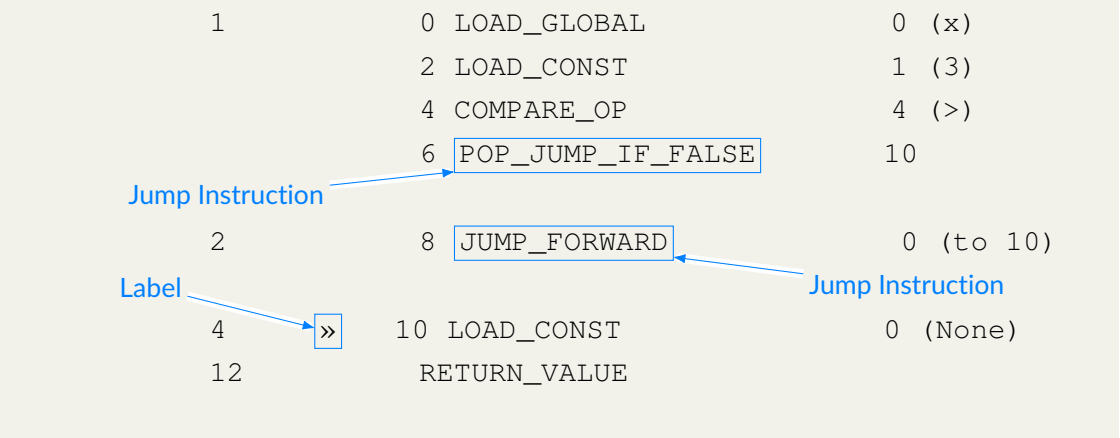
Control Flow in PATH is implemented within the block handling stage in III fashion, as mentioned in Section 3.3.2. The block handling stage is broken down in three main parts; block variable generation, control flow handling and block I/O.

Block Variable Generation

Block variables are only generated if the start of a new block is detected. New blocks are based on jumps and labels, whereby a new block starts after a jump instruction or at a label. The latter is a jump target (i.e: which bytecode a jump instruction would jump to) and the former is self-explanatory. The detection of a new block triggers the block variables to be generated along with certain facts pertaining to block information.

```
1              if x>3:
2                  #do something#
3              else:
4                  #do something else#
5
```

Listing 3.1: Conditional statement script



Listing 3.2: Bytecode Dissasmbly of Listing 3.1

Control Flow Generation

Following the initialization, links between the edges of basic blocks are generated; as shown in Figure 3.2. It is ensured that upon encountering a jump, no redundant relations are created between blocks; taking care to only create relations between linked block edges (i.e: a block's *if* block cannot enter the same *else* block). These type of relations are avoided by creating a unique identifier (`block_link_id`); uniquely identifying a link between two blocks.

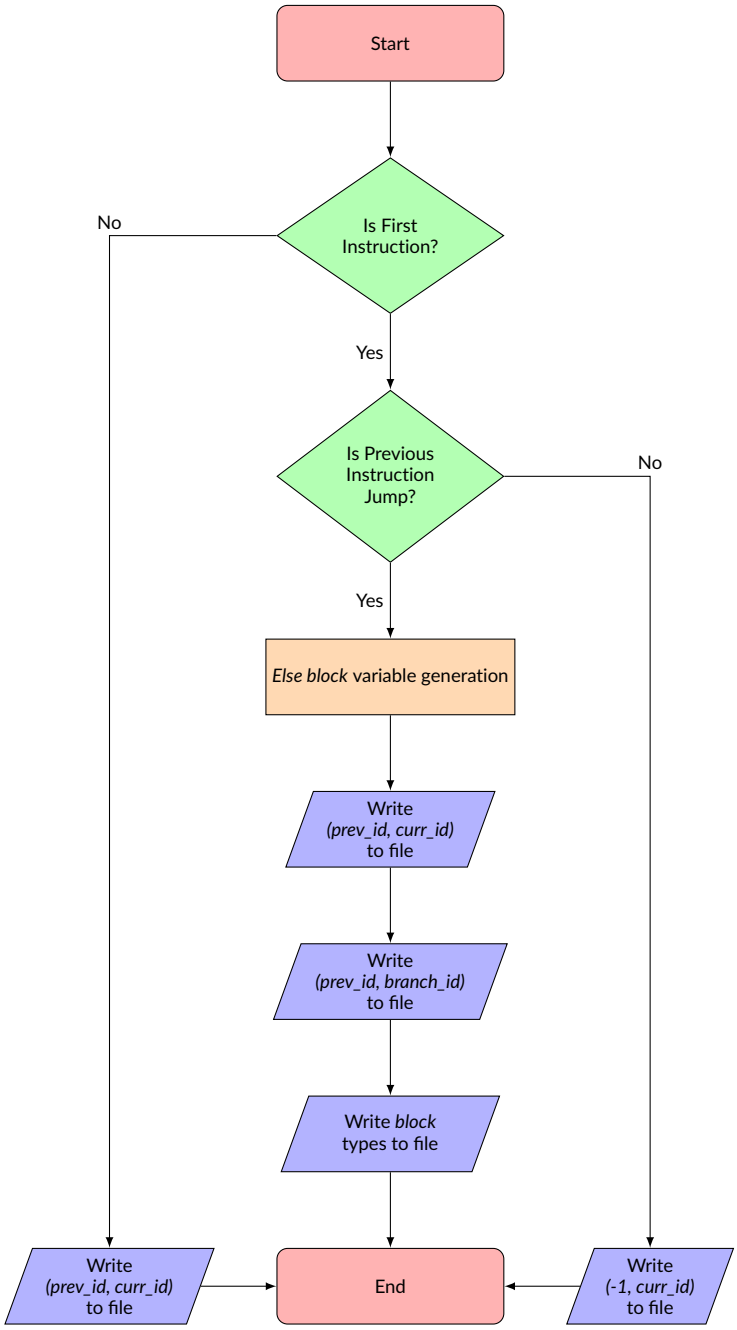


Figure 3.2: Control Flow Handling

3.6 IR Generation

The IR that PATH creates is intended to reduce the engineering complexity for further program analysis. The complex task of recording variables that are consumed and created

in a more concise form was needed, so as to resolve the tedious task of manually keeping note of all stack operations. PATH currently offers this functionality for three primary operation subtypes:

1. Binary Operation Binding
2. Function Calling Binding
3. Method Loading Binding
4. Return Value Binding

Providing a standardized representation for the cases above greatly simplifies the final representation of a program. IR generation takes a III approach; following a sequential iterative generational pattern.

Binary Operation Binding takes the arguments from a binary operation (see Opcode Table), binding them to the variable specified, summarizing a binary operation as shown in Listing 3.3.

```
<inst_id> <var_id> = <inst_name> <arg1> <operation> <arg2>
```

Listing 3.3: Binary Operation Binding Syntax w/variable

Binary operations which do not have a variable bound to them produce the IR shown in Listing 3.4

```
<inst_id> <inst_name> <arg1> <operation> <arg2>
```

Listing 3.4: Binary Operation Binding Syntax w/no variable

Function Calling Binding takes the function and arguments passed to the said function, binding them to the function identifier, indicated by a succeeding `STORE_FAST` instruction. This is shown below.

```
<inst_id> <func_id> CALL_FUNCTION <function_name><arg1> ...
    <argN>
```

Listing 3.5: Function Calling Binding

Method Loading Binding takes any `LOAD_METHOD` calls, merging them with the arguments of its respective `CALL_METHOD`. The latter instruction indicates the number of positional arguments that are passed to the method loaded by the former instruction.

```
<inst_id><func_id>LOAD_METHOD<function_name><arg1>...<argN>
```

Listing 3.6: Method Loading Binding

Return Value Binding simply represents the return value bound with its instruction identifier.

```
<inst_id> RETURN_VALUE <arg1>
```

Listing 3.7: Function Calling Binding

3.7 Technical Implementation

The technical implementation is documented within the source code, as inline comments.

Chapter 4

Evaluation

In this chapter PATH is tested and evaluated. For testing, programs with varying complexity are used; simple programs ranging to more complex programs. There are three primary Research Questions, which will be tested in Sections 4.2, 4.3 and 4.4; validating PATH.

4.1 Research Questions

The research questions below systematically investigate the need and use of PATH. These questions are delved into more depth in Sections 4.2, 4.3, and 4.4.

RQ 1: How is further analyses facilitated? PATH creates *.facts* files. Such files contain metrics pertaining to the function analysed that are of interest to end users. These files' contents are tabulated in Facts Table.

RQ 2: What findings are concluded from this project? Throughout this project, several undocumented findings have been concluded from results generated by PATH processing the functions in *project/tests*. The primary undocumented findings are listed below:

- Elementary Blocks in CPython retain their content on the frame stacks, propagating through the blocks.
- `LOAD_GLOBAL` is reserved for storing function names.
- Bytecode operations on the frame stack.
- CPython blocks are not elementary blocks.

These findings amongst others are further discussed in Section 4.3.

RQ 3: Is PATH scalable? For PATH to be of any practical use, it must be able to scale appropriately, and traverse through different function calls. This is possible as it follows a recursive methodology when dealing with `MAKE_FUNCTION` calls, and also scales linearly in relation to the number of bytecodes in a function.

4.2 Facilitating further analyses

Program analysis is a computationally intensive and time-consuming process. This process is facilitated by the use of automated tools, such as PATH. The *facts* generated by PATH are thought out in such a way so as to analyse functions and to enable any further analyses, such as carrying out a Data Flow Analysis with the information generated by PATH. For a Data Flow Analysis one requires a CFG (generated as mentioned in Section 3.5), and data-flow equations for every node of the CFG. A popular approach to Data Flow Analysis is the Reaching Definitions Method (See section 2.3.2). This would be facilitated with PATH via the CFG generated and the following *.facts* files: *StatementUsesLocal.facts* and *PushValue.facts* (refer to Facts Table); demonstrating that it does indeed reduce the engineering complexity of further analysis.

This tool enables teams to focus on the analysis itself, negating the preliminary step for the creation of different intermediary representations which analysis is conducted on. The lack of professionals in this area makes complex analysis tools expensive and rare to come across.

4.2.1 Relating PATH with current frameworks

In reality, program analysis cannot be simply bisected into two categories; it is more accurately depicted by the scope shown in Figure 4.1

Similar Frameworks

PATH is a pure basic program analysis framework, in which its stereobate lies with the generation of an IR. This framework takes inspiration from several other Static frameworks which operate similarly:

doop A Java pointer and Taint Analysis framework that conducts analysis by the Soufflé Datalog Engine (Bravenboer and Smaragdakis, 2009).

soot A Java optimization framework, providing analyses ranging from CFG construction to Taint analysis (Vallée-Rai et al., 2010).

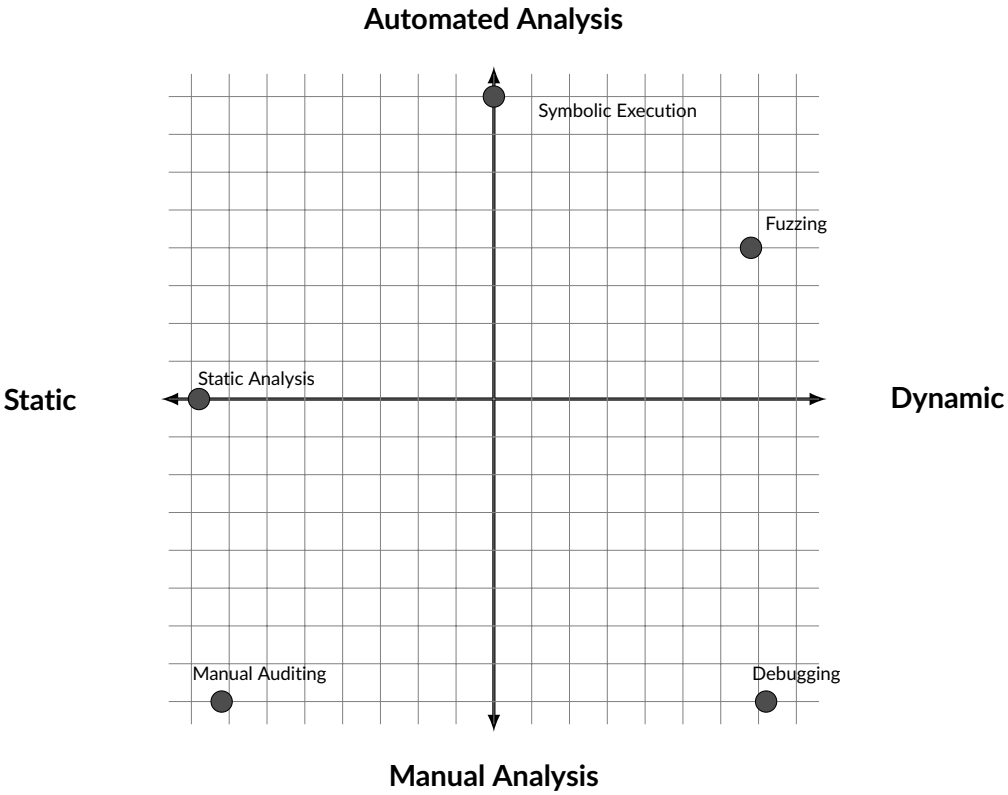


Figure 4.1: Scope of Program Analysis

Gigahorse An Ethereum analysis framework, specializing in the decompilation of smart contracts (Grech et al., 2019).

Vandal A fast and robust security analysis framework for Ethereum smart contracts (Brent et al., 2018).

The frameworks mentioned above are **pure** program analysis frameworks (Static Analysis), whereby the analysis is conducted on an IR.

Alternative Frameworks

As seen in Figure 4.1, there are different methods by which program analysis can be conducted. A popular alternative to the Static Analysis Methodology is the Symbolic Execution method. Symbolic Execution is a technique for automatic software validation and verification; differing from Static analysis in the way that it does not form an opinion about the accuracy of the approximation generated. It is simply used to show an expected symbolic result about a computation.

Mythril A security analysis tool for Ethereum smart contracts, following a symbolic execution methodology (Di Angelo and Salzer, 2019).

Manticore A symbolic execution tool to analyse binaries and Etherum smart contracts (Mossberg et al., 2019).

angr A platform-agnostic binary analysis framework making use of a symbolic execution engine (Wang and Shoshitaishvili, 2017).

Pex A .NET white box test generation framework using a dynamic symbolic execution engine (Tillmann and de Halleux, 2008).

4.3 Insights

This section collects the undocumented insights that have been noted from the results generated by PATH alongside the extensive study into CPython itself.

4.3.1 Elementary Block Behaviour

In CPython, an elementary block differs from a block. This is not clear in the documentation provided as one might assume that an elementary block in PA terms is interchangeable with a CPython block. CPython considers a section of code to be a block if it is encapsulated within the following code-blocks:

- `try` block.
- `with` block.
- `except` block.
- `else` block.
- `finally` block.

The start of a CPython block is indicated by a `SETUP_FINALLY` instruction. The end of a CPython block, is indicated with the `POP_BLOCK` instruction whereby all elements of the block are popped along with the block itself, shown in Listing 4.1 and 4.2.

```
def test():
    try:
        print("Hello")
    except Exception:
        print("This is an exception")

```

Listing 4.1: `try...except` script

```
            6           0 SETUP_FINALLY           7 (to 16)

            7           2 LOAD_GLOBAL             0 (print)
                        ...
                        8 POP_TOP
                       10 POP_BLOCK
                       12 LOAD_CONST              0 (None)
                       14 RETURN_VALUE
            8     >>   16 DUP_TOP
                        ...
            9          28 LOAD_GLOBAL             0 (print)
                        ...
                       40 RETURN_VALUE
            8     >>   42 RERAISE                  0
```

Listing 4.2: Excerpt Bytecode Dissasmbly of Listing 4.1

```
def foo():
    global a
    a=3

```

Listing 4.3: `LOAD_GLOBAL` script

4.3.2 `LOAD_GLOBAL` versus `LOAD_NAME`

From the results generated, it was noted that the `LOAD_GLOBAL` instruction was used solely to look-up function names, as they cannot be local variables. Typically, the compiler compiles source code with `LOAD_NAME` unless the variable stored is implicitly a global variable (such as a function name), or defined explicitly, as shown in Listing 4.3

```
              3           0 LOAD_CONST               1 (3)
                          2 STORE_GLOBAL             0 (a)
                          4 LOAD_CONST               0 (None)
                          6 RETURN_VALUE
```

Listing 4.4: Bytecode Dissasmbly of Listing 4.3

4.3.3 Bytecode Operations

One of the most important findings that has come from the creation of PATH is the clear, concise Opcode Table. Official documentation provided by Python does not explicitly tabulate the operations on the stack performed by each instruction. CPython source code was inspected (namely *ceval.c*), taking note of `PUSH()/STACK_GROW()` and `POP()/SHRINK()` which both grow the value stack and shrink the value stack, respectively.

The creation of the Opcode Table has allowed for an accurate dictionary driven implementation of the opcodes and their respective effect on the value stack in PATH.

4.4 Scalability & Experimental Results

Individual opcode features of PATH were verified with basic functions found in *project/tests/opcode*, logical functionality of PATH (such as CFG generation) was verified with functions found in *project/tests/logic*, and finally, general purpose testing was conducted via open source programs[1].

Table 4.1: PATH Performance Results

Performance Results			
File Name::main()	Execution Time(ms)	Lines	Bytecode Length
P01_hello.py	0.34	1	10
P02_VariableScope.py	0.74	3	26
P03_ListsOperations.py	5.75	20	238
P04_Factorial.py	1.3	6	50
P05_Pattern.py	4.1	11	156
P06_CharCount.py	1.62	9	64
P07_PrimeNumber.py	2.62	13	104
P08_Fibonacci.py	0.98	4	34
P09_Factorial	0.97	4	34
P10_LCM.py	1.69	8	58
P11_BinaryToDecimal.py	2.20	8	90
P12_DecimalToBinary.py	1.04	3	38
P13_Palindrome.py	1.17	5	44
P14_CheckGreater.py	1.31	6	50

[1] obtained from `https://github.com/OmkarPathak/Python-Programs/tree/master/Programs`

P15_Arguments.py	FAILED	6	82
P16_CountVowels.py	1.18	6	40
P17_EvenOdd.py	1.65	8	62
P18_Logging.py	1.57	9	62
P19_SimpleStopWatch.py	2.41	11	96
P20_OsModule.py	2.30	6	92
P21_GuessTheNumber.py	2.21	11	90
P22_SequentialSearch.py	1.68	8	66
P23_BinarySearch.py	2.83	14	104
P24_SelectionSort.py	2.73	7	106
P25_BubbleSort.py	2.71	4	100
P26_InsertionSort.py	2.76	8	114
P27_MergeSort.py	1.96	7	80
P28_QuickSort.py	1.42	6	54
P29_ArgumentParser.py	1.49	7	60
P30_Array.py	3.56	10	148
P31_SinglyLinkedList.py	3.46	13	142
P32_Multithreading_Client.py	3.58	14	154
P33_DoublyLinkedList.py	1.97	7	80
P34_Stack.py	2.74	9	112
P35_NarySearch.py	11.25	40	422
P36_SimpleReaderWriter.py	2.34	8	96
P37_HangmanGame.py	9.23	47	372
P38_HashingFile.py	2.99	9	126
P39_Queue.py	1.58	7	64
P40_ChiperText.py	1.99	7	80
P41_PortScanner.py	1.02	3	38
P42_MultiprocessingPipes.py	2.31	7	96
P43_BinarySearchTree.py	4.14	15	158
P44_Closures.py	0.89	5	10
P45_MoreOnClosures.py	1.85	8	74
P46_Decorators.py	1.07	4	36
P47_MoreOnDecorators.py	1.21	4	38
P48_CountingSort.py	3.35	13	140
P49_RockPaperScissors.py	11.39	35	434
P50_ListComprehensions.py	10.32	36	378
P51_PythonJSON.py	FAILED	6	62
P52_BucketSort.py	7.54	21	286
P53_ShellSort.py	4.06	11	150
P54_PythonCSV.py	FAILED	4	38
P55_Isogram.py	1.61	7	56
P56_Panagram.py	2.21	14	86
P57_Anagram.py	1.68	6	66
P58_PerfectNumber.py	1.19	5	46
P59_PascalTriangle.py	0.93	4	34
P60_PickleModule.py	2.33	8	80

Performance results covering all test functions can be found in Table 4.1. The table shows that PATH scales with Equation 4.1 as shown in Figure 4.2.

$$TimeTakenToAnalyse[ms] = 0.02592(No.of Bytecodes) + 0.05145 \tag{4.1}$$

Throughout testing, results show that PATH has 95% success-rate of analysis from over 60 functions, which is a satisfactory result.

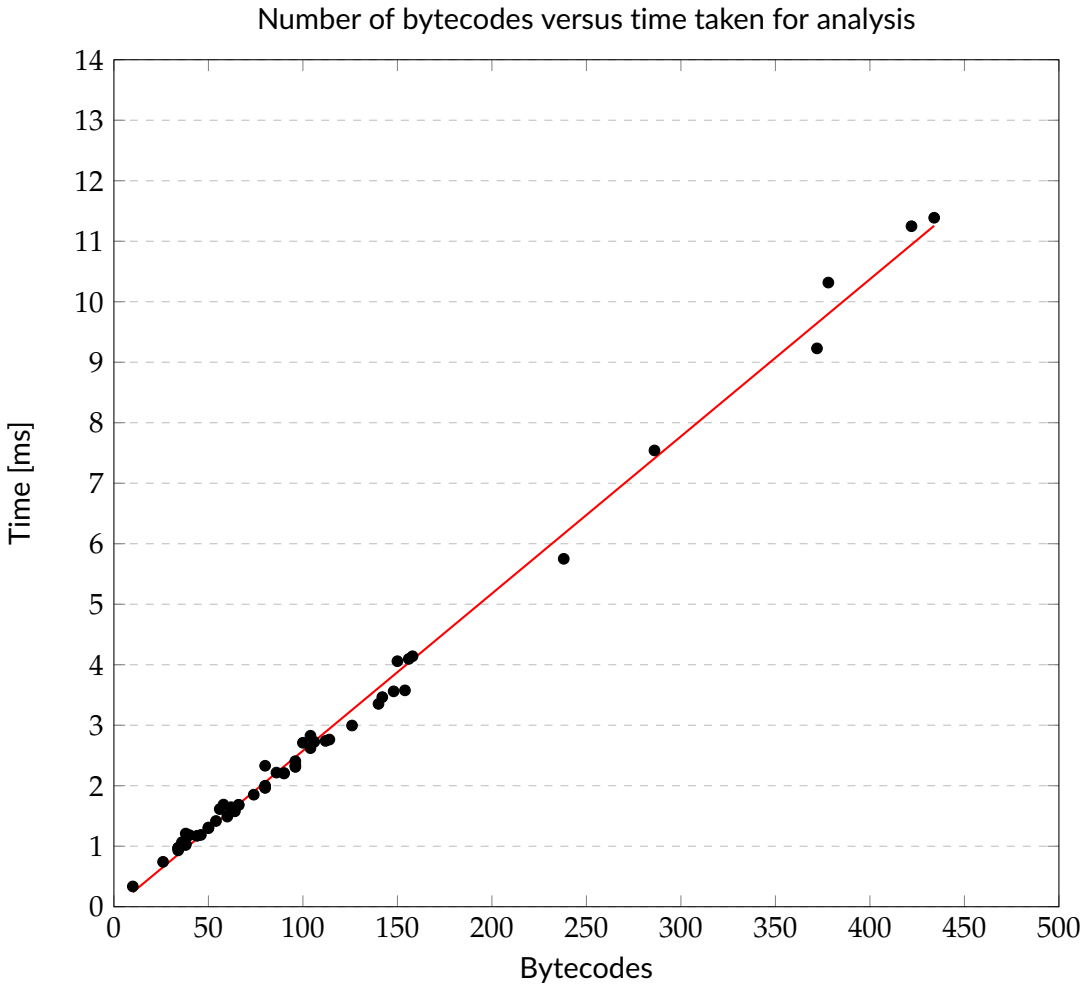


Figure 4.2: PATH performance

As functions increase in size, Manual Auditing becomes an incomprehensibly complicated and time-consuming task. PATH automates this process in a resource and time-efficient way. Performance results are shown in table 4.1. PATH does not enter function calls, but represents them in a shorthand notation as shown in Listing 4.5. Function calls are saved in *resources/FunctionsNotAnalysed.facts*, as functions that are referenced by the function analysed. Functions that are referenced from the Python Standard Library (*stdlib/builtins.pyi*) are excluded, as analysis of user functions is of interest.

```
<filepath>  <function_not_analysed>
```

Listing 4.5: Boilerplate of *resources/FunctionsNotAnalysed.facts*

The time complexity of an Analysis Tool directly affects the adoption of the tool. A lower time complexity is prioritized over accuracy. This is shown in the industry with Andersen's Points-To Analysis and Steensgaard's Points-To Analysis. Andersen's has an average time complexity of $\mathcal{O}(n^c)$ in comparison to Steensgaard's $\mathcal{O}(n)$. Even though Andersen's Points-To Analysis is more accurate than Steengaard's, the latter has been adopted by the industry.

Similarly, PATH scales linearly in relation to the amount of bytecodes in a function (refer to Figure 4.2). $\mathcal{O}(n)$ time complexity gives PATH the potential to be widely adopted, as was Steensgaard's Analysis.

4.4.1 Case Study

To further demonstrate the scaleability of PATH, a Case Study on a simple program shall be conducted. An address book application *project/tests/samplePrograms/P61_AddressBook.py* that performs storing, searching and deletion of records, is tested out in this subsection.

Initial Analysis `main()` is the first function analysed, as it is the entry point of the address book. From this initial analysis we see that the class `AddressBook()` is initialized from the main function, from *resources/FunctionsNotAnalysed.facts* (Listing 4.6). Thus, all functions that lie in the `AddressBook()` class need to be analysed.

Subsequent Analyses The functions pertaining to the `AddressBook()` class are then analysed; `addContacts()`, `getDetailsFromUser()`, `displayContacts()`, `searchContacts()` and `modifyContacts()`. All these results are tabulated in Table 4.2

```
Python Bytecode Analyzer/project/tests/samplePrograms/
   P61_AddressBook.py  AddressBook.addContacts()
Python Bytecode Analyzer/project/tests/samplePrograms/
   P61_AddressBook.py  AddressBook.getDetailsFromUser()
Python Bytecode Analyzer/project/tests/samplePrograms/
   P61_AddressBook.py  AddressBook.displayContacts()
Python Bytecode Analyzer/project/tests/samplePrograms/
   P61_AddressBook.py  AddressBook.searchContacts()
Python Bytecode Analyzer/project/tests/samplePrograms/
   P61_AddressBook.py  AddressBook.modifyContacts()
```

Listing 4.6: *resources/FunctionsNotAnalysed.facts*

```
Vd8b5        CALL_FUNCTION   AddressBook     NO ARGS
Va7fd        LOAD_METHOD     addContacts     NO ARGS
V0a7e        LOAD_METHOD     searchContacts  NO ARGS
Vfadc        LOAD_METHOD     modifyContacts  NO ARGS
Vfa7f        LOAD_METHOD     displayContacts NO ARGS
```

Listing 4.7: *resources/SimpleIR.facts*

Address Book		
Function	Execution Time(ms)	Bytecode Length
`main()`	3.09	124
`AddressBook:: addContacts()`	5.1	200
`AddressBook:: getDetailsFromUser()`	2.83	108
`AddressBook:: displayContacts()`	2.82	110
`AddressBook:: searchContacts()`	5.17	186
`AddressBook:: modifyContacts()`	11.05	452
Totals		
	30.4	1180

Table 4.2: Address Book case study

For verification, the result produced by the PATH Equation 4.1 is 30.64 ms. This result is within the margin of error (error of 0.76%) for timing functions, considering rounding.

Chapter 5

Conclusion & Future Work

In the first section, the importance of Program Analysis was outlined, along with the lack of Program Analysis frameworks. This project aimed to create a framework that simplifies the engineering complexity of further analyses by creating an IR and automating processes, such as fact generation and control flow analysis.

Developed using the Python programming language, PATH is an accurate fact generating tool that produces an intuitive IR and is also able to conduct Control Flow Analysis. The framework created operates with linear time complexity due to its iterative design. Thus, providing users with an advantage compared to other polynomial-time frameworks, such as Andersen's Points-To analysis. PATH also covers over 90% of the CPython bytecodes, proving to be versatile.

Further work on the IR is possible, such as implementing more instructions. The implementation of other types of analyses, such as Data Flow analysis would also be a non-trivial task to implement as the backbone (creation of the CFG) is already implemented. Implementing these features would make the framework more versatile, possibly contending with currently available Program Analysis Tools.

References

Andersen, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

Aycock, J. Converting python virtual machine code to c. In *Proceedings of the 7th International Python Conference*, pages 76–78, 1998.

Bacon, J. W. The stack frame, 2011. URL `http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s07.html`.

Bennett, J. An introduction to python bytecode, 2018. URL `https://opensource.com/article/18/4/introduction-python-bytecode`.

Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

Bonta, V. and Janardhan, N. K. N. A comprehensive study on lexicon based approaches for sentiment analysis. *Asian Journal of Computer Science and Technology*, 8(S2):1–6, 2019.

Bravenboer, M. and Smaragdakis, Y. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., and Scholz, B. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

Conrad, P. Python function calls and the stack lesson 2, 2010. URL `https://sites.cs.ucsb.edu/~pconrad/cs8/topics.beta/theStack/02/`.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. The MIT Press, 3rd edition edition, 2009.

Developers, B. Bandit, 2022. URL `https://bandit.readthedocs.io/en/latest/`.

Di Angelo, M. and Salzer, G. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

Forcier, J., Bissex, P., and Chun, W. J. *Python web development with Django*. Addison-Wesley Professional, 2008.

Foundation, P. S. dis-disassembler for python bytecode, 2022a. URL `https://docs.python.org/3/library/dis.html#dis.Bytecode`.

Foundation, P. S. Code objects, 2022b. URL `https://docs.python.org/3/c-api/code.html`.

Geurts, L., Meertens, L., and Pemberton, S. *ABC programmer's handbook*. Prentice-Hall, Inc., 1990.

Goldsborough, P. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.

Grech, N., Brent, L., Scholz, B., and Smaragdakis, Y. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

Index, T. The python programming language, 2022. URL https://www.tiobe.com/tiobe-index/python/.

Intel. Dynamic analysis vs. static analysis, 2013. URL https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm.

Juneau, J., Baker, J., Wierzbicki, F., Muoz, L. S., Ng, V., Ng, A., and Baker, D. L. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

Landin, P. J. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

Logilab, P. Pylint, 2022. URL https://pylint.pycqa.org/en/latest/.

Lutz, M. *Programming python*. " O'Reilly Media, Inc.", 2001.

Møller, A. and Schwartzbach, M. I. Static program analysis. *Notes. Feb*, 2012.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., and Dinaburg, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

Mueller, J. P. *Professional IronPython*. John Wiley & Sons, 2010.

Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer Science & Business Media, 2004.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

Rossum, G. V. et al. python/cpython, 2022. URL https://github.com/python/cpython/blob/d93605de7232da5e6a182fd1d5c220639e900159/Python/ceval.c#L716.

Saabith, A. S., Fareez, M., and Vinothraj, T. Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development*, 6(10), 2019.

Shaw, A. Your guide to the cpython source code, 2022. URL https://realpython.com/cpython-source-code-guide/.

Srinath, K. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.

Summerfield, M. *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education, 2007.

Sweigart, A. *Making Games with Python & Pygame*. 2012.

Thomas, D. Benefits of dynamic typing. Online, 2013. URL https://wiki.c2.com/?BenefitsOfDynamicTyping.

Tillmann, N. and de Halleux, P. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008. URL https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/.

Tismer, C. Continuations and stackless python. In *Proceedings of the 8th international python conference*, volume 1, 2000.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

van Rossum, G. Python (programming language) 1 cpython 13 python software foundation 15.

Van Rossum, G. Python reference manual. *Department of Computer Science [CS]*, (R 9525), 1995.

van Rossum, G. and de Boer, J. Interactively testing remote servers using the python programming language. *CWi Quarterly*, 4(4):283–303, 1991.

Van Rossum, G. and Drake Jr, F. L. Python reference manual, 1994.

Van Rossum, G. and Drake Jr, F. L. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 2020.

Van Rossum, G. et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36, 2007. URL `https://www.wikizero.com/en/Python_(language)`.

Wang, F. and Shoshitaishvili, Y. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

Zorn, B. *Barrier methods for garbage collection*. Citeseer, 1990.

Appendix A

Opcode Table

Opcode List			
Class	Opname	Push Value	Pop Value
GENERAL OPERATIONS	NOP	0	0
	POP_TOP	0	1
	ROT_TWO	0	0
	ROT_THREE	0	0
	ROT_FOUR	0	0
	DUP_TOP	1	0
	DUP_TOP_TWO	2	0
UNARY OPERATIONS	UNARY_POSITIVE	0	0
	UNARY_NEGATIVE	0	0
	UNARY_NOT	0	0
	UNARY_INVERT	0	0
	GET_ITER	0	0
	GET_YILED _FROM_ITER	0	0
BINARY OPERATIONS	BINARY_POWER	1	2
	BINARY_MULTIPLY	1	2
	BINARY_MATRIX _MULTIPLY	1	2
	BINARY_FLOOR _DIVIDE	1	2
	BINARY_TRUE _DIVIDE	1	2
	BINARY_MODULO	1	2
	BINARY_ADD	1	2
	BINARY_SUBTRACT	1	2

	BINARY_SUBSCR	1	2
	BINARY_LSHIFT	1	2
	BINARY_RSHIFT	1	2
	BINARY_AND	1	2
	BINARY_XOR	1	2
	BINARY_OR	1	2
INPLACE OPERATIONS	INPLACE_POWER	0	1
	INPLACE_MULTIPLY	0	1
	INPLACE_MATRIX _MULTIPLY	0	1
	INPLACE_FLOOR _DIVIDE	0	1
	INPLACE_TRUE _DIVIDE	0	1
	INPLACE_MODULO	0	1
	INPLACE_ADD	0	1
	INPLACE_ADD	0	1
	INPLACE_SUBTRACT	0	1
	INPLACE_LSHIFT	0	1
	INPLACE_RSHIFT	0	1
	INPLACE_AND	0	1
	INPLACE_XOR	0	1
	INPLACE_OR	0	1
COROUTINE OPERATIONS	GET_AWAITABLE	0	3
	GET_AITER	0	0
	GET_ANEXT	1	0
	END_ASYNC_FOR	(andre:) check	(andre:) check
	BEFORE_ASYNC_WITH	1	0
	SETUP_ASYNC_WITH	(andre:) CHECK	(andre:) CHECK
MISC. OPERATIONS	GET_AWAITABLE	0	1
	PRINT_EXPR	0	1
	LIST_APPEND	0	1
	MAP_ADD	0	2
	RETURN_VALUE	0	1
	YIELD_VALUE	0	1
	YIELD_FROM	0	1
	YIELD_FROM	0	0
	SETUP_ANNOTATIONS	0	0
	IMPORT_STAR	0	1

	POP_EXCEPT	0	3
	RERAISE	0	0
	WITH_EXCEPT_START	1	0
	LOAD ASSERTION ERROR	1	0
	LOAD BUILD CLASS	1	0
	GET_LEN	1	0
	MATCH_MAPPING	1	0
	MATCH_SEQUENCE	1	0
	MATCH_KEYS	1	0
	STORE_SUBSCR	0	3
	DELETE_SUBSCR	0	2
ARGUMENT OPERATIONS	STORE_NAME	0	1
	DELTE_NAME	0	0
	UNPACK_SEQUENCE	**ARG**	1
	UNPACK_EX	0	1
	STORE_ATTR	0	2
	DELETE_ATTR	0	1
	STORE_GLOBAL	0	1
	DELETE_GLOBAL	0	0
	LOAD_CONST	1	0
	LOAD_NAME	1	0
	BUILD_TUPLE	1	**ARG**
	BUILD_LIST	1	**ARG**
	BUILD_SET	1	**ARG**
	BUILD_MAP	1	**ARG**
	BUILD_CONST _KEY_MAP	1	**ARG**
	BUILD_STRING	1	**ARG**
	LIST_TO_TUPLE	1	1
	LIST_EXTEND	0	1
	SET_UPDATE	0	1
	DICT_MERGE	0	1
	LOAD_ATTR	0	0
	COMPARE_OP	0	1
	IS_OP	0	1
	CONTAINS_OP	0	2
	IMPORT_NAME	0	1
	IMPORT_FROM	0	1
	JUMP_FORWARD	0	1

	POP_JUMP_ IF_TRUE	0	1
	POP_JUMP_ IF_FALSE	0	1
	JUMP_IF_ NOT_EXC_MATCH	0	1
	JUMP_IF_ TRUE_OR_POP	0	**ARG**
	JUMP_IF_ FALSE_OR_POP	0	**ARG**
	JUMP_ABSOLUTE	0	0
	FOR_ITER	1	0
	LOAD_GLOBAL	1	0
	LOAD_FAST	1	0
	STORE_FAST	0	1
	DELETE_FAST	0	0
	LOAD_CLOSURE	1	0
	LOAD_DEREF	1	0
	LOAD_CLASSDEREF	1	0
	STORE_DEREF	0	1
	DELETE_DEREF	1	0
	RAISE_VARARGS	0	**ARG**
	CALL_FUNCTION	1	**ARG**
	CALL_FUNCTION_KW	0	1
	CALL_FUNCTION_EX	0	1
	LOAD_METHOD	2	1
	CALL_METHOD	1	**ARG**
	MAKE_FUNCTION	1	**ARG**
	BUILD_SLICE	0	**ARG**
	EXTENDED_ARG	0	0
	FORMAT_VALUE	1	**ARG**
	MATCH_CLASS	0	2
	GEN_START	0	1
	ROT_N	0	0

Appendix B

Facts Table

Facts List	
.facts file	Description
BlockInputContents.facts	(Block_ID: `int`, Input_Active_Statment_ID: `int`)
BlockOutputContents.facts	(Block_ID: `int`, Output_Active_Statement_ID: `int`)
BlockSummary.facts	Block_ID: `int`, Start_Statement_ID: `int`, End_Statement_ID: `int`, Start_Instruction_Offset: `int`, End_Instruction_Offset: `int`
BlockToBlock.facts	(Block_ID: `int`, Next_Block_ID: `int`)
BlockType.facts	(Block_ID: `int`, Block_Type: `int`, Block_Link_ID: `int`)
PushValue.facts	SET[(Statement_ID: `int`, Value: `value`)]
SimpleIR.facts	See IR Generation
StatementBlock.facts	SET[(Statement_ID: `int`, Block_ID: `int`)]
StatementBlockHead.facts	(Statement_ID: `int`, Block_ID: `int`)
StatementBlockStackSize.facts	(Statement_ID: `int`, Stack_Size: `int`)
StatementBlockTail.facts	(Statement_ID: `int`, Block_ID: `int`)
StatementCode.facts	SET[(Statement_ID: `int`, CodeObject: `code`)]
StatementDetails.facts	SET[(Statement_ID: `int`, Block_ID: `int`, Push_Value: `value`), Opname: `String`, Line_ID: `float`)]

StatementMetadata.facts	SET[(Statement_ID: `int`, Line_ID: `float`)]
StatementNext.facts	SET[(Statement_ID: `int`, Next_ID: `int`)]
StatementOpcode.facts	SET[(Statement_ID: `int`, Opname: `String`)]
StatementPopDelta.facts	SET[(Statement_ID: `int`, Pop_Delta: `int`)]
StatementPops.facts	SET[(Statement_ID: `int`, Pop_No: `int`)]
StatementPushes.facts	SET[(Statement_ID: `int`, Push_No: `int`)]
StatementUsesLocal.facts	SET[(Statement_ID: `int`, Used_Statement_ID: `int`)]
TotalStatementPopDelta.facts	(Statement_ID: `int`, Running_Pop_Count: `int`)
FunctionsNotAnalysed.facts	(Filepath: `String`, Function_Not_Analysed: `String`)

Appendix C

CPython VM Reference

C.1 CPython

C.1.1 Overview

CPython works transparently, via the PVM (visualised in Figure C.3); an interpreter loop (*ceval.c*) is run and there is no direct translation between the Python code to C (Aycock, 1998, pp.1–2). The PVM is a stack machine whereby PVM instructions retrieve their arguments from the stack, just to be placed back onto the stack after execution of the instruction. The Python compiler generates PVM code (a *.pyc* file) for the Python VM to execute. The CPython interpreter resembles a classic interpreter with a straightforward algorithm (Aycock, 1998, pp.2–4):

1. Firstly, the opcode of a VM instruction is fetched, along with any necessary arguments.
2. Secondly, the instruction is then executed.
3. Finally, steps 1-2 are repeated till no more opcodes can be fetched. This is done by raising an exception when an invalid (empty) opcode is found.

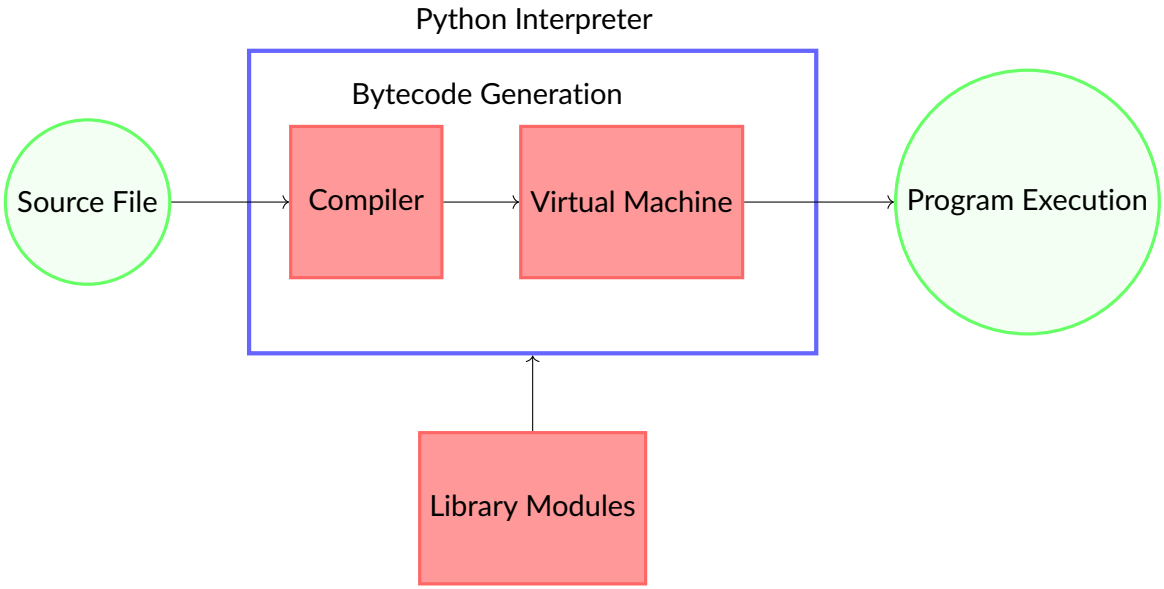


Figure C.1: Python Code Execution

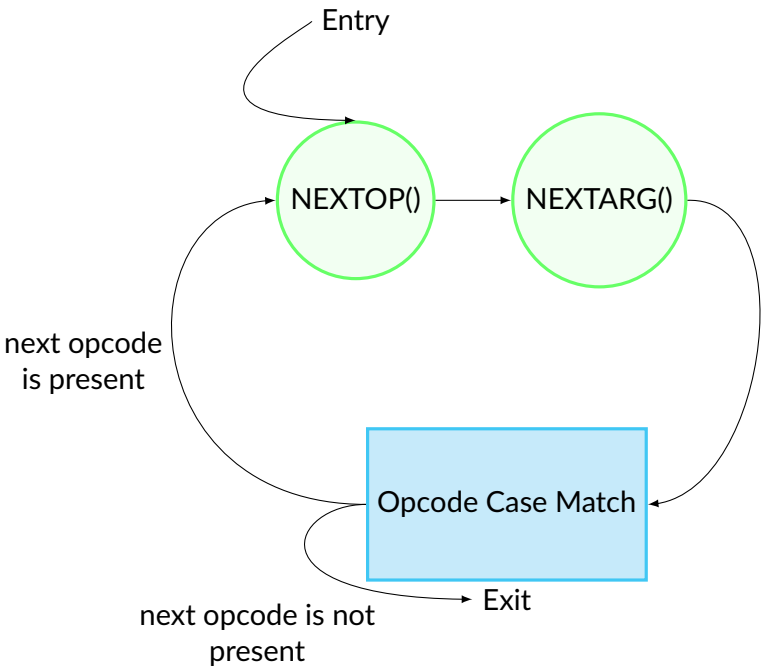


Figure C.3: VM Innards

This simple algorithm is known as the CPython Evaluation Loop. The evaluation loop is for-

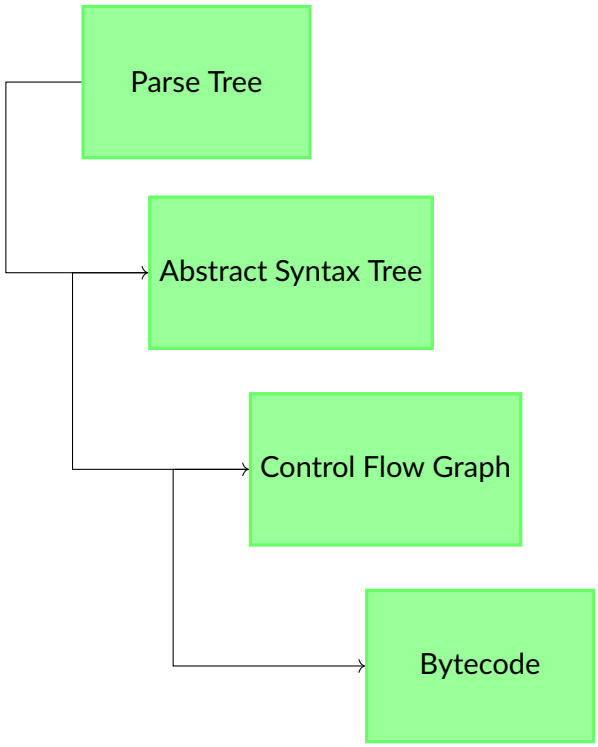


Figure C.2: Compiler Innards

mulated as shown in Listing C.1[1].

Evaluation is computed frame by frame (See **Frames**), with what is essentially a (very long) switch statement; reading every opcode and delegating accordingly.

C.1.2 Stacks

A stack data structure is a dynamic structure that operates with a LIFO policy (Cormen et al., 2009). Since CPython does not directly interact with the hardware for compilation, it makes both the call stack and stack frames rely on the PVM. In CPython, there is one main stack that the PVM requires for proper functionality; the call stack. The other two stacks (Value Stack and Call Stack) are essential for the proper computation of any variables that there are in the frame (See **Frames**). Most instructions manipulate the value stack and the call stack (Bennett, 2018). The nature of CPython stacks is visualised in Figure C.5

Call Stack

The call stack contains call-frames (See **Frames**). This is the main structure of the running program. A function call results in a pushed frame onto the call stack, whilst a return call results in a pop of

[1]Actual code differs from what is presented. The source code has been edited as to be more readable and concise.

```
      for(**indefinite condition**){
        oparg=null;
        opcode = NEXTOP();
        if(ARGUMENT_PRESENT(opcode)){
          oparg = NEXTARG();
        }
        switch(opcode){
          case **opcode_name**:
            manipulate stack & set variables accordingly
          ...
          ...
          ...
          default:
            raise error
        }
      }

```

Listing C.1: Evaluation Loop

the function frame off of the stack (Conrad, 2010). A visual representation of a call stack is shown in Figure C.4. In this figure, a sample script can be seen run, step by step, showing the frames being pushed onto the call stack and popped from the call stack. The first frame pushed onto the frame stack is inevitably called the `__main__` call frame.

Value Stack

This stack is also known as the evaluation stack. It is where the manipulation of the object happens when evaluating object-manipulating opcodes. A value stack is found in a call-frame, implying bijectivity. Any manipulations performed on this stack (unless they are namespace related) are independent of other stacks and do not have the permissions to push values on other value stacks.

Block Stack

The block stack keeps track of different types of control structures, such as; loops, try/except blocks, and with blocks. These structures push entries onto the block stack, which are popped whenever exiting the said structure. The block stack allows the interpreter to keep track of active blocks at any moment

```
def foo():
  print("Hello")

def intermediary():
  foo()

def start():
  intermediary()

start()

```

Listing C.2: Python script

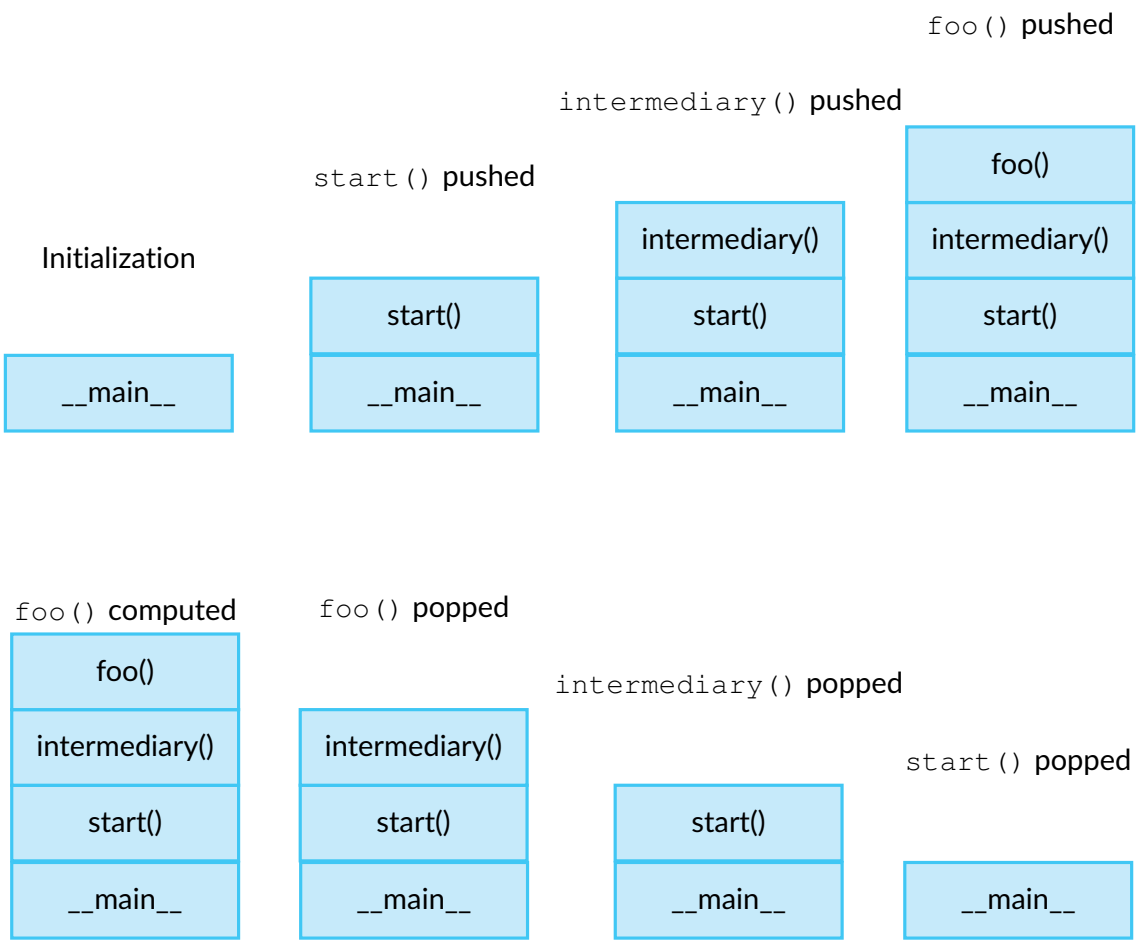


Figure C.4: Simulation of Call Stack running Listing C.2

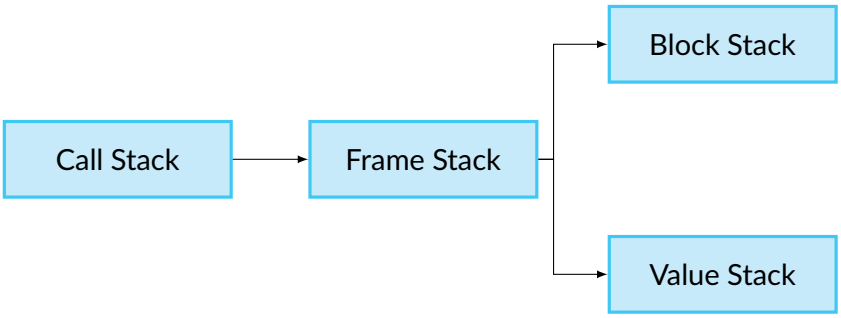


Figure C.5: Overview of CPython stacks

C.1.3 Frames

A frame (call-frame) is an object which represents a current function call (subprogram call); more formally referred to as a code object. It is an internal type containing administrative information useful for debugging and is used by the interpreter (Van Rossum and Drake Jr, 1994, pp.18–19). Frame objects are tightly coupled with the three main stacks (See **Stacks**) by which every frame is linked to another. Every frame object has two frame-specific stacks; value stack (See **Value Stack**) and the block stack (See **Block Stack**). Frames are born from function calls, and die when that function is returned.

Frame Attributes

Along with the properties mentioned above, a frame object would have the following attributes:

f_back points to the previous stack frame object (return address).

f_code points to the current code object (See **Code Objects**) being executed, in the current frame.

f_builtin points to the builtin symbol table.

f_globals points to the dictionary used to look up global variables.

f_locals points to the symbol table used to look up local variables.

f_valuestack this holds the pointer, pointing to the value of the top of the value stack.

f_lineno this gives the line number of the frame.

f_lasti this gives the bytecode instruction offset of the last instruction called.

f_blockstack contains the block state, and block relations.

f_localsplus is a dynamic structure that holds any values in the value stack, for evaluation purposes.

These attributes are retrieved from the declaration found in *./Include/frameobject.h*.

Frame Stack

The stack frame is a collection of all the current frames in a call-stack-like data structure (See **Call Stack**). A frame is pushed onto the stack frame for every function call (every function has a unique frame) as shown in Figure C.6.

The stack frame contains a frame pointer which is another register that is set to the current stack frame. Frame pointers resolve the issue created when operations (pushes or pops) are computed on the stack hence changing the stack pointer, invalidating any hard-coded offset addresses that are computed statically, before run-time (Bacon, 2011). With frame pointers, references to the local variables are offsets from the frame pointer, not the stack pointer.

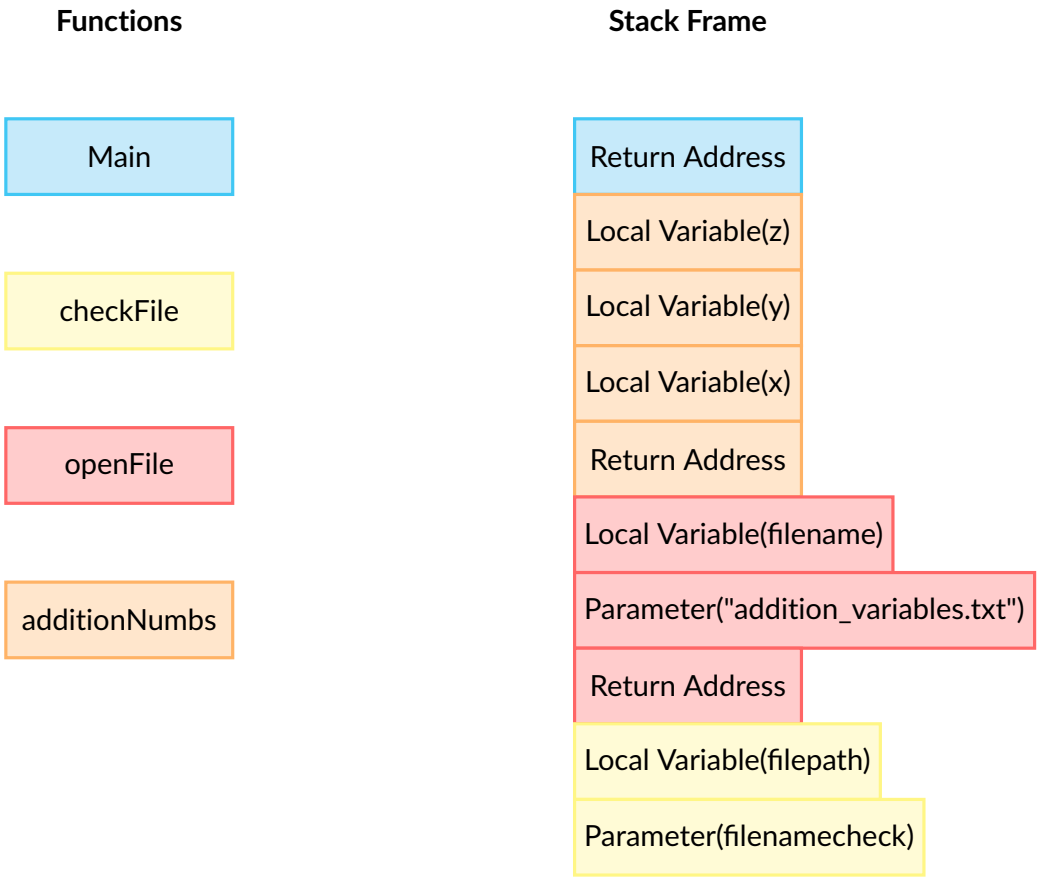


Figure C.6: Stack frames example

C.1.4 Code Objects

A code object is a low-level detail of the CPython implementation. When parsing Python code, compilation creates a code object for processing on the PVM. A code object contains a list of instructions directly interacting with the CPython VM; hence coined a low-level detail. Code objects are of the type `PyCodeObject`, with each section of the code object representing a chunk

of executable code that has not been bound to a function (Foundation, 2022b). The structure of the type of these code objects change throughout different CPython versions, thus there is no set composition (Foundation, 2022b). For reference, the source code in Listing C.1.4 produces the code object displayed in Listing C.1.4, following the standard convention shown in Listing C.1.4.

```
    <code object addition_numbers at 0x1047f10b0, file "
filepath", line 3>
```

Listing C.3: Code object of Listing C.1.4

```
    code object <functionName> at <address>, file <path>, line
<firstLineNo.>
```

Listing C.4: Code object standard convention

Disassembler

Code objects are expanded by using the `dis` module in Python. This module contains several analyses functions which; all of which directly convert the input code object into the desired output (Foundation, 2022a). The function that is of a particular interest in this paper is the `dis.dis` function which disassembles a code object into its respective bytecodes, alongside other relevant information, as seen in Figure C.1.4. When applying the analysis function `dis.dis`, the disassembled code object takes the following format for every instruction:

```
    <lineNumber><label><instructionOffset><opname><opargs><var>
```

Listing C.5: Dissasembled instruction convention

It is interesting to note that the value for `opargs` is computed in little-endian order. Typically, as is shown in ... ,the arguments associated with the instructions are used for specific stack manipulations.

```
1  import dis
2
3  def addition_numbers(x,y):
4      z=x+y
5      return z
```

```
6
7                 dis.dis(addition_numbers)
8
```

Listing C.6: Python source code

```
        4      0 LOAD_FAST          0(x)
               2 LOAD_FAST          1(y)
               4 BINARY_ADD
               6 STORE_FAST         2(z)

        5      LOAD_FAST          2(z)
               10 RETURN_VALUE
```

Listing C.7: Disassembly of Listing C.6

Bytecode

Bytecode is a form of portable code (p-code) executed on a virtual machine. The introduction of the concept of bytecode came about when a generalized way of interpreting complex programming languages was required to simplify information structures, characterizing them in their essentials (Landin, 1964). This ideology gave birth to portable code; a generalized form of code, that can cross-compile, assuming that the Virtual Machine which interpreted the p-code was compatible with the native machine architecture.

In CPython V3.10, there are over 100 low-level bytecode instructions (Foundation, 2022a). The classification of bytecode operations is defined below:

- Unary instructions.
- Binary instructions.
- Inplace instructions.
- Coroutine instructions.
- Argument instructions.
- Miscellaneous instructions.

C.1.5 Execution of Code Objects

The evaluation stage firstly makes use of the public API `PyEval_EvalCode()` (Rossum et al., 2022, lines 716–724), which is used for evaluating a code object created at the end of the compilation stages (Figure C.2). This API constructs an execution frame from the top of the stack by calling `_PyEval_EvalCodeWithName()`. The first execution frame constructed must conform to these requirements:

1. The resolution of keyword[2] and positional arguments[3].
2. The resolution of *args[4] and **kwargs[4] in function definitions.
3. The addition of arguments as local variables to the scope (A scope is the membership of a variable to a region).
4. The creation of Co-routines and Generators (Tismer, 2000, pp.2–3).

Code execution in CPython is the evaluation and interpretation of code object. Below, we delve into a more detailed description of frame objects; their creation, and execution (Shaw, 2022).

Thread State Construction

Prior to execution, the frame would need to be referenced from a thread. The interpreter allows for many threads to run at any given moment. The thread structure that is created is called `PyThreadState`.

Frame Construction

Upon constructing the frame, the following arguments are required:

_co A `PyCodeObject` (code object).

globals A dictionary relating global variable names with their values.

locals A dictionary relating local variable names with their values.

It is important to note that there are other arguments that might be used but do not form part of the basic API, thus will not be included.

[2] A keyword argument is a value that, when passed into a function, is identifiable by a specific parameter name, such as variable assignment.

[3] A positional argument is a value that is passed into a function based on the order in which the parameters were listed during the function definition.

[4] *args and **kwargs allow multiple arguments to be passed into a function via the unpacking (*) operator.

Keyword & Positional Argument Handling

If the function definition contains a multi-argument keyword argument [4], a new keyword argument dictionary (`kwdict` dictionary) is created in the form of a `PyDictObject`. Similarly, if any positional arguments[4] are found they are set as local variables.

The dictionary that was created is now filled with the remaining keyword arguments which do not resolve themselves to positional arguments. This resolution comes after all the other arguments have been unpacked. In addition, missing positional arguments [5] are added to the `*args`[4] tuple. The same process is followed for the keyword arguments; values are added to the `**kwargs` dictionary and not a tuple.

Final Stage

Any closure names are added to the code object's list of free variable names, and finally, generators and coroutines are handled in a new frame. In this case, the frame is not pre-evaluated but it is evaluated only when the generator/coroutine method is called to execute its target.

C.1.6 Execution of Frame Objects

The local and global variables are added to the frame preceding frame evaluation; handled by `_PyEval_EvalFrameDefault()`.

`_PyEval_EvalFrameDefault()` is the central function which is found in the main execution loop. Anything that CPython executes goes through this function and forms a vital part of interpretation.

[5] Positional arguments that are provided to a function call, but are not in the list of positional arguments.