Semantic Analysis of Programs using Graph Neural Networks

Aayan Kumar

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-269
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-269.html

December 16, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank Prof Koushik Sen for his advice and guidance on this project, and Shadaj Laddad for the design and implementation of the instrumentation framework. I also thank Moustafa AbdelBaky, Prof Alvin Cheung, Sahil Bhatia, Jathushan Rajasegaran, and Aniketh Reddy for their helpful feedback and discussions. I also thank Arpita Banerjee for her unwavering support throughout the duration of the project.

Semantic Analysis of Programs using Graph Neural Networks

by Aayan Kumar

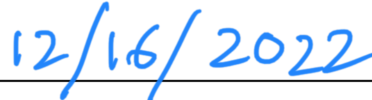
Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Koushik Sen
Research Advisor



(Date)

* * * * * * *

Professor Alvin Cheung
Second Reader

Dec 15, 2022

(Date)

Semantic Analysis of Programs using Graph Neural Networks

AAYAN KUMAR, University of California, Berkeley, USA

We present a framework that given a piece of code and an input to it, can recommend the NumPy API which is semantically closest to the input code. Our approach is based on executing the provided code on the given input and creating a representation capturing the execution of the code, rather than relying on the syntax. The representation is expressed in the form of a graph which would be analyzed further using a Graph Convolutional Network for predicting the relevant API.

Existing work in semantic program analysis invariably depends on some analysis of source code or Abstract Syntax Trees and similar static representations, which is brittle as it can be sensitive to the style of writing of programs, and can get fooled using simple semantic-preserving transformations, like different variable names, dead code and so on. The representation we use is agnostic of the style of writing the code and hence generalizes across different implementations from the wild of the same functionality, eliminating the need to mine different implementations of the same method.

1 INTRODUCTION

Software has made its way into almost every sphere of life imaginable over the course of the last few decades. With the increase in the amount of software, so has the demand for tools that enable developers to write code more easily and correctly while ensuring the code is performant. That has led to the development of a concept known as Application Programming Interface (API). An API is a library that implements some functionality, but most of the details are abstracted away from a developer who wants to use those functionalities. All that is exposed is a set of function signatures that the user can use to interact with the library.

The concept of an API has several advantages:

- The implementation of the API can be very highly optimized and can use optimizations that may not be known by non-experts in the domain
- The user need not know about the details of the internals of the APIs to be able to use it. (However, some understanding would be useful for the user to use the APIs in an effective manner which may not be obvious.)

Due to the design of an API, all a user needs to worry about is how to use the API method. However, there are thousands of different libraries which expose an API, and often there are multiple libraries that solve a similar problem. The vast availability has made it possible for users with similar but slightly different use cases to find a library that would work for them, like NumPy and TensorFlow, both of which can be used to express tensor computations efficiently, with a small difference in the devices supported by TensorFlow. Even in one API library, there can be hundreds or even thousands of different function signatures, and each signature can have dozens of parameters that can be tuned. Hence, even using the APIs becomes a challenging task due to the sheer volume of different APIs available.

There has been a lot of work in creating tools that make it easier for developers to use popular APIs, like [9, 10]. These tools tend to rely on specifications expressed in the form of formal conditions, or input/output examples. Depending on the target domain, formal conditions are exact but can be extremely tedious to write and I/O examples are easy to write but are severe under-approximations requiring several iterations to work properly. In some cases, it may even be hard to specify I/O examples itself.

Apart from APIs, there has also been a lot of work on tools that help make programming easier by recommending code snippets/variable names/function names depending on the code snippet already provided by the user [3, 31]. Such tools typically rely on analysis of source code, or static analysis to predict the properties of programs. However, this approach

Author's address: Aayan Kumar, aayan@berkeley.edu, University of California, Berkeley, USA.

is prone to get misguided by examples that could be slightly out of distribution, examples being code transformations that otherwise do not affect the semantics of the program, but affect the performance of these syntax-guided tools.

In addition, static analysis tools work well when the program has enough static information for the tools to exploit. However, for dynamically typed languages like Python, these tools are not that effective as Python lacks a static type system which makes it hard for these tools to work effectively. Recent articles [1] suggest that the most popular languages being used are dynamically typed languages, hence it is important to explore techniques which would work well on these languages.

In this Technical Report, we study the NumPy API, an API which is commonly used to specify tensor computations used in Deep Learning. We consider the problem of predicting a NumPy API which is closest to a given code snippet. We use dynamic analysis, to account for the fact that since Python is dynamically typed, static analyses might not be effective, but dynamic analysis would give us access to a lot more information than static techniques would do. We use code as our specification, as most developers who use NumPy are at least novice Python programmers for whom it would be straightforward to provide a code snippet expressing their desired behavior, rather than specifying I/O examples, which can be hard to specify for a tensor or formal specifications which are anyway hard to write for scalar inputs.

Our contributions in this report can be summarized as follows:

(1) We propose a graph representation that is generated from the execution of a program on a valid input, rather than using a representation that is generated purely from the text of the program.
(2) We propose a framework that given a code snippet and a valid input, generates the graph representation and processes it to predict a NumPy API closest to the code snippet.
(3) We demonstrate that the proposed dynamic representation is able to generalize to code not seen at all in the training dataset, and is immune to semantics preserving changes

We release our codebase for future research[2]. The rest of the report is organized as follows: Section 2 covers some basic preliminaries which are used in our framework. Section 3 walks through a simple example showing how our framework works end-to-end. Section 4 covers in-depth technical details used in our implementation. Section 5 discusses evaluations of our framework, Section 6 discusses the limitations in our current approach and discusses future directions of work. Section 7 discusses related work and Section 8 concludes.

2 PRELIMINARIES

In this section, we discuss a few preliminary topics and ideas that will be used in our framework.

2.1 Symbolic Execution

Symbolic execution [11] is a type of execution of a given program, in which an interpreter goes through the instructions of a program in the same order as they would be executed in a typical concrete execution scenario. However, instead of executing the program with concrete inputs, the interpreter assigns symbolic values to each variable which enables the interpreter to construct expressions of each variable of the program in terms of the original symbolic values.

[1]https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/
[2]The code can be found at github.com/aayan636/semantic-analysis-python

Consider an example code as follows:

```
def func(a, b):
    c = a + b
    return c
```

When executing the function *func* concretely, we would supply the function with concrete values for a and b, say 3 and 4 respectively. Concrete execution of the function would return the value 7.

However, while executing *func* symbolically, the interpreter would supply the function with symbolic values, ‘a’ and ‘b’, concrete operations (like + in the example) would be performed symbolically, and the variable c would be assigned the expression ‘a’ + ‘b’ (instead of a concrete integer value) which is what would be returned.

Symbolic execution can be used to evaluate the mathematical expressions which relate different variables of a program. We can apply the same technique on branch conditions in the program to obtain ‘path conditions’, a set of constraints on the input symbolic values, solving which would provide us with a set of values that would force the program along a particular set of branches. This technique is used extensively in software testing [39, 40] to find out sets of inputs that would force the user program to cover different branches, which typically involves executing the program under test multiple times.

In our framework, we create a representation that combines symbolic expressions and path conditions into a unified graph representation (an example shown in Section 3, and formally described in Section 4), based on the execution of the user-provided code on a single valid input. The graph thus created is analyzed using the help of a neural network described in Section 2.2 to predict what algorithm has been implemented in the user code, and what API would be semantically equivalent to it.

2.2 Graph Neural Networks

Many machine learning applications rely on some regularity in the structure of the data, like images [30], text [23, 44], videos, etc. However, there are many applications like molecular analysis [24], and program analysis [5] where the data may not conform to a rigid structure like image height and width. However, in such applications, it is normally possible to represent the irregular data as a graph: a set of nodes and edges signifying relations between two nodes.

Graph Neural Networks [36] are a class of neural networks that are designed to operate on such irregular data which can be expressed in the form of graphs. One common type of Graph Neural Networks is the Graph Convolutional Network (GCN) [37], which tries to generalize the inference in a typical Convolutional Neural Network [30].

Inference in a GCN can be understood as a message-passing procedure. Each node, each edge, and the graph as a whole has a hidden state assigned to it, typically an N-dimensional vector initialized using zeros. In our framework, N = 1. For example in Figure 1, the node $N1$ has a vector h_{n1} associated with it, the edge $E2$ has a vector h_{e2} associated with it, and the entire graph has a vector h_G associated with it. At each timestep:

- For each node, the hidden states of the immediate neighboring nodes, the immediately connected edges, and the global graph state are individually accumulated and concatenated, and combined with the state of the given node according to the following equation:

$$h'_n = f\left(W^1_{node} \times \frac{\sum_{v \in Nbr(n)} h_v}{|Nbr(n)|} + B^1_{node}\right) \oplus f\left(W^2_{node} \times \frac{\sum_{v \in Conn(n)} h_v}{|Conn(n)|} + B^2_{node}\right) \oplus f\left(W^3_{node} \times h_G + B^3_{node}\right) \quad (1)$$

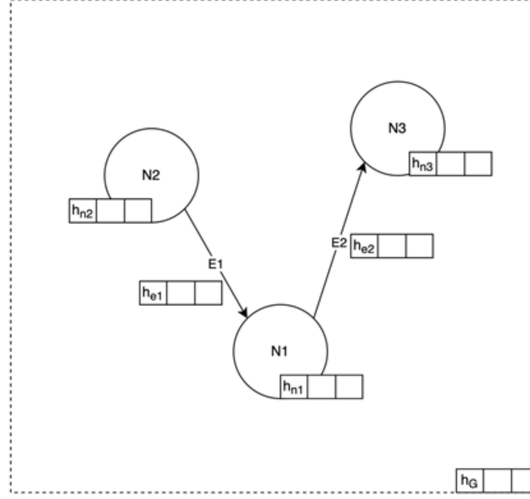


Fig. 1. Example graph and metadata used in a GCN

In this equation, h_n is the old hidden state of node n while h'_n is the new hidden state after the update. W^1_{node}, W^2_{node}, W^3_{node}, B^1_{node}, B^2_{node}, B^3_{node} are learned weight and bias parameters, $Nbr(n)$ denotes the set of nodes connected to node n, $Conn(n)$ denotes the set of edges coming into or going out of node n. h_G represents the global hidden state of the graph. f refers to any non-linearity used in Neural Networks (like *tanH*, *ReLU*, *sigmoid* etc.) and $\oplus$ refers to the concatenation operator.

- For each edge, the hidden states of the attached nodes and the global graph state are individually averaged and concatenated, and combined with the state of the given edge according to the following equation:

$$h'_e = f\left(W^1_{edge} \times \frac{\sum_{\eta \in Node(e)} h_\eta}{2} + B^1_{edge}\right) \oplus f\left(W^2_{edge} \times h_G + B^2_{edge}\right) \tag{2}$$

In this equation, h_e is the old hidden state of the edge e, while h'_e is the new hidden state. W^1_{edge}, W^2_{edge}, B^1_{edge}, B^2_{edge} are learned weight and bias parameters, and $Node(e)$ refers to the nodes attached directly to edge e. This formulation assumes simple edges which only connect two nodes and no hyper-edges (edges between edges) although the inference framework can be generalized to handle such a scenario.

- For the global state, the states of all the edges and nodes are individually averaged and concatenated with the previous global state according to the following equation:

$$h'_G = f\left(W^1_{glb} \times \frac{\sum_{\nu \in G.nodes} h_\nu}{|G.nodes|} + B^1_{glb}\right) \oplus f\left(W^2_{glb} \times \frac{\sum_{\nu \in G.edges} h_\nu}{|G.edges|} + B^2_{glb}\right) \oplus f\left(W^3_{glb} \times h_G + B^3_{glb}\right) \tag{3}$$

In this equation, h_G is the old hidden state of the graph, while h'_G is the new hidden state. W^1_{glb}, W^2_{glb}, W^3_{glb}, B^1_{glb}, B^2_{glb}, B^3_{glb} refer to the learned weight and bias parameters, G refers to the entire graph, $G.nodes$ refers to the set of all nodes in the graph, and $G.edges$ refers to the set of all edges in the graph.

One step described above corresponds to a 1-hop message passing, which is communication between two immediately connected nodes. The inference step (all 3 bullets above together correspond to one step) can be repeated n times

to simulate a n-hop message passing architecture. As we shall see in the subsequent sections, 1-hop turns out to be sufficient in our problem setting.

For problems where one has to classify individual nodes in the graph (for example, a social network graph where one wants to predict which node (users) might be operating a business), one can attach a fully-connected layer (a layer that performs a matrix multiplication and an addition on the input feature vector) after the hidden state of each node which would predict the final class for each node. However, as we shall see in subsequent sections, our problem involves classifying the entire graph, hence we attach a fully-connected layer after h_G, the global hidden state as it captures information from all nodes in the graph, and we use that for predicting the properties of the graph.

3 WORKING EXAMPLE

In this section, we walk through a simple example to demonstrate how our framework works end-to-end from accepting a user code expressing a numerical computation to recommending a NumPy API which would resemble it most closely.

3.1 Generating Graph from Execution

Consider a toy example of a user code snippet as shown:

```
def func(a, b, c):
    a, b = b + c, a
    return a, b
```

Our framework would first compile the source code down to Python bytecode, which would look like the following (the actual generated bytecode might differ according to the Python version and interpreter):

```
1. LOAD_FAST "a"
2. LOAD_FAST "b"
3. LOAD_FAST "c"
4. BINARY_ADD
5. STORE_FAST "a"
6. STORE_FAST "b"
```

We work on the Python bytecode instead of the source. Python provides a lot of ways to perform the same operation, like decorators, multiple assignments, fancy indexing (indexing into a collection object using non-integral values, like $a[1:4]$), monkey patching, etc. It is easier to reason about the code at the bytecode level as compared to the source code level, as bytecode does away with a lot of the syntactic sugar in Python. Existing work [4, 5] invariably deals with performing some analysis on the source code level, like predicting the program behavior by just looking at the source as compared to actually executing the code. To that end, they use a representation that is close to the source, like the source code text, or Abstract Syntax Trees, Data/Control Flow Graphs, etc. However, as we have seen previously in Section 1 and shall see in Section 5, static analyses are vulnerable to semantics-preserving syntactical modifications. This motivates our aim to perform analysis based on the execution of the code. Therefore, we would like to create a representation that is constructed from the program runtime rather than program text.

To this end, our framework executes the generated bytecode symbolically and concretely to generate a graph that captures what happens when a program is actually run. Using the symbolic execution, we construct a graph that

roughly captures how different memory locations accessed by a program interact with each other and how the values are dependent on each other, which roughly resembles symbolic execution as described in Section 2.

We walk through step by step as our framework executes the code snippet *func* above. When the framework executes the code, it mimics the Python interpreter [43] which uses a stack machine to perform computations. Unlike a concrete executor, the stack machine would consist of symbolic values. It also maintains a set of symbolic values for each of the function argument variables. The state is represented in Figure 2.

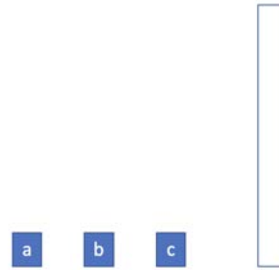


Fig. 2. Starting state of the symbolic execution engine. The blocks ‘a’, ‘b’, ‘c’ are symbolic values for the function arguments, while the vertical line resembles the compute stack

After Line 1 is executed, the runtime pushes a symbolic value onto the symbolic compute stack corresponding to a. It also traces a dependency from the original symbolic value ‘a’ to the value that has been pushed onto the stack. This behavior mimics what happens in a concrete Python interpreter. The state of the executor is given in Figure 3. The number 0 in the symbolic variable on the stack corresponds to the version of the original symbolic variable (If ‘a’ is overwritten 5 times, then the next time it is read onto the stack, it would have a version 5).

Fig. 3. State after executing Line 1. The > represents the instruction just executed.

Similarly, after Lines 2 and 3 are executed, the runtime pushes symbolic values for ‘b’ and ‘c’ respectively and records the appropriate dependencies, as can be seen in Figure 4.

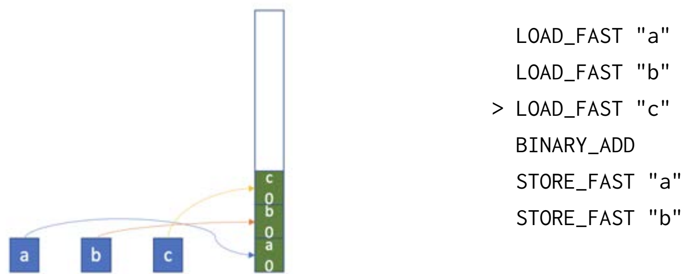


Fig. 4. State after executing Lines 2 & 3. The > represents the instruction just executed.

After Line 4 is executed concretely, which is *BINARY_ADD* which is a binary operation, the concrete executor pops the top two elements from the stack, adds them, and pushes the result back onto the stack. To mimic this behavior, the symbolic executor also pops the top two symbolic elements off and pushes a new symbolic variable corresponding to a new temporary variable 't', which also stores information about the operation used to create the symbolic variable, in this case addition. This can be seen in Figure 5

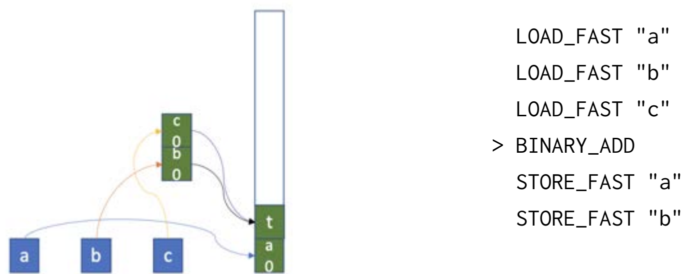
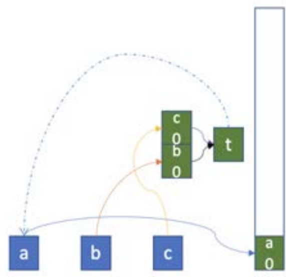
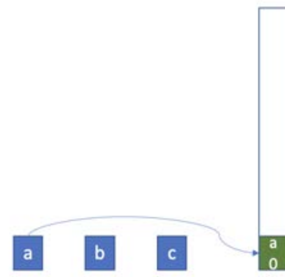


Fig. 5. State after executing Line 4. The > represents the instruction just executed.

When Line 5 is executed concretely, we actually have to store the new updated concrete values back to the original variable. In the symbolic execution, that is equivalent to assigning the temporary symbolic variable 't' back to the symbolic variable 'a', as shown in Figure 6. Instead of overwriting the symbolic value 'a', we simply remove all of the symbolic values 'b0', 'c0' and 't' (as shown in Figure 7), and then given the tracked dependencies between the symbolic values as seen in Figure 6, construct a graph which captures them, as shown in Figure 8. In this example, the new value of 'a' is generated by adding 'b' and 'c', which is what is captured in the graph in Figure 8.






```
  LOAD_FAST "a"
  LOAD_FAST "b"
  LOAD_FAST "c"
  BINARY_ADD
> STORE_FAST "a"
  STORE_FAST "b"
```

Fig. 6. State while executing Line 5

Fig. 7. State after executing Line 5

Note in Figure 8, we have two nodes corresponding to ‘a’, ‘a0’ and ‘a1’, corresponding to the two different versions of values held by ‘a’ as the program executes. ‘a0’ corresponds to the first version which was input to the function, while ‘a1’ refers to the updated value of ‘a’, and subsequent versions would be labeled accordingly.

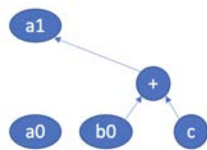


Fig. 8. Dependency graph after executing Line 5

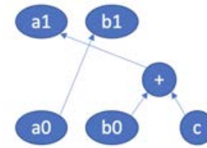
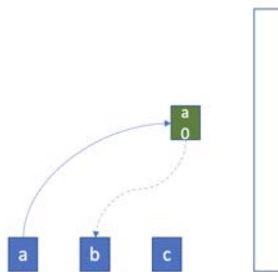


Fig. 9. Dependency graph after executing Line 6

After executing Line 6 concretely, the topmost value from the compute stack is popped and assigned to variable ‘b’. Symbolically, we pop the topmost symbolic stack value and create a dependency from the symbolic value of ‘b’ to the symbolic value of ‘a’. The dependency graph is captured in Figure 9, which shows that the updated value of ‘b’ has a dependency on version 0 of the value of ‘a’.




```
  LOAD_FAST "a"
  LOAD_FAST "b"
  LOAD_FAST "c"
  BINARY_ADD
  STORE_FAST "a"
> STORE_FAST "b"
```

Fig. 10. State after executing Line 6. The > represents the instruction just executed.

While generating the graph, we maintain all nodes corresponding to all intermediate values of each memory location used in the program. However, this can cause the size of the generated graphs to be very large. To mitigate this, we

perform a set of reductions where we remove nodes corresponding to intermediate states in the program, and only capture the states corresponding to the input variable's first versions and the final versions of the variables which are returned. These reductions are described further in Section 4. The formalism behind how the graph is constructed is also described in Section 4.

3.2 Using the generated Graph for semantic analysis

The generated graph, as shown above, essentially captures how the program transforms its inputs into outputs. However, since the graphs are generated from an execution of the program, it is possible that on different inputs, the generated graphs might be different, especially in the example of programs that operate on arrays/lists of varying sizes. An example would be a function that returns a sorted copy of an input list.

One approach could be to perform reductions on the graphs (reducing the number of nodes in the graph, similar to some trivial reductions which we use anyway as shown in 4.3.2) using manually hand-coded rules. However, coming up with these rules would be a difficult and tricky task, with no guarantees whether the hand-written rules would generalize to different implementations of the same algorithm. Hence, instead of using handwritten rules, we use a machine learning model to automatically learn these rules using which we can analyze the generated graphs and predict the properties of the programs under test. We describe the machine learning models used in detail in Section 5.1.

4 TECHNICAL DETAILS

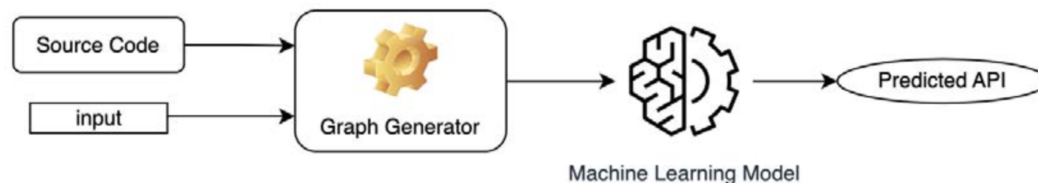


Fig. 11. Overall architecture of Inference pipeline

4.1 Overall Architecture

The overall architecture of the inference pipeline is shown in Figure 11. As an input, our framework expects a code snippet, as well as an input on which the given code snippet can be executed. The framework executes the code using the provided input and creates the graphical representation of the program runtime, which is formally described in Section 4.3. The generated graph is fed through a machine learning model as described in Section 4.3.3 which predicts the NumPy API closest to the behavior of the code snippet provided by the user. We describe how the machine learning model is trained in Section 4.4.

4.2 Instrumentation Framework

As described in Section 3, our framework first compiles the code provided by the user into Python bytecode [15]. However, to actually perform the symbolic computation to trace dependencies between variables required for our

graph, we need to extract information about different memory locations from the program runtime, which we do by means of instrumenting[3] the user code.

Given a bytecode instruction in the original user code, we add a few instructions before and/or after it to extract the state of the concrete computation stack, inputs and outputs of the currently executed bytecode instruction, and some metadata. The process is carried out in a way to not affect the execution of the user code; the result of executing the program with or without the instrumentation would result in the same concrete output, assuming no non-determinism in the original code.

4.3 Construction of Graph

4.3.1 Formal Development. We formally describe the construction of the graph in Figure 12 and Figure 13. To this end, we consider a language in Figure 12 which is a subset of the language of Python bytecode as generated by the CPython compiler [43]. The language is fairly straightfoward: a program is composed of multiple functions. Each function is a sequence of multiple instructions, which encode different operations on the computation stack like loading from memory to compute stack or storing from compute stack to memory, performing arithmetic on the top N elements on the compute stack or performing other manipulations.

$$
\begin{array}{rcl}
P & ::= & (f;)* \\
f & ::= & (I,)* \\
I & ::= & LOAD_NAME(id) \mid STORE_NAME(id) \mid BINARY_OP(op) \mid BINARY_SUBSCR \mid BINARY_LESSTHAN \mid \\
 & & CALL_FUNCTION(id) \mid RETURN \mid POP_TOP \mid DUP_TOP \mid ROT_TWO \mid STORE_SUBSCR \\
op & ::= & ADD \mid SUBTRACT \mid MULTIPLY \mid DIVIDE \mid POWER \mid XOR \mid AND \mid OR \mid MODULO \\
id & ::= & (a-z|A-Z|0-9)*
\end{array}
$$

Fig. 12. Grammar of the source language

In the operational semantics described in Figure 13, we use several different symbols representing the program state, whose meaning is described as follows:

- Call Stack C_S: list of all function frames currently in scope. It is represented through a list of frames. Its type is denoted as $[func]$, where $[T]$ represents a list of type T.
- Compute Stack K: represents the computation stack used by the Python runtime to hold the intermediate computed values. This stack holds only concrete values. It's type is $[addr \cup value]$
- Environment E: corresponds to the variables in scope in the current context (represented by the function call stack). We only consider variables local to the current scope and no global variables. Its type is $f \times id \mapsto addr$. As a simplifying assumption, this map cannot be updated; once a name is declared in the current scope, it cannot be deleted (notice the language does not contain any *delete* instruction).
- Memory M: maintains the objects assigned in the heap memory. This only corresponds to concrete execution objects. Its type is $addr \mapsto value \cup addr$.

[3]The author would like to thank Shadaj Laddad for working on the original version of the instrumentation logic and framework which was modified and used for this technical report.

The above symbols correspond to the state which is maintained by the original CPython interpreter and only correspond to the concrete execution state. The following state variables are maintained by our framework to create the graphs from the execution:

- Symbolic Compute Stack Z: Contains symbolic values which correspond to the concrete compute stack K. Its type is $[SymbolicValue]$.
- Symbolic Memory Store N: Symbolic counterpart for concrete Memory M. Its type can be interpreted as $(func \times id) \mapsto SymbolicValue$.
- Dependency List D: Keeps track of all observed dependencies between different $SymbolicValue$s from which the execution graph can be created. Effectively this contains all the edges of the graph.
  Its type is $[Dependency]$ which is equivalent to $[(SymbolicValue, SymbolicValue, Optional(op))]$, or a list of 2 symbolic values followed by an optional operator. For simple load/store instructions, the op would be ϕ, as there is no special operator used in the creation of this dependency, whereas for binary operations like add or subtract, op would contain '+' or '-' respectively. The convention used in Figure 13 assumes for a tuple (a, b, op), a is dependent on b, or that b's concrete value was used in computing the concrete value for a.

$addr$ refers to a concrete memory address type, $value$ refers to any concrete value which is not an address, $func$ refers to a function object, id is any name for a variable used in the bytecode. $new(T)$ refers to a new instance of class T.

$SymbolicValue$ is a compound type equivalent to $version \times (offset \mapsto SymbolicValue)$: each $SymbolicValue$ object keeps track of the current version of the symbolic variable, or how many times the concrete memory location has been changed, and also keeps a track of a mapping from $offset$ (which could be int for lists, str for maps, etc.) to other $SymbolicValue$s, which enable it to keep track of collection objects associated to a variable like lists, dicts, etc.

In Figure 13, the judgment C, K, Z, E, M, N, G ⊢ INST → C', K', Z', E', M', N', G' is to be read as "Under the runtime environment comprising of call stack C, compute stack K, symbolic compute stack Z, local variables E, concrete memory store M, symbolic memory store N, and dependency graph G, executing instruction INST results in a new environment having call stack C', compute stack K', symbolic compute stack Z', local variables E', memory store M' and symbolic memory store N', and generates a new dependency graph G' ".

4.3.2 Graph Simplification. Simply following the rules in Figure 13 would generate graphs that capture each operation through a separate node in the generated graph, even the trivial ones like loading a variable onto the stack, or moving an object from one memory location to another, which has a few disadvantages:

- The generated graph size becomes very large, even for a simple operation like adding two matrices having fewer than 10 elements would generate a graph with thousands of nodes.
- Different implementations of the same algorithm might result in different graphs (when executed on the same input) making the problem of identifying the implemented algorithm no easier than it would be when using a graph created directly from the source code.

In light of these observations, we perform a post-processing on the graph where we remove unnecessary nodes from the graph which convey no additional information, which is captured through algorithm 1 and algorithm 2.

In algorithm 1 and algorithm 2, the following symbols are used as helper methods:

- `GetNodesFromGraph`(G): returns the set of nodes in graph G.
- `GetEdgesFromGraph`(G): returns the set of edges in graph G.

$$\frac{n = new(SymbolicValue)}{C_S : f, K, Z, E, M, N, G \vdash \textbf{LOAD_NAME}(id) \rightarrow C_S : f, K : M(E(f, id)), Z : n, E, M, N, G \cup [(n, N(f, id), \phi)]}$$

$$\frac{N(f, id) = (w, rSym1)}{\begin{array}{c} C_S : f, K : res, Z : (v, rSym), E, M, N, G \vdash \textbf{STORE_NAME}(id) \rightarrow \\ C_S : f, K, Z, E, M[E(f, id) \mapsto res], N[(f, id) \mapsto max(v, w) + 1, rSym], G \cup (N[(f, id)], (v, rSym), \phi) \end{array}}$$

$$\frac{z = (max(v_a, v_b) + 1, n), n = new(SymbolicValue)}{\begin{array}{c} C_S, K : a : b, Z : (v_a, s_a) : (v_b, s_b), E, M, N, G \vdash \textbf{BINARY_OP}(op) \rightarrow \\ C_S, K : op(a, b), Z : z, E, M, N, G \cup [(z, (v_a, s_a), op), (z, (v_b, s_b), op)] \end{array}}$$

$$\frac{a \in addr, b \in offset}{C_S, K : a : b, Z : (v_a, s_a) : (v_b, s_b), E, M, N, G \vdash \textbf{BINARY_SUBSCR} \rightarrow C_S, K : a[b], Z : s_a[offset(b)], E, M, N, G}$$

$$\frac{a \in addr, b \in offset}{\begin{array}{c} C_S, K : a : b : c, Z : (v_a, s_a) : (v_b, s_b) : (v_c, s_c), E, M, N, G \vdash \textbf{STORE_SUBSCR} \rightarrow \\ C_S, K, Z, E, M[a[b] \mapsto c], N, G \cup (s_a[offset(b)], s_c, \phi) \end{array}}$$

$$\frac{}{C_S, K, Z, E, M, N, G \vdash \textbf{CALL_FUNCTION}(f) \rightarrow C_S : f, K, Z, E : e_{fresh}, M, N, G}$$

$$\frac{}{C_S : f, K, Z, E : e, M, N, G \vdash \textbf{RETURN} \rightarrow C_S, K, Z, E, M, N, G}$$

$$\frac{}{C_S, K : a, Z : a_s, E, M, N, G \vdash \textbf{POP_TOP} \rightarrow C_S, K, Z, E, M, N, G}$$

$$\frac{}{C_S, K : a : b, Z : a_s : b_s, E, M, N, G \vdash \textbf{ROT_TWO} \rightarrow C_S, K : b : a, Z : b_s : a_s, E, M, N, G}$$

$$\frac{}{C_S, K : a, Z : a_s, E, M, N, G \vdash \textbf{DUP_TOP} \rightarrow C_S, K : a : a, Z : a_s : a_s, E, M, N, G}$$

$$\frac{a < b}{C_S, K : a : b, Z : z_a : z_b, E, M, N, G \vdash \textbf{BINARY_LESSTHAN} \rightarrow C_S, K, Z, E, M, N, G \cup [(z_a, z_b, <)]}$$

$$\frac{a \geq b}{C_S, K : a : b, Z : z_a : z_b, E, M, N, G \vdash \textbf{BINARY_LESSTHAN} \rightarrow C_S, K, Z, E, M, N, G \cup [(z_a, z_b, \geq)]}$$

Fig. 13. Operational semantics to construct a graph from program execution

- `TopologicalSort`(nd, G): returns a topological ordering of the set of nodes *nd*, using the edge connectivity information from graph *G*.
- `GetSuccessors`(n, G): returns all the nodes in graph *G* which have an incident edge originating from *n*.
- `GetPredecessors`(n, G): returns all the nodes in graph *G* from which an edge originates and ends at *n*.
- `AddEdgeToGraph`(e, G): adds the edge *e* to the set of edges in graph *G*.
- `RemoveNode`(n, G): removes the node *n* from graph *G* as well as all edges originating from and terminating at *n*.
- `isInput`(n, G): returns true if *n* corresponds to a symbolic value associated with the input to the user code under analysis, and false otherwise.
- `isOutput`(n, G): returns true if *n* corresponds to a symbolic value associate with the output (the value which is eventually returned) of the code under analysis, and false otherwise.
- `operator`(n): returns the operator if node *n* was created as the result of an arithmetic operation like addition, multiplication etc., and ϕ otherwise.
- `outputs`(G): returns all the nodes in the graph corresponding to a symbolic value associated with one of the values returned by the code under analysis.

Algorithm 1: Algorithm for compressing the size of the generated graph

```
  Function CompressGraph(G : [Dependency]):
1     nodes ⟵ GetNodesFromGraph(G)
2     edges ⟵ GetEdgesFromGraph(G)
3     toDelete ⟵ List()
4     for n ∈ TopologicalSort(nodes, G) do
5         if IsImportant(n, G) then
6             toDelete.append(n)
7     for n ∈ toDelete do
8         succ ⟵ GetSuccessors(n, G)
9         pred ⟵ GetPredecessors(n, G)
10        RemoveNode(n, G)
11        for s ∈ succ do
12            for p ∈ pred do
13                AddEdgeToGraph((p, s), G)
14    return
```

- `pathExists`(n, N): returns true if there is a path from node n to any one of the nodes in the set N, and false otherwise.

Algorithm 2: Choosing which nodes to filter from the graph

```
  Function IsImportant(n : node, G : [Dependency]):
1     if isInput(n, G) then
2         return true
3     if isOutput(n, G) then
4         return true
5     if operator(n) is not ϕ then
6         if pathExists(n, outputs(G)) then
7             return true
8     return false
```

4.3.3 *Feeding Graph into ML model.* In the preceding sections, we have described how to construct a graph from an execution of the given program on the provided input. However, we only described the graph in terms of the nodes generated and the edges between two nodes. For a graph to be processed by an ML model, its nodes and edges need to be translated into vectors $\in \mathbb{R}^n$. Consider the following code snippet:

```
def func(a, b, c):
    d, e = b + 88, a + c
    return d, e
```

Each node in the graph is translated to a vector $\in \mathbb{R}^6$, whose elements are computed as follows:

- First element: 1 if it corresponds to an input, 2 if it corresponds to an output, and 0 otherwise. In the snippet above, for nodes corresponding to the inputs a, b and c would have this element set to 1, and d, e would have this element set to 2, and for the rest of the nodes 0.
- Second element $\in \mathbb{N}$: corresponds to the cardinality of the input/output. In the snippet above, the node for a would have this element set to 1, the node for b would have this set to 2, and the node for c would have it as 3. In addition, the node for the output d would be set to 1, and for e it would be 2. a and d would be distinguished by the first element, not the second.
- Third element $\in \mathbb{N}$: corresponds to the offset of the particular input/output. For objects which correspond to a collection, the graph would contain a node for each memory location utilized by the program. The values are normalized to lie between 0 and 1. If in the snippet above, if a were a list of length 5, there would be 5 nodes corresponding to variable a, with this element having values 0.2, 0.4, 0.6, 0.8, and 1.0.
- Fourth element $\in \mathbb{N}$: corresponds to the operation which was used to create this node. In the snippet above, the node for e is created by adding a and c, hence the node would have this element set as an integer corresponding to the addition operation.
- Fifth element: 1 if the element corresponds to a constant (same value guaranteed throughout all executions) and 0 otherwise. In the snippet above, the graph would contain a node corresponding to the constant 88, which would have this field set to 1, rest of the nodes would have this field set to 0.
- Sixth element $\in \mathbb{R}$: if the fifth element is 1, this element would contain the value of the constant (normalized using a trigonometric function to lie between -100 and 100, to ensure generalization), and 0 otherwise.

The included metadata ensures that we are able to disambiguate between graphs that are created on a different number of function inputs, differentiate nodes corresponding to inputs, outputs, and constants, and different operations which may help identify different operations, say addition from subtraction. Each edge in the graph is translated to a vector $\in \mathbb{R}^3$, whose elements are computed as follows:

- First element $\in \mathbb{N}$: cardinality of the node from which this edge originates
- Second element $\in \mathbb{N}$: cardinality of the node at which this edge terminates
- Third element $\in \mathbb{N}$: corresponds to the operation which was used to create this node. This is used in the case of a comparison between two existing nodes.

4.4 Training Pipeline

As mentioned in Section 4.1, our framework utilizes a machine learning model to analyze the graph to predict its properties. However, to make the technique effective, it is necessary to train the model using some data. Our framework for training the network is shown in Figure 14.

In order to train the network, which requires inputs in the form of graphs, we need to find a way to generate a dataset of graphs, labeled with the ground truth.

To this end, we implement a corpus of different programs which can be represented using a NumPy API. Each NumPy API is implemented through only one program, in a way that all possible behaviors of the NumPy API are captured through our implementation (NumPy APIs typically provide a high degree of flexibility in terms of size and dimensions of inputs, broadcastable operations, masking, controlling the shape of the outputs). The list of implemented APIs is shown in Appendix A.

When training our network, we can sample a batch of programs from our corpus, which will provide us with a source code and the ground truth label (since the code used for training is implemented by us, we know the implemented functionality). Given the source code, we use an input generator that will generate different valid inputs on which the source code can be executed. Executing the code on different inputs is likely to yield different graphs, hence with the source code and input generated, we can use the graph generator to create a large number of graphs that forms the dataset using which the ML model can be trained.

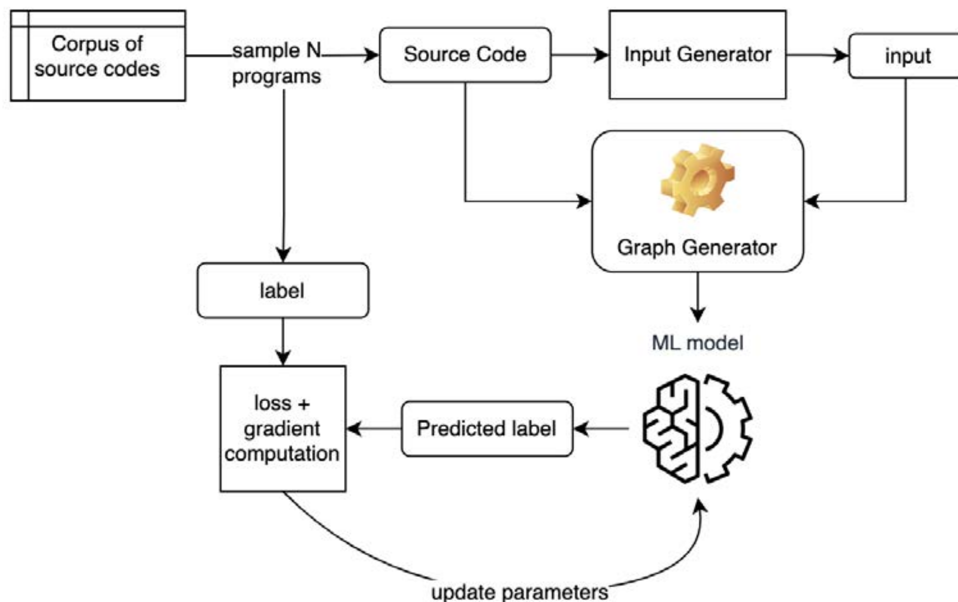


Fig. 14. Overall architecture of Training pipeline

5 EVALUATION

5.1 Experimental Setup

We use the techniques described in Section 4 to create a framework that accepts a code snippet written in pure[4] Python, and one valid input of the function. The framework instruments the code as described in Section 4.2, executes the code and creates a graph from the code execution as described in Section 4.3, which is processed by an ML model to recommend a NumPy API which is semantically closest to the provided code snippet and could be used in place of the user-provided code.

5.1.1 ML model. We use a Graph Convolutional Network (GCN) as described in Section 2 for our experiments. The network is trained in a supervised setting as described in Section 4.4 and trained to predict one out of N classes (described more in Section 5.1.3) where each class corresponds to one NumPy API. The exact architecture of the GCN used is described in Figure 15.

[4]Pure Python refers to code which only uses python code; it does not call any libraries which may be written a different language (like TensorFlow, whose compute kernels are written in C++/CUDA)

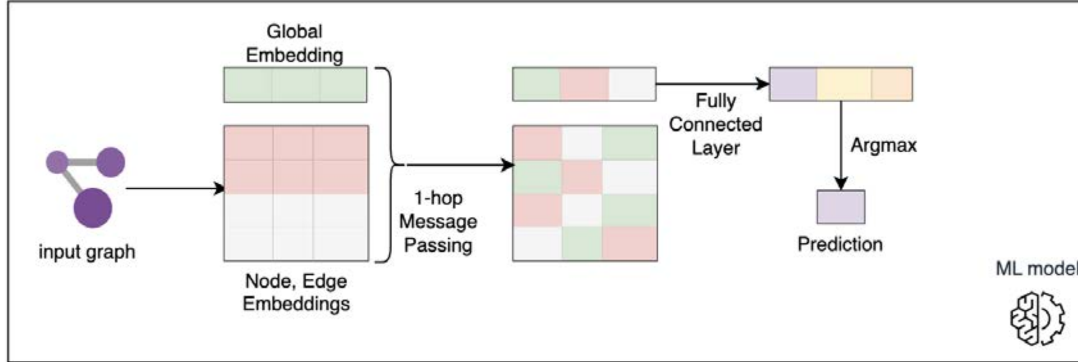


Fig. 15. Architecture of GCN used

In our network, all the hidden vectors are represented using a vector of length 512. To train the network, we use the Adam optimizer [28] for 20 epochs, with an initial learning rate of 10^{-4} which is decayed by 50% every 5 epochs. The model occupies approximately 13 MB of space or 3M parameters.

5.1.2 Implementation. Our instrumentation framework is written in 4K lines of Python. The dataset generators described in Section 4.4 are written in 1.5K lines of Python. Our corpus of implementations of different NumPy APIs is written in 3.5K lines of Python. We use an off-the-shelf implementation (with few modifications) of a Graph Convolutional Network written in JRAPH [20], a library supporting different Graph Neural Networks created by DeepMind which is built on top of JAX [12], a framework for automatic differentiation. The optax framework [7] is used for optimizing the ML model.

5.1.3 Dataset. As described in Section 4.4, we create a dataset of the execution graphs using a corpus of python code that mimics the behavior of NumPy APIs and an input generator (described in Appendix C) to run the corpus and generate the dataset of graphs. We create two datasets:

(1) **Dataset 1**: Contains implementations 26 different NumPy APIs in Appendix A. This dataset mainly deals with APIs which perform simple arithmetic operations from inputs to outputs. We have the following files:
   - **Train**: Contains 30000 labeled graphs created by running the corpus of inputs of size at most 6×6 (36 elements) (2-dimensional lists, each dimension can be at most 6). This partition is used only for training.
   - **Test**: Contains 30000 datapoints by running the graphs of inputs of size at most 6×6. This partition is used for testing.
   - **TestL**: Contains 30000 datapoints by running the graphs of inputs of size at most $7 \times 7 \times 7$. This partition is used for testing (Note the training is performed on at most 2-D lists, while this partition contains 3-D lists as well).
   - **TestLL**: Contains 15000 datapoints by running the graphs of inputs of size at most $10 \times 10 \times 10$. This partition is used for testing.
   - **TestLLL**: Contains 2300 datapoints by running the graphs of inputs of size at most $20 \times 20 \times 20$ (8000 elements). This partition is used for testing.

(2) **Dataset 2**: Contains implementations 46 different NumPy APIs in Table A. This dataset deals with some simple arithmetic operations but also some significantly more complex operations as well (like sorting, binary searching, and longer computations). NumPy is known for providing redundant APIs (two different APIs performing identical operations with minuscule differences), and this dataset deals with such APIs as well (for example *transpose*, *moveaxis*, *stack* etc., which perform identical operations in many cases). The following files are used:
   - **Train**: Contains 45000 labelled graphs created by running the corpus of inputs of size at most $4 \times 4 \times 4$ (64 elements). This partition is used only for training.
   - **Test**: Contains 45000 datapoints by running the graphs of inputs of size at most $4 \times 4 \times 4$. This partition is used for testing.
   - **TestL**: Contains 15000 datapoints by running the graphs of inputs of size at most $5 \times 5 \times 5 \times 5$. This partition is used for testing (Note the training is performed on at most 3-D lists, while this partition contains 4-D lists as well).
   - **TestLL**: Contains 7500 datapoints by running the graphs of inputs of size at most $10 \times 10 \times 10$ (1000 elements). This partition is used for testing.

Note that instead of just one testing dataset, we create multiple different testing datasets created by executing the corpus on different sizes, but only one training dataset. The reason for this is that since our graph representation is dependent on the size of the input with which the code is executed, we want to examine whether the patterns learned by the ML model indeed generalize across graphs of different sizes or do they overfit to the sizes seen only in the training examples. Note in Dataset 2, the training is done on inputs of size at most 64, while testing is done on inputs of size at most 1000 (15+ $\times$ larger), while in Dataset 1, the training is done on inputs of size at most 36, and testing is done on inputs of size at most 8000 (200+ $\times$ larger).

5.2 Baseline

As a baseline, we use the Large Language Model (LLM) Codex [14] released by OpenAI, which is based on the GPT [13] series of models but fine-tuned on a large corpus of code data. It is regarded as one of the state-of-the-art models in a wide variety of programming-related tasks like code recommendation, translation, summarization, and so on. They can also be tuned to a variety of different tasks by careful engineering of the prompt provided to the model. We create a prompt as described in Appendix D to tune the model to predict a NumPy API closest to the code snippet provided to the model.

As a baseline, we construct a dataset of code snippets generated by Codex which correspond to different NumPy APIs. An example prompt to codex could look like "#Write a python program equivalent to NumPy.sum which operates on Python lists". Using such prompts, we create a dataset of 59 different python-only implementations (the entries corresponding to 'Wild1' in Table 1 and Table 2) of different NumPy libraries, which we then use a prompt similar to the one in Appendix D to ask the LLM to predict the NumPy API closest to the code snippet provided, an example of self-consistency. Note that we make a few modifications to the LLM-generated code in case the generated code is incorrect, to fix the bugs.

To demonstrate the issue with text-based approaches like LLMs, we also perform some semantic preserving transformations like changing method and variable names, rearranging functions, and dead code insertion, as described in [22]. The entry 'Wild2' in Table 1 and Table 2 correspond to the transformation of changing method and variable names, and the entry 'Wild3' corresponds to rearranging functions, and dead code insertion in addition to the changes in 'Wild2'.

A reason for the huge popularity and success of LLMs is the amount of training data using which they are trained. The Codex model *code-davinci-002* is trained on 159 GB of GitHub code data which covers a significant fraction of all code available on the internet. In this light, we expect the performance of LLMs on the above benchmarks to be pretty high. We use two Codex models, *code-cushman-001* (7 billion parameters) which is a faster model but also less accurate, and *code-davinci-002* which has 175 billion parameters, and is regarded as the state-of-the-art model. The numbers in Table 1 and Table 2 correspond to *pass*@1, *pass*@2, *pass*@3 as seen in existing literature [14]. In the following experiments, a temperature of 0.1 was used as is recommended for Codex models [14].

Table 1. Codex Performance on Baseline (code-cushman-001)

Dataset	Top 1	Top 2	Top 3
Wild 1	74.6	74.6	74.6
Wild 2	54.2	55.9	55.9
Wild 3	52.5	55.9	59.3

We note that the performance of *code-cushman-001* suffers significantly as the input code is subjected to transformations which are ideally supposed to be semantics preserving, showing that the model is not truly able to capture the semantics of the program just from the source code.

Table 2. Codex Performance on Baseline (code-davinci-002)

Dataset	Top 1	Top 2	Top 3
Wild 1	84.7	84.7	84.7
Wild 2	81.4	81.4	83.1
Wild 3	78	79.7	79.7

The performance of *code-davinci-002* is a significant improvement over *code-cushman-001*, however we still see a drop in accuracy as the input programs are subject to the semantics preserving transformations, which should not affect the performance of any tool which aims to perform semantic analysis.

5.3 Scaling across size of Graphs

We mention in Section 4 that the size of the graph created by our framework depends on the size of the input being provided to the code snippet under analysis. In this light, it is necessary to examine whether the ML model learns a pattern that is independent of the size of the input or not. We run experiments to test this scaling across the two datasets 1 & 2 as described in Section 5.1.3.

To this end, for each of Dataset 1 & 2, we train the ML model as described in Section 4.4 on the **Train** partition, and test it on the partitions **Test**, **TestL** and so on. The results for both datasets are shown in Table 3 and Table 4.

Table 3. Scaling Performance of Framework on Dataset 1

Partition	Accuracy
Train	98.1
Test	98.1
TestL	98.1
TestLL	97.7
TestLLL	95.3

As we can see in Table 3, even though the model is trained on the **Train** partition which is generated on inputs of at most 36 elements, it is able to generalize well to the **TestLLL** partition which is made out of graphs 200× larger, showing that the pattern learned by the network does not depend on the size of the inputs used to create the graph.

Table 4. Scaling Performance of Framework on Dataset 2

Partition	Accuracy
Train	95.3
Test	94.8
TestL	93.4
TestLL	89.9

In Table 4, we see that the generalization is not as good as seen in Table 3, which is expected as we are using the same ML model for a harder dataset (as can be seen in Table A, Dataset 2 contains much harder APIs). We also note that the model is not able to overfit on the training data as well as in Dataset 1, the reason being overlapping behaviors of APIs (described in Section 5.1.3). However, in spite of that, we see the model generalizes fairly well to graphs in **TestLL** which contain graphs 15× larger than the training dataset.

5.4 Scaling across Implementations

In our previous experiments, the **Train** and all the **Test**ing datasets were created from the same implementation of a given NumPy API which were a part of a corpus. The corpus contained only one implementation of each API, but the implementation was written in a way to capture all possible behaviors of the respective APIs. For example, consider the documentation [21] of NumPy's *sum* function, which says it supports summation by axis, summation using a mask, whether to keep the original dimensions or not, and so on. All of these behaviors are covered by the python only implementation of the NumPy APIs we use in our corpus used to generate graph samples for the generator.

However, our graph representation (and the entire methodology of using the representation as the starting point for machine learning instead of simply the source code) would have limited use if it were only specific to the implementation using which the graphs in the training and testing datasets were created. We would like to examine how our graph representation can generalize to implementations of the same algorithm written by someone else. This evaluation would be similar to an out-of-distribution evaluation performed in Machine Learning literature.

To this end, we use the dataset 'Wild1' which was created by querying the LLM Codex's *code-davinci-002* model. We use the model we trained on Dataset 2 as described in Section 5.3. We write a simple input generator for each code in the corpus 'Wild1', and use it to generate inputs. Using the generated inputs and code snippets, we create the graph for

each of the code samples under inputs of different lengths as shown in Table 5 and check the accuracy of the prediction of our framework as compared to the ground truth. Note that all of the codes in ‘Wild1’ are significantly different implementations as compared to the codes used to generate the training data graphs in Datasets 1 & 2.

We also measure the Best of 1, 2, and 3 accuracy (Best of N means that if the ground truth is encountered in one of the top N predicted classes, we treat it as correct), to compare it with the baseline in Table 1 and Table 2.

Table 5. Generalization Performance of Framework on ‘Wild1’ Dataset

Length	**Top 1**	**Top 2**	**Top 3**
1	36.9	47.8	55.8
2	64.4	80.3	84.4
5	71.7	81.5	89.8
10	71.3	79.7	88.5

In the table above, we see that if we generate the graphs using inputs of size 1, the performance is poor. This is expected because if the input contains only one element, it would be hard to classify the algorithm being implemented just by observing the graph. Consider the example of *numpy.sum* and *numpy.prod*. If provided with an input with only one element, say a singleton list [7], the result would be the same (7 in both cases as the operations are degenerate). When we generate graphs using inputs of size 2, we note a significant improvement, the Top 1, 2 & 3 performance already surpasses the performance seen in *code-cushman-001* model as seen in Table 1.

Generating slightly larger graphs, on inputs of sizes 5 and 10 shows improved performance, we see the Top 3 accuracy even surpasses the performance of *code-davinci-002*, which has 13000 × more parameters. We see that our framework has significantly improved performance when we consider the top 3 predictions, compared to the LLM models which do not show significant improvement when going from the top 1 to the top 3 predictions.

In this experiment, we see that the performance of our framework is better than the performance of the LLMs when we consider the top 3 predictions from both models while having several orders of fewer parameters. However, we note that LLMs are designed to handle a wide variety of tasks, which require a significantly higher number of parameters.

6 LIMITATIONS AND FUTURE WORK

We discuss some limitations of our proposed framework and possible directions for future work to address the drawbacks and improve upon our technique.

- The framework we use to construct the graphs from execution relies on the assumption that the source code is written in pure Python, in that it does not call any libraries which may be written in another language. Due to this restriction, we were unable to directly use NumPy implementations themselves as a part of our training corpus in Section 4.4, and had to re-implement the functions in pure Python code, as shown in Appendix B. As future work, we can make our framework more robust to commonly used Python libraries: whenever our executor encounters a call to a library that is known to be written not in Python, we can intercept the module loader and replace the original implementation with implementation which can be handled by our library. This behavior is supported by the architecture of our graph generation framework.
- The framework currently handles only Python bytecode. Although this is an intended behavior given the huge popularity of Python, extending the framework to other languages and libraries would involve a significant

engineering effort. On the flip side, it would be a one-time effort for a given language/library as the APIs of existing functions do not get changed very frequently, though new functions may be added. One possible approach, though not applicable to Python, would be to use a common Intermediate Representation (IR) like LLVM [29], adding support for which would automatically add support for several different libraries and languages as it is a very commonly used IR.

- Python as a language itself also has a wide variety of supported behavior which makes it very easy to use as a programmer, but it makes the job of any analysis tool hard. Python supports dynamic typing, monkey patching, lambdas, threading, closures, lazy indexing, dynamic object attributes, operator overriding, and many more. All of these behaviors have to be implemented which is a significant engineering effort, however, it would basically be a one-time effort to implement a majority of these functionalities as they do not get changed a lot as the language goes through several iterations.
- NumPy itself is a complex library that supports a wide variety of functions. As mentioned in Section 5.1.3, many functions can be used to perform similar functions. To handle such functions with overlapping behaviors, we have to construct the input generators carefully to avoid significant overlaps which cause the classifier to not work very well to distinguish similar APIs. This is the reason why in Section 5.4 we not only consider the most likely prediction, but also the top 2 and top 3 predictions of the ML model to account for this complexity.
- In the current state, there are cases where our framework might not work. Consider the following two implementations of swap:

```python
def swap1(a, b):
    a, b = b, a
    return a, b

def swap2(a, b):
    a = a + b
    b = a - b
    a = a - b
    return a, b
```

  Both the implementations are semantically equivalent, however in the current state our framework would consider both implementations as different, due to the different computation graphs obtained for these implementations. However, to tackle this problem, we can use the concept of equivalence graphs [46] which can be used to simplify computations into some sort of canonical form, which would be able to identify the above two snippets as equivalent. Another example of semantically equivalent but following different computations would be normal matrix multiplication and Strassen's matrix multiplication algorithm [17, 42].
- An interesting application of the proposed technique would be legacy code analysis. Many times developers are faced dealing with code written by other people which may not be intelligible and hard to decipher just by reading the code. In cases like these, it would make sense to run such a code in a sandboxed environment, instrumented using a compatible framework, generate the graphs by executing the program, and use the generated graph to figure out properties of the program under analysis.
- We currently use the framework for classifying the given program into one of N known categories. Another application of the framework would be to perform code clone detection. Code clones could refer to programs

which are syntactically similar (Type-1 clones in literature [2]) or semantically equivalent (Type-4 clones in literature). Type-4 clones can only be detected through semantic analysis of the given codes, which could be difficult to do through source-code based analyses. Our framework could create graphs for each program under test on a given input, and feed the generated graphs through a class of Graph Neural Networks called Graph Isomorphism Networks [47] which could predict whether the codes being executed are indeed clones or not.

- The framework could also potentially be used for program synthesis. In this report, we use our framework to predict one NumPy API which is semantically closest to the user provided code. We can extend the idea to predict a set of APIs which can most likely be assembled to a program semantically equivalent to the user code. the graph analysis can be used to prune our APIs which would be useless to search given the specified behavior, thus making the synthesis problem tractable and scalable. Similar approaches have been proposed, though in different contexts in [9] and [8]. This approach would require some non-trivial effort to generate a dataset from which a recommendation model would be able to learn how to prune out irrelevant APIs.
- Our evaluation currently deals on simple numerical transformations on array-based algorithms as shown in Section 5.3, but also on complex numerical transformations as well as simple iterative procedures like sorting, searching, and operations involving a combination of such procedures. Most operations on the NumPy library can be expressed through array-based operations itself. It would be interesting to study how the framework generalizes to operations which involve different data structures like maps, sets, user defined classes etc. and more complex algorithms like graph transformations.

7 RELATED WORK

The work presented in this technical report has similarities with a number of different active areas of research. The work falls under the broad umbrella of program analysis, which encompasses a wide range of techniques ranging from compilation, optimization [38] to code summarization [45] and code clone detection [2] and many more. Program analysis techniques can be classified into two main categories: static analysis and dynamic analysis.

Static analysis refers to techniques which attempt to perform analysis on a piece of code without executing the code. The most fundamental example of such an analysis can be thought to be compilers [1], which take in code written in one (typically high level) language, perform a series of operations to an executable which can be directly executed by a machine [41], or more commonly to another (typically low level, like [29]) language which can be processed further by existing compilers.

Dynamic analysis refers to techniques which perform analysis on a piece of code by executing the code fragment. The simplest and most common example can be thought to be software testing, where a piece of code is executed and its output tested against a gold standard to estimate whether a piece of code performs as expected. Apart from simple testing, several other techniques have come up which execute code to perform analysis like concolic testing [35, 39, 40], fuzzing [18, 33, 34]. For a dynamically typed language like Python, pure static analysis can be hard to reason with as Python does not have a static type system, hence most analysis engines for Python involve some degree of dynamic analysis as well. Our framework is an example of a dynamic analysis engine for Python.

Our work is closely related to two existing classes of problems in literature: function name prediction and API recommendation/synthesis. Function name recommendation refers to the problem of taking a piece of code which does not have a name, or has a poorly chosen name, and recommending a name for it depending on the code in the body of the function. It is an extremely popular area of research [3, 19, 27, 31]. Invariably though, the approach to this problem is based in static analysis, and for the task of function name prediction, one would note that most of the

benchmarks are done in Java as (1) Java is statically typed, and (2) Java methods tend to have very descriptive names, like "GetFourthElementFromSortedList" using which it is possible to define precision and recall based on whether the predicted class has missed some keywords, or whether there are extra keywords.

API Recommendation/Synthesis deals with recommending APIs from a given library which would perform the operation as specified by the user. Examples of such tools are [9], [10], which are built for the Pandas library. Other examples include [8] and [16] which operate on limited domain-specific languages. However, a lot of such tools deal with specifications provided as input/output examples. In some domains, it may be easy to specify I/O examples, but for operations on tensors which NumPy provides, I/O examples can be hard to specify for high dimensional tensors or else can be under-specifications which would require more intense involvement from the user. We believe that simple, unoptimized code snippets (which would not be under- or over-specifications) can be an excellent way of specifying numerical computations and algorithms, the type which we encounter in NumPy.

Our problem dealt with learning the behavior of different NumPy APIs by executing them across different combinations of inputs. We were unable to use existing benchmarks like [32] as they deal with codes covering a diverse set of operations but not in too much detail, however we needed implementations which would cover all possible behaviors of the NumPy APIs according to the documentation [5], which are not freely available. In addition, a limitation of our framework as discussed in Section 6 was that it could only handle pure Python code, hence we re-implemented the APIs in pure Python as alternatives were not available (NumPy's own source code which carries out the main computations is written in C/C++).

Our idea of using the computation graphs to express how the inputs get converted into outputs have also been used particularly in deep learning [26, 48] frameworks which try to optimize computations depending on the properties of computation kernels. Existing work operate at the granularity of matrix-level operations like matrix multiplication, addition, transpose etc, while our approach operates at the granularity of individual scalar-level operations.

Recently generative Large Language Models have gained a lot of popularity and traction due to the fact that they are able to solve a wide variety of tasks. These are language models which simply predict the next word given a prompt repeatedly, however models which have hundreds of billions of parameters [14] tend to work very well for a wide variety of programming tasks like program synthesis [6], code repair [25] and others. We use LLMs in our project both to generate a dataset of code not seen by our model, as well as a baseline for the API recommendation problem as it is able to provide the closest functionality to the task we perform. The largest of the Codex models, *code-davinci-002* has excellent performance and our framework can only beat it in half of the numbers reported in Table 2 and Table 5. In spite of these successes, the models still can be prone to misclassification when encountered with a code with unexpected variable names, not an ideal property for a developer to rely on the model's predictions.

8 CONCLUSION

In this technical report, we have proposed a graph representation that is generated using information generated from the execution of a program on a given input, which can be used to predict a NumPy API semantically closest to the provided program. Since the representation is generated from the execution of the program, rather than just the source code, can be a better starting point for semantic analysis especially for dynamically typed languages like Python where there is not a lot of information for static analyses to work with. We evaluate the performance of our framework against the performance of a Large Language Model (Codex) on the same task and see comparable performance in spite of

[5]https://numpy.org/doc/

our framework using several orders of magnitude fewer data and lesser parameters in the ML model. We believe the proposed representation can be used for further more complex analyses and be a useful element in an ever-increasing toolkit for software developers.

ACKNOWLEDGMENTS

I would like to thank Prof Koushik Sen for his advice and guidance on this project, and Shadaj Laddad for the design and implementation of the instrumentation framework. I also thank Moustafa AbdelBaky, Prof Alvin Cheung, Sahil Bhatia, Jathushan Rajasegaran, and Aniketh Reddy for their helpful feedback and discussions. I also thank Arpita Banerjee for her unwavering support throughout the duration of the project.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference on Machine Learning (ICML).*

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations.* https://openreview.net/forum?id=H1gKYo09tX

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *ArXiv* abs/2108.07732 (2021).

[7] Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, John Quan, George Papamakarios, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Luyu Wang, Wojciech Stokowiec, and Fabio Viola. 2020. *The DeepMind JAX Ecosystem.* http://github.com/deepmind

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. https://doi.org/10.48550/ARXIV.1611.01989

[9] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. 3, OOPSLA, Article 168 (oct 2019), 27 pages. https://doi.org/10.1145/3360594

[10] Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: Program Synthesis by Reasoning over Graphs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 134 (oct 2021), 29 pages. https://doi.org/10.1145/3485511

[11] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Reliable Software.*

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[15] Matthieu Dartiailh. 2022. https://github.com/MatthieuDartiailh/bytecode

[16] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. https://doi.org/10.48550/ARXIV.2006.08381

[17] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610 (2022), 47 – 53.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *WOOT @ USENIX Security Symposium*.

[19] Fan Ge and Li Kuang. 2021. Keywords Guided Method Name Generation. https://doi.org/10.48550/ARXIV.2103.11118

[20] Jonathan Godwin*, Thomas Keck*, Peter Battaglia, Victor Bapst, Thomas Kipf, Yujia Li, Kimberly Stachenfeld, Petar Veličković, and Alvaro Sanchez-Gonzalez. 2020. *Jraph: A library for graph neural networks in jax.* http://github.com/deepmind/jraph

[21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[22] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner53432.2022.00070

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735

[24] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[25] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2021. Jigsaw: Large Language Models meet Program Synthesis. https://doi.org/10.48550/ARXIV.2112.02969

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[27] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1631–1645. https://doi.org/10.1145/3548606.3560612

[28] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis &; Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[30] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[31] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. https://doi.org/10.48550/ARXIV.2201.10705

[32] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. (07 2017).

[33] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[34] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

[35] Fayozbek Rustamov, Juhwan Kim, Jihyeon Yu, and Joobeom Yun. 2021. Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection. *IEEE Access* 9 (2021), 131166–131190. https://doi.org/10.1109/ACCESS.2021.3114202

[36] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[38] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2012. Stochastic Superoptimization. https://doi.org/10.48550/ARXIV.1211.0557

[39] Koushik Sen. 2005. DART: directed automated random testing. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

[40] Koushik Sen, Darko Marinov, and Gul A. Agha. 2005. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13*.

[41] Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA.

[42] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (aug 1969), 354–356. https://doi.org/10.1007/BF02165411
[43] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.
[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762
[45] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. https://doi.org/10.48550/ARXIV.1811.07234
[46] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304
[47] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? https://doi.org/10.48550/ARXIV.1810.00826
[48] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. https://doi.org/10.48550/ARXIV.2201.12023

APPENDICES

A SUPPORTED NUMPY APIS

Table 6. APIs supported in Dataset 1

Sum	Any	Power	ZerosLike	Eye
Prod	Add	Abs	Ones	Concatenate
Mean	Subtract	Exp	OnesLike	Dot
Max	Multiply	Sqrt	Shape	Arange
Min	Divide	Zeros	Transpose	Linspace
All				

Table 7. APIs supported in Dataset 2

Sum	Argmax	Arange	Concatenate	Diag
Prod	Argmin	Linspace	Vstack	Full
Max	Add	Allclose	Hstack	Sort
Min	Where	Transpose	Stack	Argsort
Mean	Dot	Ravel	Tile	Percentile
All	Matmul	Moveaxis	Append	Median
Any	Outer	Squeeze	Insert	Unique
Count_Nonzero	Clip	Reshape	Repeat	Searchsorted
Std	Diff	Atleast_1d	Eye	Take
Cumsum				

B EXAMPLE PYTHON-ONLY IMPLEMENTATION IN TRAINING DATASET

As mentioned in Section 5.1.3, we had to re-implement NumPy functions to only use Python code, but we ensured that the implementations cover all the functionalities as given in the NumPy documentation. We show an example implementation of ‘sum

```
def sum_1(a, axis=None, keepdims=None, initial=None, where=None):
```

```
shape_input = shape.shape_1(a)
dim_input = len(shape_input)

if axis is None:
        axis = [i for i in range(dim_input)]
elif isinstance(axis, int):
        axis = [axis]
for i in range(len(axis)):
        if axis[i] < 0:
                axis[i] += dim_input

if where is not None:
        where_numeric = ones_like.ones_like_1(where)
        for indices in itertools.product(*[list(range(j)) for j in
            shape.shape_1(where)]):
                out = where_numeric
                inn = where
                for i, index in enumerate(indices):
                        inn = inn[index]
                        if i + 1 == len(indices):
                                out[index] = 1 if inn else 0
                        else:
                                out = out[index]
        masked_a = mul.mul_1(a, where_numeric)
else:
        masked_a = a

new_shape_input = shape.shape_1(masked_a)
new_dim_input = len(new_shape_input)

shape_output = []
index_flags = []
for i_raw in range(new_dim_input):
        if i_raw < new_dim_input - dim_input:
                index_flags.append(None)
        else:
                i = i_raw - (new_dim_input - dim_input)
                if i in axis:
```

```
                if keepdims:
                    shape_output.append(1)
                    index_flags.append(False)
                else:
                    index_flags.append(None)
            else:
                if shape_input[i] == new_shape_input[i_raw]:
                    shape_output.append(shape_input[i])
                    index_flags.append(True)
                else:
                    shape_output.append(1)
                    index_flags.append(False)

output = zeros.zeros_1(shape_output)
if initial is not None:
    output = add.add_1(output, initial)

for indices in itertools.product(*[list(range(j)) for j in
    new_shape_input]):
    inp = masked_a
    out = output
    prev_out = [out, None]
    for i, index in enumerate(indices):
        inp = inp[index]
        if index_flags[i] == True:
            prev_out = [out, index]
            out = out[index]
        elif index_flags[i] == False:
            prev_out = [out, 0]
            out = out[0]
    if prev_out[1] is not None:
        prev_out[0][prev_out[1]] += inp
    else:
        output += inp

return output
```

C EXAMPLE GENERATOR TO GENERATE DATASET

To actually generate the dataset, the example given in Appendix B had to be provided with an input generator. The generator for ‘sum’ was given by the following snippet:

```
def get_broadcast_compatible_shape_small(original, max_dim):
  import numpy as np
  if np.random.randint(0, 2):
    return original
  new_shape = []
  for i in original[::-1]:
    if i == 1:
      new_shape.append(i)
    else:
      if np.random.randint(0, 2):
        new_shape.append(1)
      else:
        new_shape.append(i)
    if np.random.randint(0, 2):
      return new_shape[::-1]
  return new_shape[::-1]

def get_random_init_reduction(dims, dim_size_max):
  import numpy as np
  size = tuple([np.random.randint(1,dim_size_max+1) for i in range(np.
      random.randint(1, dims+1))])
  size_where = get_broadcast_compatible_shape_small(size, dim_size_max)
  a = np.random.uniform(low=-10., high=10., size=size).tolist()
  if len(size) == 1:
    axis = None
  else:
    axis = np.random.choice([None, np.random.choice(len(size), size=np.
        random.randint(1, len(size)), replace=False).tolist()])
  keepdims = np.random.choice([None, True])
  initial = np.random.choice([None, 1])
  where = np.random.choice([None, np.random.choice([True, False], p=[0.8,
      0.2], size=size_where).tolist()])
  return size, size_where, a, axis, keepdims, initial, where
```

```
def testReductionSum(dims, dim_size_max):
    import numpy as np
    import APIs.sum as sum
    size, size_where, a, axis, keepdims, initial, where =
        get_random_init_reduction(dims, dim_size_max)
    with receiver:
        ans = sum.sum_1(a, axis=axis, keepdims=keepdims, initial=initial,
            where=where)
    print("reduction␣sum:␣", ans)
```

Note how we also take into account the broadcasting operation which NumPy's 'sum' method supports.

D EXAMPLE PROMPT TO CODEX LLM FOR BASELINE EVALUATIONS

To get Codex to recommend a NumPy API closest to the provided code, the following prompt was used:

```
<Query Code Inserted Here>

#Recommend a NumPy API which is equivalent to func.
import numpy as np
#The numpy API is np.
```

After prompting the model, the next tokens would contain the prediction by the Codex LLM.