LExecutor: Learning-Guided Execution

Beatriz Souza
University of Stuttgart
Stuttgart, Germany
beatrizbzsouza@gmail.com

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

Executing code is essential for various program analysis tasks, e.g., to detect bugs that manifest through exceptions or to obtain execution traces for further dynamic analysis. However, executing an arbitrary piece of code is often difficult in practice, e.g., because of missing variable definitions, missing user inputs, and missing third-party dependencies. This paper presents LExecutor, a learning-guided approach for executing arbitrary code snippets in an underconstrained way. The key idea is to let a neural model predict missing values that otherwise would cause the program to get stuck, and to inject these values into the execution. For example, LExecutor injects likely values for otherwise undefined variables and likely return values of calls to otherwise missing functions. We evaluate the approach on Python code from popular open-source projects and on code snippets extracted from Stack Overflow. The neural model predicts realistic values with an accuracy between 79.5% and 98.2%, allowing LExecutor to closely mimic real executions. As a result, the approach successfully executes significantly more code than any available technique, such as simply executing the code as-is. For example, executing the open-source code snippets as-is covers only 4.1% of all lines, because the code crashes early on, whereas LExecutor achieves a coverage of 51.6%.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

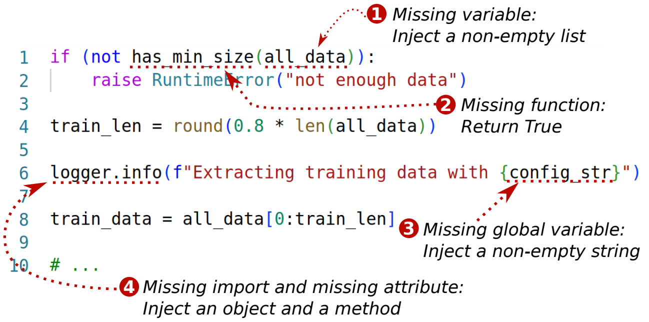
KEYWORDS

execution, neural models, dynamic analysis

ACM Reference Format:
Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616254

1 INTRODUCTION

The ability to execute code enables various dynamic program analysis applications. For example, running a program may expose bugs that trigger obviously wrong behavior, such as runtime exceptions. Moreover, dynamic analysis has been shown its usefulness for various tasks, e.g., mining API protocols [31, 39, 69, 89], taint analysis [19, 76], and type inference [8, 74]. Generally, dynamic analysis is well known to complement static analysis [25, 80], providing a valuable technique for understanding and improving software.



```
if (not has_min_size(all_data)):
    raise RuntimeError("not enough data")

train_len = round(0.8 * len(all_data))

logger.info(f"Extracting training data with {config_str}")

train_data = all_data[0:train_len]

# ...
```

❶ *Missing variable: Inject a non-empty list*

❷ *Missing function: Return True*

❸ *Missing global variable: Inject a non-empty string*

❹ *Missing import and missing attribute: Inject an object and a method*

Figure 1: Python code to execute and how LExecutor guides the execution.

Unfortunately, executing arbitrary code is challenging, both at the small and the large scale. Small-scale code snippets, e.g., posted on Stack Overflow, often are missing contextual information, such as imports and definitions of variables and functions [45]. Large-scale code bases often are non-trivial to set up and run, e.g., due to missing third-party dependencies, complex build procedures, and the lack of inputs that reach deeply into the project. Overall, the difficulties of executing code limit the potential of dynamic analysis.

For example, consider the Python code snippet in Figure 1, which could be the body of a function extracted from a complex code base or code posted on Stack Overflow. Executing this code is difficult because of various missing pieces of contextual information. At first, the code tries to read variable `all_data`, which does not exist, and hence, will cause the code to crash. Even we had a definition for this variable, then other obstacles would prevent the code from executing, such as the missing function `has_min_size`, the missing (global) variable `config_str`, and the missing imported method `logger.info`. A skilled human can intuitively fill in the missing information by predicting it to be likely that, e.g., `all_data` is a list, `has_min_size` will return a boolean, `config_str` is a string, and that `logger.info` is a method call likely to succeed. Given this information, a human could mentally emulate the execution and hence reason about the runtime behavior of the code. The key question we ask in this work is: *Can we automate the prediction of likely values and use them to execute otherwise non-executable code?*

This paper presents LExecutor, which offers a learning-guided approach for executing arbitrary code snippets in an underconstrained way. The key idea is to let a neural model predict suitable

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616254

values whenever the program usually would be stuck. The approach is enabled by three components:

- A specifically designed and trained neural learning model that predicts runtime values based on the code context in which a value is used. We realize this component by fine-tuning a model with hundreds of thousands of examples gathered from regular executions of real-world programs.
- An execution environment that prevents crashes due to missing values and instead fills in model-predicted values.
- An AST-based instrumentation technique that turns arbitrary Python code into "lexecutable" code.

For the example in Figure 1, LExecutor prevents the program from crashing and instead predicts suitable values for all undefined variables and functions. The figure shows what values the approach predicts and then injects into the execution. For example, when the code tries to access the otherwise missing value `has_min_size`, then LExecutor predicts it to be a function that returns the boolean value `True`. Given the injected values, the code successfully executes without prematurely terminating. There is no guarantee that the values that LExecutor injects exactly match those that would occur in a "real" execution of the given code. However, we find that LExecutor is able to predict realistic values in *most* situations where a regular execution would simply get stuck, enabling the execution of otherwise non-executable code.

Our work has overlapping goals with four prior streams of research: (1) Test generators [38, 62], as they also provide missing input values. (2) Concolic execution [15, 41, 79], which abstracts missing inputs into symbolic variables and constraints. (3) Micro-execution [40] and underconstrained symbolic execution [72], which execute arbitrary code by injecting random values into memory on demand. In contrast to (1) and (2), LExecutor injects otherwise missing values at arbitrary points within an execution, whereas the prior work provides values only at well-defined interfaces, e.g., command-line arguments or method arguments. In contrast to all the above work, LExecutor uses a neural model to predict realistic values, whereas (1) and (3) inject random values, and (2) overapproximates possible values via symbolic reasoning. (4) Neural models that predict various properties of code [67, 73]. In contrast to (4), LExecutor predicts runtime values during an execution, whereas prior work focuses on predicting static properties of code.

We evaluate our work by applying it to two sets of code snippets: functions extracted from popular open-source projects and code snippets extracted from Stack Overflow posts. As baselines, the evaluation compares LExecutor to regular execution, injecting random values, and a state-of-the-art, function-level test generator for Python [54]. We show that the neural model at the core of LExecutor predicts realistic values with an accuracy of 79.5% to 98.2% (depending in the configuration). Moreover, our results show that the approach enables the execution of 51.6% and 65.1% of all lines in the open-source code and Stack Overflow snippets, respectively, which improves over the best baseline by 29% and 12.2%. Finally, we apply LExecutor to detect semantics-changing commits by comparing the execution behavior of a function before and after a commit.

We envision LExecutor to enable various dynamic analysis applications. For example, "lexecuting" code could help detect bugs that manifest through obvious signs of misbehavior, such as runtime exceptions or assertion violations. Likewise, our approach could be used to validate code generated by code synthesis techniques [37] or generative language models [29, 88]. Other potential applications include to check if and how a code change to modifies the observable behavior [72], and classical dynamic analyses, such as detecting security vulnerabilities via taint analysis.

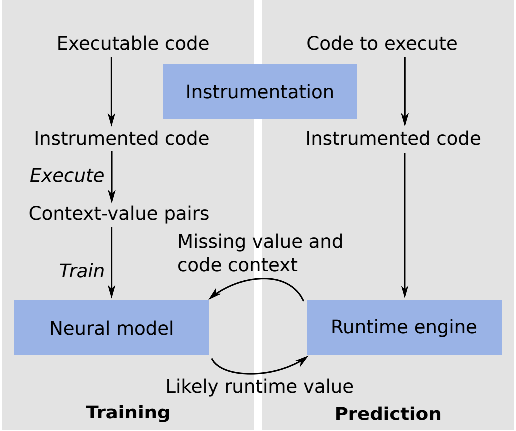


Figure 2: Overview of learning-guided execution.

In summary, this paper contributes the following:

- A novel way of executing arbitrary code in an underconstrained and learning-guided way.
- A neural model designed and trained to predict realistic values to be injected into an execution. The task addressed by the model fundamentally differs from prior work on predicting various properties of code [67, 73], because we here predict runtime values.
- An instrumentation-based runtime environment that prevents programs from crashing due to missing values and instead injects values provided by the model.
- Empirical evidence that the approach effectively injects realistic values and outperforms several baselines at covering and successfully executing code.

2 OVERVIEW

Figure 2 gives an overview of our approach. LExecutor has a *training phase* and a *prediction phase*. The training phase yields a *neural model* that, given the code context in which a value is used during an execution, predicts a suitable value. To this end, we train a model based on pairs of code context and a value used in this context, which we gather by executing a corpus of code. Once the neural model is trained, LExecutor is ready for the prediction phase. The input to the approach is a possibly incomplete piece of code to execute. To execute this code despite possibly missing information, a *runtime engine* intercepts any use of a value, such as a read of a variable. If the value is not defined, i.e., the program would usually terminate prematurely, the runtime engine queries the neural model and then injects the value proposed by the model into the execution.

Both training and prediction are enabled by source-to-source *instrumentation*, which takes regular Python code and adds new instructions to it. The added instructions wrap all uses of values into special *value loaders* that implement the functionality of LExecutor while otherwise preserving the original semantics.

For the example in Figure 1, an execution starts at line 1 by using two values, first `all_data` and then `has_min_size`. LExecutor wraps both into value loaders that allow the runtime engine to observe that these values are missing, and to hence inject values that would likely occur in a regular execution. As shown in the figure, the approach injects a non-empty list for `all_data` and a function returning `True` for `has_min_size`. As a result, the code continues to execute, passes line 4 without any need for guidance by LExecutor, and then reaches line 6. Here, several values are missing, which the runtime engine provides successfully. For the missing method `logger.info`, the approach first injects a dummy object for `logger`, then resolves the missing attribute `info` into a function, and finally injects the return value `None` when calling the function. By guiding the execution in this way, LExecutor successfully executes all code shown in the figure.

3 APPROACH

The following presents in detail the three main components of LExecutor: the code instrumentation, the neural model, and the runtime engine.

3.1 Code Instrumentation

The goal of the instrumentation is to observe runtime values to be used during the training phase and to inject otherwise missing values during the prediction phase. We perform a series of AST-based code transformations via a top-down, left-to-right pass over all nodes in the given AST. The pass transforms three kinds of AST nodes and the corresponding AST subtrees under them:

- *Variable reads.* The instrumentation transforms all nodes that refer to the name of a variable that gets read. In contrast, we do not instrument variable writes. For example, in `y = x + 1` the reference to `x` gets instrumented, as it might read an undefined value, which LExecutor tries to prevent. In contrast, the assignment to `y` is not instrumented and hence executed as in the original code.
- *Attribute reads.* The instrumentation transforms all nodes that refer to an attribute that gets read. Again, we do not instrument writes of attributes. For example, in `y.foo = x.bar` both the read of the variable `x` (as described above) and the read of the attribute `bar` get instrumented, because they might access otherwise undefined values.
- *Calls of functions and methods.* The instrumentation transforms all calls of functions and methods (we say "functions" to mean both). For example, in `y = foo()` the call of `foo` gets instrumented so LExecutor can inject a return value if the function is undefined.

Each of the three kinds of instrumented nodes is replaced by a call to a special *loader function.* For variable reads, the instrumented code calls a loader function `_n_` and passes two pieces of information to it: the name of the variable and a newly created lambda function that tries to read and then return the value of the variable. The lambda function enables LExecutor to try to read the variable in a controlled manner and to react accordingly depending on whether the value is available. For attribute reads, the instrumented code passes two pieces of information to a loader function `_a_`: the value of the base object (which when accessing the attribute has already been evaluated) and the name of the attribute. The loader function will then try to read the attribute of the base object and react accordingly depending on whether the attribute is available. Finally, for function calls, the instrumented code passes the callee (i.e., the called function) to a loader function `_c_`. The loader function will then invoke the callee, while handling cases where the function is undefined. In addition to the parameters mentioned above, each of the loader functions also receives a unique identifier of the instrumented source code location, which LExecutor later uses to access the code context of the location.

Original code:

```
x = foo()
y = x.bar + z
```

Instrumented code:

```
x = _c_(536, _n_(535, "foo", lambda: foo))
y = _a_(538, _n_(537, "x", lambda: x), "bar") \
        + _n_(539, "z", lambda: z)
```

Figure 3: Example of code instrumentation.

Figure 3 illustrates the instrumentation with an example, where the original and the instrumented code are shown at the top and bottom, respectively. The call of `foo()` in the first line consists of two separately instrumented instructions: reading the variable `foo`, which gets wrapped into `_n_`, and calling the function stored in this variable, which gets wrapped into `_c_`. The numbers passed as the first argument to the loader functions are the unique identifiers of code locations. The second line of the original code contains three instructions to instrument: the two variable reads of `x` and `z`, and the attribute access wrapped into `_a_`. Note that the calls to the loader functions are nested into each other. For example, in the first instrumented line, the value returned by `_n_` as the result of reading variable `foo` is passed as the callee to `_c_`.

In addition to the instrumentation described above, the approach adds to each instrumented file imports of the loader functions. With these imports, the instrumented file serves as a drop-in replacement of the original file. Apart from changing the uses of values, the instrumentation does not modify the semantics of the code.

3.2 Neural Model

We next describe the neural model, which predicts likely runtime values to use in a given code context.

3.2.1 Gathering Training Data. To gather data for training the model we execute and dynamically analyze tests suites of popular Python projects. The training executions are not guided by LExecutor, but regular executions with all inputs and dependencies given. Via dynamic analysis the approach gathers a trace of value-use events:

Definition 3.1. A *value-use event* is a tuple (n, v, k, l), where

- n is the name of the used reference, i.e., a variable name, an attribute name, or the name of a function;

- v is an abstracted version of the value that gets used;
- k indicates the kind of used value, which is either *variable*, *attribute*, or *return value*; and
- l is a unique identifier of the code location where the value gets used.

To gather a trace of such events, LExecutor instruments all executed code (Section 3.1). The loader functions invoked by the instrumented code load the requested value, as the original code, and in addition, add a value-use event to the trace. At the end of the execution, LExecutor stores the recorded trace into a file.

3.2.2 Representing Values for Learning. Given traces of value-use events, we need to represent these data in a format suitable for deep learning. An important part of this step is to abstract the concrete runtime values observed during an execution into a finite number of classes, which will be the prediction targets of the neural model. The abstraction has two conflicting goals. On the one hand, we aim for a fine-grained, precise prediction of what value to inject into an execution. For example, if the code uses a variable `age`, predicting the exact integer to load would enable LExecutor to produce the most realistic possible execution. On the other hand, we aim for a highly accurate model, which becomes easier if there are only few, coarse-grained classes of values to choose from. For the above example, such a coarse-grained prediction could simply say that the value should be an integer, without any more information about the specific value.

LExecutor strikes a balance between these two conflicting goals by using a fixed-size set of abstracted values. By default, this set comprises 23 classes as shown in Table 1. The left column gives the classes of values that the neural model distinguishes. For some very common primitive values, namely `None`, `True`, and `False`, the approaches preserves the concrete values, enabling LExecutor to predict them exactly. For numeric types, such as `int` and `float`, we abstract the concrete values into three classes per type, which represent negative, zero, and positive values, respectively. To represent strings and common data structure types, such as lists, the approach distinguishes between empty and non-empty values, e.g., an empty list as opposed to a non-empty list. All callable values, such as functions, are represented as a class *callable*. The *resource* class represents values that can be opened and closed, such as file pointers, which are often used with the `with` keyword in Python. Finally, all remaining non-primitive types are abstracted as *object*. We refer to the value abstraction described above as *fine-grained*.

To better understand the impact of the number of classes that values are abstracted into, LExecutor also supports a more *coarse-grained* value abstraction with only 12 classes, as shown in Table 2. Instead of distinguishing between different values of a type, the coarse-grained value abstraction maps all values of a type into a single class. For example, the abstraction does not distinguish between negative, zero, and positive integers, but simply represents all of them as integers. Unless otherwise mentioned, all results are based on the fine-grained variant of the approach.

3.2.3 Representing Code Context for Learning. The goal of the neural model is to predict one of the abstract value classes for each missing value. To this end, we provide the following contextual information as an input to model:

Table 1: Fine-grained abstraction and concretization of values.

Abstract class of values	Concretization (Python)
Common primitive values:	
None	`None`
True	`True`
False	`False`
Built-in numeric types:	
Negative integer	`-1`
Zero integer	`0`
Positive integer	`1`
Negative float	`-1.0`
Zero float	`0.0`
Positive float	`1.0`
Strings:	
Empty string	`""`
Non-empty string	`"a"`
Built-in sequence types:	
Empty list	`[]`
Non-empty list	`[Dummy()]`
Empty tuple	`()`
Non-empty tuple	`(Dummy())`
Built-in set and dict types:	
Empty set	`set()`
Non-empty set	`set(Dummy())`
Empty dictionary	`{}`
Non-empty dictionary	`{"a": Dummy()}`
Functions and objects:	
Callable	`Dummy`
Resource	`DummyResource()`
Object	`Dummy()`

Table 2: Coarse-grained abstraction and two modes for concretizing values.

Abstract class	Concretization (Python)	
of values	Deterministic	Randomized
Common primitive values:		
None	`None`	
Boolean	`True`	`True`, `False`
Built-in numeric types:		
Integer	`1`	`-1`, `0`, `1`
Float	`1.0`	`-1.0`, `0.0`, `1.0`
Strings:		
String	`"a"`	`""`, `"a"`
Built-in sequence types:		
List	`[Dummy()]`	`[]`, `[Dummy()]`
Tuple	`(Dummy())`	`()`, `(Dummy())`
Built-in set and dict types:		
Set	`set(Dummy())`	`set()`, `set(Dummy())`
Dictionary	`{"a": Dummy()}`	`{}`, `{"a": Dummy()}`
Functions and objects:		
Callable	`Dummy`	
Resource	`DummyResource()`	
Object	`Dummy()`	

Definition 3.2. The *input to the model* is a sequence of tokens

$$n \ \langle sep \rangle \ k \ \langle sep \rangle \ c_{pre} \ \langle mask \rangle \ c_{post}$$

where

- n is the name used to refer the to-be-predicted value;
- $\langle sep \rangle$ is a special separator token
- k is the kind of value to predict, i.e., *variable*, *attribute*, or *return value*.
- c_{pre} are the code tokens just before the reference to the to-be-predicted value;
- $\langle mask \rangle$ is a special masking token; and
- c_{post} are the code tokens just after the reference to the to-be-predicted value.

The prediction task is related to the well-known unmasking task, which is commonly used to train language models [22, 36]. In contrast to unmasking, we do not ask the model to predict a masked token, which would be trivial, as it is given in n as part of the input. Instead, we ask the model to predict what value the masked token most likely evaluates to.

Example. Suppose the code in Figure 1 gets executed with all required contextual information, e.g., as part of a larger project, and that we gather training data from its execution. The approach keeps track of all used values, such as the read of `all_data` at line 1. Following Definition 3.1, the approach records a value-use event (n, v, k, l) with n ="all_data", v ="non-empty list", v ="variable", and l being a unique identifier of the code location. The corresponding input to the model is a tokenized version of the following:

`all_data` $\langle sep \rangle$ *variable* `if (not has_min_size(` $\langle mask \rangle$
`)): raise RuntimeError( ...`

The prediction target for this example is "non-empty list".

3.2.4 Training and Prediction. The data preparation steps described above yield pairs of input sequences and abstracted values. Before training the model, we deduplicate these pairs, which greatly reduces the amount of training data because programs often repeatedly load the same value at the same location, e.g., in a loop. We then train a single model for all three kinds of values to predict, allowing the model to generalize across values stores in variables and attributes, as well as return values of function calls.

For the neural model, our approach can build on any model that accepts a sequence of tokens as the input and then either acts as a classifier or predicts tokens from a given vocabulary. Our current implementation integrates two pre-trained models, CodeT5 [86] and CodeBERT [36], which we fine-tune for our prediction task. Building upon a pre-trained model enables our approach to benefit from this model's abilities in "understanding" both code and natural language. CodeT5 uses the T5 architecture [71], a transformer-based neural network architecture that maps a sequence of input tokens to a sequence of output tokens. CodeBERT follows the BERT [22] architecture and is pre-trained to unmask missing tokens and predict whether a token has been replaced. For both models, LExecutor tokenizes the input and output using the tokenizer that comes with the respective model. To fit the input sequence into the fixed input size of the models (512 tokens), we truncate and pad the code tokens in c_{pre} and c_{post} (Definition 3.2). For training, we use the Adam optimizer with weight decay fix [53]. We leave all hyperparameters at the defaults, except for the batch size, which we reduce to 50 (CodeT5) and 13 (CodeBERT) so it fits into our available GPU memory.

Algorithm 1 Value loading and injection in the runtime engine.

```
Input: Kind $k$ of value, model input $i$
Output: Concrete value $v$
1:  if $k$ = "name" then
2:      $v \leftarrow$ load value or catch NameError
3:      if no exception while loading the value then:
4:          return $v$
5:      end if
6:  else if $k$ = "attribute" then
7:      $v \leftarrow$ load value or catch AttributeError
8:      if no exception while loading the value then:
9:          return $v$
10:     end if
11: else if $k$ = "return value" then
12:     if callee $f$ is a regular function then
13:         return $v \leftarrow f()$
14:     end if
15: end if
16: $v_{abstr} \leftarrow model(i)$
17: $v \leftarrow concretize(v_{abstr})$
18: return $v$
```

3.3 Runtime Engine

Once the model is trained, the prediction phase of LExecutor executes possibly incomplete code via learning-guided execution. The core of this step is the runtime engine, which intercepts all loaded values and injects otherwise missing values on demand. To this end, the runtime engine implements the loader functions added to the original code during the instrumentation (Section 3.1). The basic idea is to load and return the original value whenever possible, and to fall back on querying the model otherwise.

Algorithm 1 presents how the engine loads and, if needed, injects values. For uses of variables and attributes (lines 1 to 10), the runtime engine tries to read the value while catching any exception thrown when a variable or attribute is undefined. If reading the value succeeds, then this value is returned and the regular execution of the code continues. Otherwise, the engine will predict and inject a value, as described below. For calls of functions (lines 11 to 15), the algorithm distinguishes two cases. If the callee is a regular function, i.e., a function that was successfully resolved without any help by LExecutor, then the runtime engine simply calls this function and returns its return value. If, instead, the function is a dummy value that was injected by LExecutor because the code tried to read a function value that does not exist, then the engine predicts and injects a suitable return value.

Whenever Algorithm 1 reaches line 16, it has failed to load a regular value. In a regular execution, the code would terminate prematurely in this case. Instead, the algorithm queries the neural model for a suitable value to inject, and the model returns one of the abstract classes in Tables 1 and 2. To enable the code to continue its execution, the runtime engine concretizes the abstract class into a runtime value. In our default configuration, i.e., using the fine-grained value abstraction, the concrete values are those shown in the right column of Table 1. They are simple default values, such as `-1` for negative integers or `"a"` for non-empty strings. Whenever the engine injects an object, it creates a new instance of an empty dummy class `Dummy`. For injecting callables, i.e., in situations where the code tries to access a non-existing function, the engine returns the constructor of the `Dummy` class. The reason is that constructors are the most common type of callable in our dataset.

When using the coarse-grained value abstraction, LExecutor supports two modes for deciding what concrete values to inject. In *deterministic* mode, the approach always injects the value shown in the middle column of Table 2. In contrast, in *randomized* mode, the approach randomly picks from the values shown in the right column of the same table. For example, when the model predicts a string, then the deterministic mode always provides the value `"a"`, whereas randomized mode returns either `""` or `"a"`. To avoid executions that normally would be be impossible, the randomized mode ensures that repeatedly using the same variable returns a consistent value. Future work could further improve the value concretization, e.g., by trying to predict meaningful string values or by predicting the types of values a non-empty list should contain.

Importantly, the runtime engine does not prevent all runtime errors that might occur during an execution. Instead, LExecutor prevents only those exceptions that would be caused by trying to use a non-existing value. That is, code executed with LExecutor may still fail, e.g., due to some logical bug in the code.

Example. Suppose that we are trying to execute the code in Figure 1 during LExecutor's prediction mode. When reaching the use of `all_data` at line 1, Algorithm 1 attempts to load the value of the variable, which leads to a `NameError` because the variable is undefined. The runtime engine catches this error and instead queries the neural model for a suitable value to inject. The model predicts "non-empty list", which Algorithm 1 concretizes, as shown in Table 1, into a list that contains a dummy object. This injected value along with others injected later during the execution enable the code to execute despite some missing values.

4 IMPLEMENTATION

LExecutor is implemented as a fully automated tool that executes arbitrary Python code snippets. To implement the code instrumentation we build upon the LibCST library,[1] which offers a parser and utilities for transforming the AST. The neural models are based on CodeT5 made available by Salesforce[2] and CodeBERT made available by Microsoft[3], both accessed via the Hugging Face API. Due to hardware constraints, we use the "small" CodeT5 model and the "base-mlm" CodeBERT model, and fine-tune them for ten epochs each. To speed up the prediction of values, our implementation loads the trained model once and accesses it via REST queries to an HTTP server. The experiments are conducted on three servers, each with an NVIDIA Tesla GPU (P100, T100, and T4, respectively) using 16GB of GPU memory per server. Deploying LExecutor does not require a powerful server, except for running the neural model.

Table 3: Projects used for gathering training data.

Project	Description	Unique value-use events
Ansible	Automation of software infrastructure	43,090
Django	Web framework	121,567
Keras	Deep learning library	30,709
Requests	HTTP library	5,273
Rich	Text formatting in the terminal	25,370
Total		226,009

Table 4: Projects used for gathering functions.

Project	Description	Functions	LoC
Black	Code formatting	200	2,961
Flask	Web applications	200	1,354
Pandas	Data analysis	200	2,015
Scrapy	Web scraping	200	1,198
TensorFlow	Deep learning	200	2,125
Total		1,000	9,653

5 EVALUATION

We address the following research questions:

- RQ1: How accurate is the neural model at predicting realistic runtime values?
- RQ2: How much code does an execution guided by LExecutor cover, and how does the coverage compare to alternative ways of executing the code?
- RQ3: How efficient is "lexecuting" code?
- RQ4: As an application of the approach, can we use LExecutor to identify semantics-changing commits?

5.1 Experimental Setup

5.1.1 Datasets.

Value-use events. To gather a corpus of value-use events for training the neural model, we execute the test suites of real-world Python projects. We select projects that are popular (based on the number of GitHub stars), cover different application domains, and provide a test suite that is easy to execute yet covers a significant amount of the project's code. Table 3 shows the selected projects and the number of unique value-use events gathered from them. In total, the dataset consists of 226k entries, which we shuffle and then split into 95% for training and 5% to answer RQ1.

Open-source functions. One usage scenario of LExecutor is to execute code that is part of a large project, without having to set up the project and its dependencies, and without having to provide inputs that reach the targeted code. To evaluate LExecutor's effectiveness in this scenario, we gather a dataset of functions extracted from open-source Python projects on GitHub. We search

[1] https://github.com/Instagram/LibCST
[2] https://github.com/salesforce/CodeT5
[3] https://github.com/microsoft/CodeBERT

for projects implemented in Python and sort them by the number of their stars. Then, from the top of the list we choose projects that cover different application domains and that are not used for constructing the dataset of value-use events. Then, for each of the projects, listed in Table 4, we randomly select 200 functions and extract the entire function body into a separate file. The task of LExecutor is to execute the extracted code without access to any contextual information, such as other code in the same project, imports, or third-party libraries. In total, the dataset is composed of 1,000 randomly selected functions, that amount to 9,653 non-empty, non-comment lines of code.

Stack Overflow snippets. Another usage scenario is to execute code snippets that are inherently incomplete, such as code posted in web forums. To evaluate LExecutor in this scenario, we gather a dataset of code snippets from Stack Overflow. Specifically, we search for questions tagged with Python and sort them by the number of votes. Then, from each of the top 1,000 questions, we randomly select an answers and extract the code given in this answer. After discarding code snippets with invalid syntax, the final dataset consists of 462 code snippets. The total number of non-empty, non-comment lines of code is 3,580 lines.

5.1.2 Baselines.

As-is. This baseline executes a code snippet with the standard Python interpreter, i.e., without making any value predictions, and instead letting the code crash whenever it tries to access a missing value.

Naive value predictor. This and the following two baselines are simplified variants of our approach, which use the runtime engine to intercept exceptions caused by undefined references and then inject values instead. The naive value predictor always predicts *object*, and hence always injects a freshly created dummy object.

Random value predictor. This baseline randomly samples (with uniform distribution) a value from the 23 classes of abstracted values available in Table 1.

Frequency-based value predictor. This baseline associates names of variables, attributes, and functions in the training dataset with their distribution of abstract values. When asked for a value for a particular name, it samples from the observed distribution, and falls back to the naive value predictor in case of previously unseen names.

Test generation. This baseline uses Pynguin, a function-level test generator for Python [54]. For a fair, we run Pynguin on a single function at a time, which contains the code to execute only. Usually, test generators are applied to code with all dependencies fully resolved, where they are more effective than when being applied to a single function in isolation. The results of this baseline hence do not represent Pynguin's effectiveness in general, but in a usage scenario that matches that of LExecutor. As many code snippets in the Stack Overflow dataset are not functions, we apply this baseline only to the open-source functions.

Neural type prediction. Because our value abstraction resembles types, this baseline builds on a recent stream of work on predicting type annotations in Python [3, 43, 60, 64, 68]. We compare against the Type4Py model [60] because it outperforms earlier models, and because we can build on their publicly available, trained model. Type4Py predicts types for local variables defined in the code, as well as the parameters and return values of all functions. We concretize these types using the deterministic mapping in Table 2. For values where Type4Py does not predict any type, e.g., undefined variables, this baseline falls back to the random value predictor described above.

Table 5: Accuracy of the neural model.

	Value abstraction			
	Fine-grained		Coarse-grained	
	CodeT5	CodeBERT	CodeT5	CodeBERT
Top-1	80.1%	79.5%	88.1%	87.3%
Top-3	88.4%	94.5%	92.1%	96.5%
Top-5	91.7%	96.8%	94.2%	98.2%

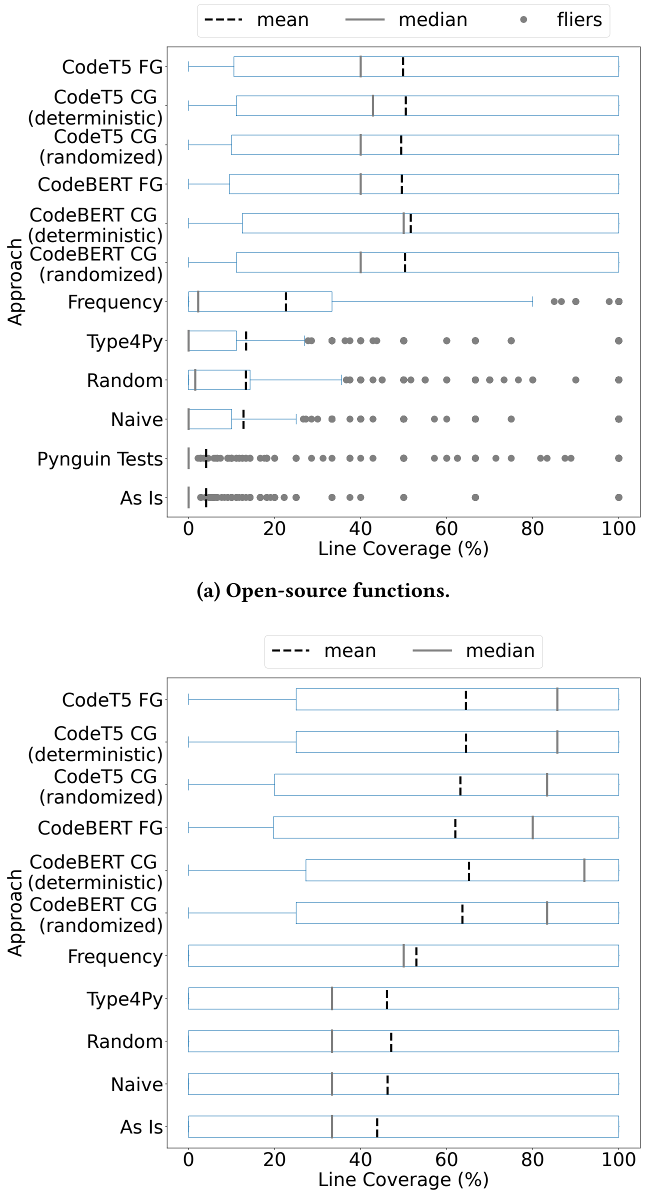
5.2 RQ1: Accuracy of the Neural Model

To measure the effectiveness of the model at predicting realistic runtime values, we compare the model's predictions against values observed during regular executions. Specifically, we compare the predictions against a held-out subset of 5% of the value-use events used for training the model. These values serve as a ground truth because they are obtained by executing developer-written test suites, i.e., realistic executions performed without any guidance by LExecutor.

For each of the two ways of representing concrete values (fine-grained abstraction and coarse-grained abstraction, Section 3.2.2), we train and evaluate a separate model. To measure accuracy, we query the model for the most likely value in a given context and report how often this prediction exactly matches the ground truth (top-1). In addition, we also report how often the ground truth value occurs in the top-3 and top-5 predictions by the model.

The results in Table 5 show that the neural models predict realistic values in the vast majority of cases. The accuracy of the coarse-grained model ranges between 88.1% and 94.2% for CodeT5 and 87.3% and 98.2% for CodeBERT, depending on how many of the top-k predictions are considered. Because the fine-grained models have a harder prediction task (i.e., more classes to choose from), their accuracy is slightly lower and ranges from 80.1% to 91.7% for CodeT5 and 79.5% and 96.8% for CodeBERT. As we will see in RQ2, despite its lower accuracy, the fine-grained value abstraction is overall slightly more effective at guiding executions. Moreover, the accuracy per kind of value achieved by the models is close to the overall accuracy, e.g. the CodeT5 model with fine-grained value abstraction achieves 80.1%, 84.1%, and 76.0% accuracy on name, attribute, and return value predictions, respectively.

To better understand the abilities and limitations of the neural model, we qualitatively study some of its mispredictions. The most common cause is to predict a string when the ground truth value is `None`, and vice versa. These are inherently difficult cases, because practically any value could be `None` in Python. The second most common cause for mispredictions is that the model confuses lists and tuples, i.e., two kinds of values that often can be used



(a) Open-source functions.

(b) Stack Overflow snippets.

Figure 4: Line coverage on two datasets.

interchangeably. Inspecting specific examples of wrongly predicted values shows that the model is wrong mostly in situations where a human also cannot easily decide what the most realistic value is. Specifically, we repeatedly observe two reasons. First, the value is simply not known before loading it, e.g., in `if v is None:`, i.e., a branch depending on the value. Second, the source code and its runtime behavior are inconsistent, e.g., when a docstring says that a function returns a dict, but actually returns a list.

5.3 RQ2: Effectiveness at Covering Code

To measure the effectiveness of LExecutor at successfully executing code, we apply the approach to our two datasets and measure the percentage of all non-empty, non-comment lines of code that are successfully executed. We call a line “covered” if the entire line executes without crashing. We run all three variants of LExecutor, as described in Sections 3.2.2 and 3.3, using top-1 predictions: (i) fine-grained value abstraction, (ii) coarse-grained value abstraction with deterministic predictions, and (iii) coarse-grained value abstraction with randomized predictions. We use the Wilcoxon signed-rank test ($p = 0.05$) to compare the significance of coverage differences between techniques.

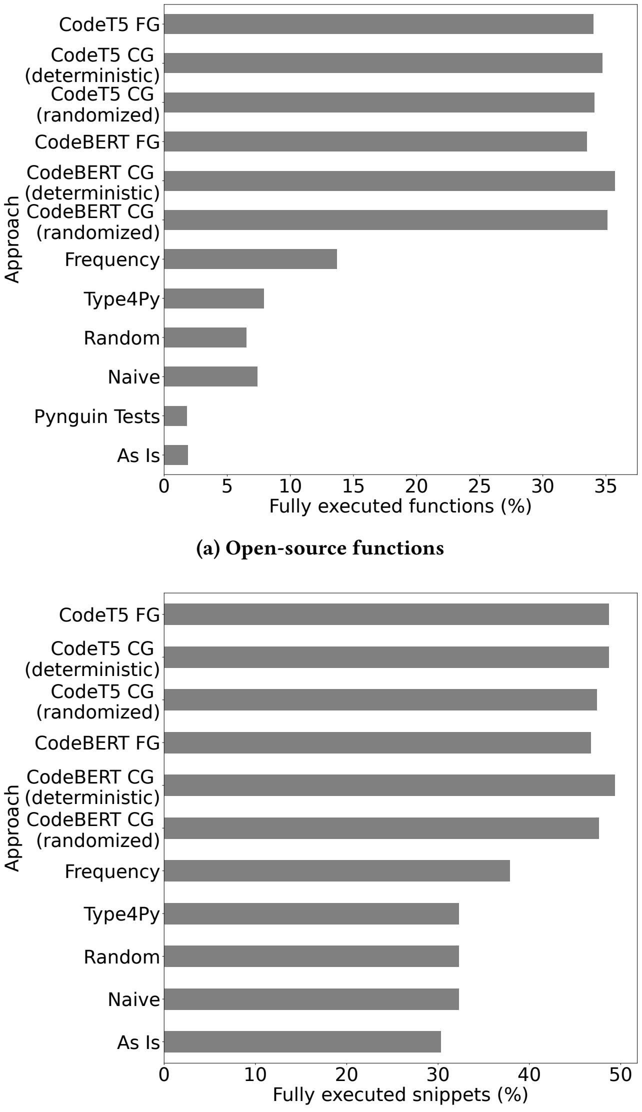
Figure 4a shows the coverage achieved by LExecutor and the baselines on the open-source functions. On average, as-is execution and generated tests cover only 4.1% of the lines (mean across all functions). The naive, random, type (Type4Py), and frequency-based value predictors increase the mean coverage to 12.8%, 13.3%, 13.3% and 22.6%, respectively. We attribute the negligible difference between the random predictor and the type predictor to the fact that the type predictor often falls back to the random predictor, because Type4Py does not predict any type for variables not defined in the given code (which is exactly the problem that LExecutor addresses). Finally, LExecutor covers 51.6% of the lines with CodeT5 and 51.6% of the lines with CodeBERT, which is significantly higher than the other techniques. Figure 4b shows the coverage results on the Stack Overflow code snippets. The coverage achieved by both the baselines and LExecutor is higher, as many of these code snippets are meaningful without additional code context. On average, the open-source functions and the Stack Overflow code snippets contain 13 and seven missing values, respectively. For example, as-is execution and the random predictor cover 43.8% and 46.7% of all lines, respectively. Nevertheless, LExecutor can significantly improve and achieves 65.1% coverage.

Figure 6 shows the code of a function from the Black project, which is difficult to execute due to five missing pieces of information. Running the code as-is crashes on the first line because the `message` variable is missing. When running the code with the random value predictor, it will predict `message` to be an empty dictionary, `styles` to be a non-empty dictionary, and `style` to be an empty tuple. These predictions cause a `TypeError: 'tuple' object is not callable` exception at line 5. In contrast, LExecutor can fully execute the code by filling in the missing values with the realistic predictions shown on the figure.

As another metric of success, we measure how many of all code snippets we achieve 100% line coverage. Figure 5 shows for how many of all considered code snippets the different approaches achieve this goal. Similar to the other coverage results, LExecutor is more effective overall and clearly outperforms the baselines. For example, for the open-source functions, LExecutor fully executes the code of 35% of all functions, whereas the best baseline (frequency value predictor) achieves this goal for only 13% of the functions. These results reinforce that LExecutor significantly improves over alternative approaches.

5.4 RQ3: Efficiency at Guiding Executions

We evaluate the efficiency of learning-guided execution by measuring (i) the time required to instrument the code and (ii) the average time to execute a line of code. The instrumentation takes 38.9 seconds for the 1,000 functions in Table 4 and 20.6 seconds for the

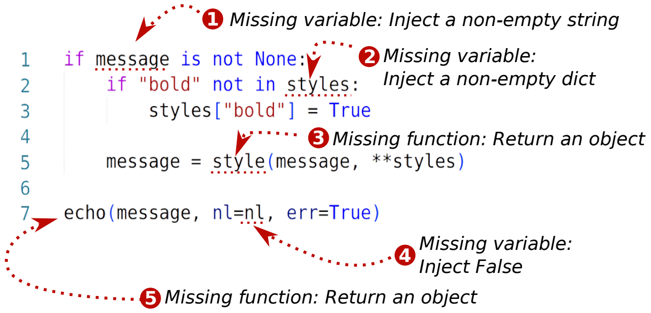


(a) Open-source functions

(b) StackOverflow snippets

Figure 5: Percentage of code examples that are fully executed.

462 Stack Overflow snippets. That is, instrumenting 1,000 lines of code takes 4.5 seconds, on average. Table 6 summarizes the time for executing code within LExecutor and with the baselines. As expected, as-is execution is the most efficient, as it runs code in the bare Python interpreter. The three baselines that inject values without querying the neural model, i.e., naive, frequency-based, and random value prediction, moderately slow down the execution, for example by a factor of roughly 2x for the functions dataset. Because LExecutor queries the neural model for each missing value, it takes the longest to execute the code, and causes a slowdown of one to two orders of magnitude. Taking the results from RQ2 and RQ3 together, an increased execution time is the cost to pay for being able to execute many lines of code at all. We consider LExecutor's efficiency to be acceptable for many dynamic analysis applications, as dynamic analyses often impose similar overheads [24, 49, 78].



```
if message is not None:
    if "bold" not in styles:
        styles["bold"] = True

    message = style(message, **styles)

echo(message, nl=nl, err=True)
```

❶ Missing variable: Inject a non-empty string

❷ Missing variable: Inject a non-empty dict

❸ Missing function: Return an object

❹ Missing variable: Inject False

❺ Missing function: Return an object

Figure 6: Open-source function code to execute and how LExecutor guides the execution.

Table 6: Average execution time (ms) per LoC.

Approach	Dataset	
	Functions	Stack Overflow
CodeT5 FG	178.69	47.29
CodeT5 CG (deterministic)	185.08	46.23
CodeT5 CG (randomized)	167.48	46.31
CodeBERT FG	464.83	133.76
CodeBERT CG (deterministic)	479.89	126.47
CodeBERT CG (randomized)	438.64	127.20
Random	3.94	5.93
Frequency	3.61	5.73
Naive	3.62	5.42
As Is	1.50	5.19

5.5 RQ4: Using LExecutor to Find Semantics-Changing Commits

To illustrate an application of our approach, we apply LExecutor to find semantics-changing commits. Finding such commits is useful to understand whether a commit that changes a function's implementation also changes its behavior. For example, if a commit described as a refactoring is found to actually be semantics-changing, then this commit could be prioritized for manual inspection. However, executing the code involved in a commit usually is possible only if a test suite exists that covers the modified code.

We use LExecutor to execute the code of a function before and after a commit, and then report the commit as semantics-changing if the values returned by the two functions differ. As a dataset, we extract from the 1,000 most recent commits of each of the projects in Table 4 those commits that change a single function from f_{old} to f_{new}. For each such pair, we generate a file that contains f_{old}, f_{new}, and code that invokes both functions and then compares their return values. The consistency mechanism described in Section 3.3 ensures that LExecutor predicts the same values for those parts of the functions that have not changed. We consider two return values to be different if (i) they have different types, (ii) both have a primitive type, e.g., `int` or `str`, but a different value, or (iii) if both values are collections, e.g., a list or a set, but differ in size. In contrast, we do not report a difference if the two values are different objects, as accurately comparing complex objects is difficult.

Comparing executions may lead to three possible outcomes:

Table 7: Results from finding semantics-changing commits.

Project	Commits			
	Total	Exceptional	Same behavior	Semantics-changing
Black	68	41	27	0
Flask	114	78	36	0
Pandas	611	403	207	1
Scrapy	522	292	220	10
TensorFlow	320	241	77	2
Total	1,635	1,055	567	13

```
1  def _retry(self, req, reason, spider):
2    retries = req.meta.get('retry_times', 0) + 1
3
4    if 'max_retry_times' in req.meta:
5      self.max_retry_times = req.meta['max_retry_times']
6
7    stats = spider.crawler.stats
8    if retries <= self.max_retry_times:
9      logger.debug("Retrying ...")
10     # (more code)
11     return retryreq
12   else:
13     stats.inc_value('retry/max_reached')
14     logger.debug("Gave up ...")
```

Figure 7: Commit found to be semantics-changing (the highlighted code was newly added).

(1) *Exceptional*. At least one of the functions raises an exception. We do not draw any conclusion in this case because the exception may be due to an unrealistic value injected by LExecutor.
(2) *Same behavior*. Both functions return the same value, i.e., that the approach has not found any change in the semantics.
(3) *Semantics-changing*. The two functions return different values, i.e., the approach has detected a semantics-changing commit.

Table 7 shows the results. The approach finds 567 commits that preserve the behavior and 13 commits that are semantics-changing. Figure 7 shows an example. The commit adds lines 4 and 5, which may update the value of `max_retry_times`. This value influences the branching decision at line 8, which in turns determines whether the function returns in object or `None`. Usually, executing the old and new code in isolation is impossible, as it requires complex input objects and refers to imported functions, but with LExecutor, we can successfully execute the code. Moreover, the execution of the old code takes the branch at line 8 and hence returns an object, whereas the execution of the new code enters the `else` branch and hence returns `None`, i.e., the commit is semantics-changing.

Being based on dynamic analysis, the approach underapproximates the semantics-changing commits. That is, not finding any difference does not guarantee that a commit is semantics-preserving, as another execution might still expose a behavioral difference.

6 DISCUSSION

Threats to Validity. As a ground truth for judging whether a value is realistic (RQ1), we use values observed during regular executions. However, sometimes more than one kind of value may realistically appear at a given code location, e.g., `None` vs. some other value. As a result, our accuracy metric underestimates the ability of the model to predict realistic values. We implement LExecutor for Python and hence cannot conclude how well our ideas would work in another language. We believe that the approach could easily be adapted to other languages though. In particular, a statically typed language, such as Java or C++, even provides additional information about the values to inject, which would be used to constrain the model's search space. As usual, all conclusions are limited to the datasets we use. To mitigate this threat, we select datasets that cover different usage scenarios and different applications domains.

Limitations. Because LExecutor offers a form of underconstrained execution, it cannot guarantee that an execution may occur in practice. Further improvements of the neural model could address this limitation. In addition, future work could reduce the risk to diverge from a realistic execution by combining LExecutor with executions of existing tests or with constraint-based reasoning. Another limitation is to inject only a relatively small set of concrete values (Tables 1 and 2), which limits our ability to inject realistic values. For example, given a program that usually reasons about strings that contain email addresses, LExecutor can at best inject a string `"a"`, but would never inject a valid email address. We plan to explore continuous value representations and a richer concretization mechanism in the future.

7 RELATED WORK

Neural software analysis. Analyzing software using neural networks has gained a lot of traction over the past few years [67]. One key question is how to represent the code in a way that enables effective neural reasoning, with answers including techniques for representing individual tokens [17, 46, 83], token sequence-based models [34, 35, 43], AST-based models [7, 61, 94], and models that operate on graph representations of code [4, 23, 87]. Neural models of code are shown to be effective for various tasks, e.g., bug detection [32, 44, 51, 70], code completion [6, 10, 48], code summarization [1, 2, 5, 6, 52, 93], program repair [23, 27, 34, 35, 50, 55, 82, 90], type prediction [3, 43, 56, 68, 87], and code search [42, 75, 81]. More recently, researchers have started to explore the predictive power of large-scale, pre-trained models [26, 29, 36, 86, 88], which reduce the per-task training effort, either via fine-tuning [16, 28] or few-shot learning [11, 33, 66]. LExecutor differs by making predictions about runtime values, and by using these predictions to manipulate an ongoing execution.

Neural models of executions. A few neural software analyses focus on executions. Some models predict the behavior of an entire program [13, 92], but are limited to small, synthetically created code snippets that do not suffer from any missing values. Other work passes runtime traces into models [28, 85], predicts whether a code snippet will raise an exception [12], uses runtime information as feedback during training [27], and ask a model to produce a trace of intermediate computation steps [30]. Nalin spots inconsistencies between values and the names that refer to them [63]. Their work exploits the predictability of many runtime values to identify outliers, whereas we exploit this property to fill in otherwise missing values. Compared to all the above work, LExecutor contributes by letting a model predict runtime values to be used when the program would crash otherwise.

Test generation. LExecutor and test generators share the goal of making incomplete code run. Popular approaches include QuickCheck [18], JCrasher [20], symbolic execution [15, 41, 79], Randoop [62], EvoSuite [38], and AFL [14, 91]. Finally, there are test generators for specific kinds of code, e.g., higher-order functions [77], code with asynchronous callbacks [9], or parsers [57, 58]. A key difference to our work is that LExecutor injects values at arbitrary points in the execution on-demand, instead of injecting values at a well-defined interface, such as a function entry point.

Underconstrained execution. Usually, every value the program operates on is known, or otherwise the program gets stuck and is aborted. In contrast, prior work proposes several ways of executing a program without having to fully define all values the program operates on, i.e., forms of underconstrained execution: micro-execution [40], which allows for executing arbitrary x86 code by injecting values into memory on demand; underconstrained symbolic execution [72], which applies symbolic execution to individual functions in isolation, instead of entire programs; and forced execution [47], which artificially forces the program to take not yet explored branches. Our work fundamentally differs from the above by predicting realistic instead of arbitrary values.

Other related work. Some static analyses support code with missing or incomplete type information [21, 59]. In contrast, our work enables dynamic, not static, analysis. Other work heuristically resolves missing dependencies [65, 84]. LExecutor addresses the complementary problem of predicting missing values. A study [45] finds most Python code on Stack Overflow to be non-executable, which LExecutor addresses at least partially. As our work enables executing otherwise non-executable code, it will enable dynamic analyses for Python [24].

8 CONCLUSIONS

Motivated by the recurring need to execute incomplete code, such as code snippets posted on the web or code deep inside a complex project, this paper introduces LExecutor. Our approach prevents exceptions due to missing values, and instead, queries a neural model to predict likely values to use instead. The result is a novel technique for executing code in an underconstrained way, i.e., without a guarantee that the execution reflects real-world behavior, but with a much higher chance of executing without crashing. We envision our approach to provide a foundation for various dynamic analysis applications, because it facilitates the execution of otherwise impossible or difficult to run code.

9 DATA AVAILABILITY

Artifact: https://zenodo.org/record/8270900
Newest version: https://github.com/michaelpradel/LExecutor

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), by the German Research Foundation within the ConcSys and DeMoCo projects, and by the National Council for Scientific and Technological Development (CNPq)/Brazil (Process 162049/2021-8).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *ACL*. 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *ESEC/FSE*. 38–49. https://doi.org/10.1145/2786805.2786849

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *PLDI*. https://doi.org/10.1145/3385412.3385997

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*. https://openreview.net/forum?id=BJOFETxR-

[5] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML*. 2091–2100.

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

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[8] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby.. In *POPL*. 459–472.

[9] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *ICSE*.

[10] Gareth Ari Aye and Gail E. Kaiser. 2020. Sequence Model Design for Code Completion in the Modern IDE. *CoRR* abs/2004.05249 (2020). arXiv:2004.05249 https://arxiv.org/abs/2004.05249

[11] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). https://doi.org/10.48550/arXiv.2206.01335 arXiv:2206.01335

[12] David Bieber, Rishab Goel, Daniel Zheng, Hugo Larochelle, and Daniel Tarlow. 2022. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions. *CoRR* abs/2203.03771 (2022). https://doi.org/10.48550/arXiv.2203.03771 arXiv:2203.03771

[13] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *NeurIPS*. https://proceedings.neurips.cc/paper/2020/hash/62326dc7c4f7b849d6f013ba46489d6c-Abstract.html

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE TSE* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX.

[16] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *ASE*. IEEE, 443–455. https://doi.org/10.1109/ASE51524.2021.9678559

[17] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. In *ICSE*.

[18] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. 268–279.

[19] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *ISSTA*. ACM, 196–206.

[20] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software Prac. Experience* 34, 11 (2004), 1025–1050.

[21] Barthélémy Dagenais and Laurie J. Hendren. 2008. Enabling static analysis for partial java programs. In *OOPSLA*. 313–328. https://doi.org/10.1145/1449764.1449790

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[23] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[24] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *ESEC/FSE*. ACM.

[25] Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *Workshop on Dynamic Analysis (WODA)*.

[26] Daya Guo et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[27] He Ye et al. 2022. Neural Program Repair with Execution-based Backpropagation. In *ICSE*.

[28] Kexin Pei et al. 2021. StateFormer: Fine-Grained Type Recovery from Binaries Using Generative State Modeling. In *ESEC/FSE*.

[29] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[30] Maxwell Nye et al. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. *CoRR* abs/2112.00114 (2021). arXiv:2112.00114 https://arxiv.org/abs/2112.00114

[31] Martin P. Robillard et al. 2013. Automated API Property Inference Techniques. *IEEE TSE* (2013). https://doi.org/10.1109/TSE.2012.63

[32] Marko Vasic et al. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *ICLR*.

[33] Naman Jain et al. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *ICSE*.

[34] Rahul Gupta et al. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[35] Zimin Chen et al. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE TSE* (2019).

[36] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[37] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: interactive program synthesis with control structures. *OOPSLA* (2021). https://doi.org/10.1145/3485530

[38] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11*. 416–419.

[39] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *FSE*. 339–349.

[40] Patrice Godefroid. 2014. Micro execution. In *ICSE*. 539–549.

[41] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. ACM, 213–223.

[42] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 933–944. https://doi.org/10.1145/3180155.3180167

[43] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *ESEC/FSE*. 152–162. https://doi.org/10.1145/3236024.3236051

[44] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *ICLR*. https://openreview.net/forum?id=B1lnbRNtwr

[45] Md. Monir Hossain, Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Abram Hindle. 2019. Executability of Python Snippets in Stack Overflow. *CoRR* abs/1907.04908 (2019). arXiv:1907.04908 http://arxiv.org/abs/1907.04908

[46] Rafael-Michael Karampatsis and Charles Sutton. 2020. SCELMo: Source Code Embeddings from Language Models. (2020). https://openreview.net/pdf?id=ryxnJlSKvr

[47] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *WWW*. https://doi.org/10.1145/3038912.3052674

[48] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *ICSE*.

[49] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ASPLOS*.

[50] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *ICSE*.

[51] Zhen Li, Shouhuai Xu Deqing Zou and, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *NDSS*.

[52] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=zv-typ1gPxA

[53] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR*. https://openreview.net/forum?id=Bkg6RiCqY7

[54] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *SSBSE*. 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

[55] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*. 101–114. https://doi.org/10.1145/3395363.3397369

[56] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *ICSE*. https://doi.org/10.1109/ICSE.2019.00045

[57] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *PLDI*. https://doi.org/10.1145/3314221.3314651

[58] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *ISSTA*. https://doi.org/10.1145/3395363.3397348

[59] Leandro T. C. Melo, Rodrigo Geraldo Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2018. Inference of static semantics for incomplete C programs. *POPL* (2018). https://doi.org/10.1145/3158117

[60] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *ICSE*.

[61] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*.

[62] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*.

[63] Jibesh Patra and Michael Pradel. 2022. Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks. In *ICSE*.

[64] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *ICSE*.

[65] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. [n. d.]. Statistical learning of API fully qualified names in code snippets of online forums. In *ICSE*. https://doi.org/10.1145/3180155.3180230

[66] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *ICLR*. https://openreview.net/forum?id=KmtVD97J43e

[67] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

[68] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *ESEC/FSE*. https://doi.org/10.1145/3368089.3409715

[69] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE*.

[70] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[71] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[72] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX*. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[73] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code".. In *POPL*.

[74] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The ruby type checker. In *SAC*. https://doi.org/10.1145/2480362.2480655

[75] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *MAPL*.

[76] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE S&P*. https://doi.org/10.1109/SP.2010.26

[77] Marija Selakovic, Michael Pradel, Rezwana Karim Nawrin, and Frank Tip. 2018. Test Generation for Higher-Order Functions in Dynamic Languages. In *OOPSLA*.

[78] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE*.

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

[80] Yannis Smaragdakis and Christoph Csallner. 2007. Combining Static and Dynamic Reasoning for Bug Detection. In *TAP*.

[81] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *ICSE*. https://doi.org/10.1145/3510003.3510140

[82] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. (2019).

[83] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *ICSE*. https://doi.org/10.1109/ICSE43902.2021.00059

[84] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *ICSE*. https://doi.org/10.1109/ICSE43902.2021.00144

[85] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *PLDI*. https://doi.org/10.1145/3385412.3385999

[86] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*. https://doi.org/10.18653/v1/2021.emnlp-main.685

[87] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. [n. d.]. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *ICLR*. https://openreview.net/forum?id=Hkx6hANtwH

[88] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *CoRR* abs/2202.13169 (2022). arXiv:2202.13169 https://arxiv.org/abs/2202.13169

[89] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. [n. d.]. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*. 282–291.

[90] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *ICML*. http://proceedings.mlr.press/v139/yasunaga21a.html

[91] Michal Zalewski. 2013. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/

[92] Wojciech Zaremba and Ilya Sutskever. 2014. Learning to Execute. *CoRR* abs/1410.4615 (2014). http://arxiv.org/abs/1410.4615

[93] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *ICSE*.

[94] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation based on Abstract Syntax Tree. In *ICSE*.

Received 2023-02-02; accepted 2023-07-27