ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution

LARS GRÖNINGER, University of Stuttgart, Germany
BEATRIZ SOUZA, University of Stuttgart, Germany
MICHAEL PRADEL, University of Stuttgart, Germany

Code changes are an integral part of the software development process. Many code changes are meant to improve the code without changing its functional behavior, e.g., refactorings and performance improvements. Unfortunately, validating whether a code change preserves the behavior is non-trivial, particularly when the code change is performed deep inside a complex project. This paper presents ChangeGuard, an approach that uses learning-guided execution to compare the runtime behavior of a modified function. The approach is enabled by the novel concept of pairwise learning-guided execution and by a set of techniques that improve the robustness and coverage of the state-of-the-art learning-guided execution technique. Our evaluation applies ChangeGuard to a dataset of 224 manually annotated code changes from popular Python open-source projects and to three datasets of code changes obtained by applying automated code transformations. Our results show that the approach identifies semantics-changing code changes with a precision of 77.1% and a recall of 69.5%, and that it detects unexpected behavioral changes introduced by automatic code refactoring tools. In contrast, the existing regression tests of the analyzed projects miss the vast majority of semantics-changing code changes, with a recall of only 7.6%. We envision our approach being useful for detecting unintended behavioral changes early in the development process and for improving the quality of automated code transformations.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software defect analysis**; *Software evolution*; *Software maintenance tools*.

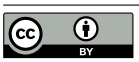
Additional Key Words and Phrases: Code changes, refactoring, differential testing

ACM Reference Format:
Lars Gröninger, Beatriz Souza, and Michael Pradel. 2025. ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE043 (July 2025), 21 pages. https://doi.org/10.1145/3715760

1 Introduction

Successful software projects are continuously evolving. Developers implement new features, fix existing bugs, refactor code to increase its readability, or optimize the performance of a frequently executed code piece. Such changes are performed either by human developers or by automated tools. Regardless of who or what performs a code change, all changes are made to achieve a specific goal, e.g., to fix a bug, improve performance, or make the changed code more readable. Depending on the goal, the semantics of the affected code is supposed to change in a particular way or to not change at all. For example, a refactoring or a performance improvement should not change the semantics, while a bug fix should change the behavior so that the new behavior matches the desired, correct behavior.

Authors' Contact Information: Lars Gröninger, University of Stuttgart, Germany, lars.groninger@gmail.com; Beatriz Souza, University of Stuttgart, Germany, beatrizbzsouza@gmail.com; Michael Pradel, University of Stuttgart, Germany, michael@binaervarianz.de.



ACM 2994-970X/2025/7-ARTFSE043
https://doi.org/10.1145/3715760

```
def _retry(self, request, reason, spider):
    retries = request.meta.get('retry_times', 0) + 1
-   retry_times = self.max_retry_times
-   if 'max_retry_times' in request.meta: retry_times = request.meta['max_retry_times']
+   retry_times = request.meta.get('max_retry_times') or self.max_retry_times
    stats = spider.crawler.stats
    if retries <= retry_times:
        # (19 more, unchanged lines)
```

Fig. 1. Motivating example of a code change meant to be semantics-preserving.

Determining whether a given code change alters the semantics of the code is undecidable in theory and far from trivial in practice. One option is for a human to statically inspect the code change and reason about its impact on the behavior of the affected code. However, this approach is time-consuming, error-prone, and does not scale. Another option is to rely on a regression test suite to exercise the code before and after the change. However, this approach is only feasible if there exist test cases that exercise the changed code locations with various inputs and provide assertions able to detect any behavioral differences. Overall, because validating code changes is difficult, developers may accidentally introduce bugs by making changes that unintentionally modify the behavior of the affected code.

As a motivating example, consider the code change in Figure 1. As evidenced by the commit message "Simplify retry_times assignment statement", the developers intended to improve the code without changing its behavior.[1] However, careful reasoning reveals that the code change actually modifies the behavior of the affected code: If the `'max_retry_times'` of `request.meta` is `0`, then `retry_times` is set to `0` in the old version, but to `self.max.retry_times` in the new version. This change is clearly unintended, yet no existing technique detected it. Interestingly, the developers realized their mistakes several days after the first code change and fixed it.[2] To detect such mistakes earlier, it would be beneficial to have an automated technique that can reason about the behavior of code changes and identify those that modify the behavior of the changed code.

An approach for automatically comparing the behavior of two versions of a code snippet must address two key challenges. The first challenge is to actually execute the changed code, ideally with a diverse set of inputs, to cover as many execution paths as possible. Since code changes may affect arbitrary code locations deep inside a complex project, reaching the changed code from the project's entry point(s) is non-trivial. The second challenge is comparing the behavior of the two versions of the code snippet. This comparison must be able to detect even subtle differences in the behavior, such as changes in the return value, the output printed to the console, the functions called, or the exceptions raised. To the best of our knowledge, there currently exists no approach that can automatically compare two versions of a code snippet and determine whether the code change is semantics-preserving or semantics-changing.

This paper presents ChangeGuard, an approach that compares two versions of a code snippet to determine whether the code change is semantics-preserving or semantics-changing. The approach builds on the recently introduced concept of learning-guided execution [32, 33], which allows executing arbitrary code snippets in isolation by predicting and injecting otherwise missing values. We introduce the new concept of *pairwise learning-guided execution*, which executes two versions of a code snippet side-by-side, while predicting and injecting any missing values to enable executing

[1] https://github.com/scrapy/scrapy/commit/694c6d3d
[2] https://github.com/scrapy/scrapy/commit/49c5afc5

the code snippets in isolation. The approach is enabled by a set of techniques to inject diverse, project-specific, and realistic values, to ensure consistency and non-interference between the two executions, and to handle calls to external functions and indexing operations. For the motivating example in Figure 1, the approach starts to execute both versions of the code at the beginning of the surrounding function, injects several values that would usually be missing and cause the code to crash, until reaching the changed code lines. By comparing the behavior of the modified function before and after the change, ChangeGuard determines that the change modifies the observable behavior of the function, which could have helped the developers to identify this issue earlier.

We evaluate ChangeGuard on four newly gathered datasets of code changes in Python code: a set of 224 code changes extracted from ten open-source projects, which we manually label as semantics-preserving or semantics-changing, and three sets of code changes obtained by applying automated code transformations proposed by a rule-based and two LLM-based tools. Our results show that ChangeGuard effectively identifies semantics-changing code changes with a precision of 77.1% and a recall of 69.5%. In contrast, the existing regression tests of the projects provide only 7.6% recall, i.e., they miss the majority of semantics-changing code changes. We also apply ChangeGuard to code changes created via automated, supposedly semantics-preserving code transformations, many of which the approach shows to be semantics-changing. We find that this effectiveness critically depends on our improvements over the state-of-the-art learning-based execution technique [32], which increase the median coverage of the changed code from 27% to 92%. Finally, we evaluate the efficiency of ChangeGuard and show that it finds semantics-changing behavior within a few seconds.

ChangeGuard is the first to adapt and systematically evaluate learning-guided execution on pairs of code snippets and the practically relevant task of reasoning about code changes. Prior work on learning-guided execution [32] provides a preliminary evaluation on code changes. However, their evaluation on code changes does not compare against a ground truth. Our evaluation shows that out-of-the-box learning-guided execution fails to correctly classify many code changes, e.g., because it does not cover the code paths affected by the change. Our work also relates to efforts toward automated reasoning about code changes, such as heuristics to guide symbolic execution toward modified code [19]. ChangeGuard differs by building on learning-guided execution, which creates more realistic input values and is effective at covering the vast majority of changed code. Another related stream of work is on just-in-time defect prediction [9, 12, 41]. Unlike our work, it aims at determining whether a code change is likely to introduce a defect, rather than whether the code change is semantics-preserving. Finally, this paper also relates to work on identifying equivalent code, e.g., by mining functionally equal code fragments via random testing [11] or by checking whether two functions in binaries are semantically similar [5, 23]. In contrast to those efforts, ChangeGuard focuses on semantic changes introduced by code changes.

In summary, our contributions are as follows:

- Pairwise learning-guided execution, a novel technique for comparing two code snippets.
- Techniques that significantly extend the robustness and coverage of learning-guided execution.
- A novel dataset of 224 manually annotated code changes from ten popular open-source projects.
- Empirical evidence that ChangeGuard can accurately identify whether a given code change is semantics-preserving or semantics-changing.

2 Approach

The following presents our ChangeGuard approach. Given a code change that modifies a function f from f_{old} to f_{new}, the goal is to determine whether the change preserves the input-output semantics of the function. We consider the code change to be semantics-changing if there exists a set of input values used by f that causes f_{old} to produce different observable behavior than f_{new}.

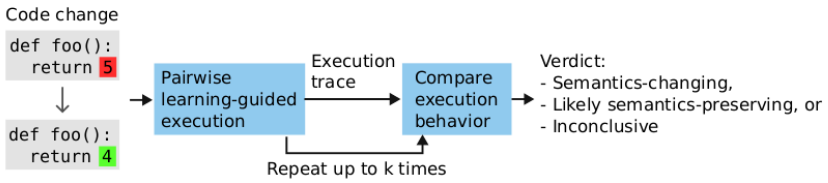


Fig. 2. Overview of ChangeGuard.

2.1 Overview

Figure 2 shows an overview of our approach. Given a code change, the approach determines how the change affects the semantics of the function. The approach consists of two phases. The first phase performs pairwise learning-guided execution, which is the main technical contribution of this work. In a nutshell, pairwise learning-guided execution executes the old and the new version of the function, while injecting any missing values to enable executing the functions in isolation. The second phase compares the execution behavior of the two versions of the function and reports the code change to be either semantics-changing, likely semantics-preserving, or in case the approach fails to perform a meaningful comparison, indicates an inconclusive outcome. Because a single pairwise learning-guided execution reasons about only one of potentially many possible executions, the approach repeats the two phases until it determines the code change to be semantics-changing or until exceeding a configurable limit of k executions.

2.2 Background: Learning-Guided Execution

Before going into the details of our approach, we provide the necessary background on learning-guided execution [32], on which ChangeGuard builds. The key idea of learning-guided execution is to enable the execution of otherwise unexecutable code by predicting any values that are not initialized in the code by querying a machine learning model during the code execution. The machine learning model is fine-tuned version of the pre-trained CodeT5 model [38]. To support learning-guided execution, the model is fine-tuned on a dataset of "normal" executions on the following task: Given a piece of code that accesses a variable, attribute, or calls a function, predict the value that the code would read or observe as the return value of the function call. Learning-guided execution is implemented via source-to-source instrumentation of the code to be executed. The instrumentation intercept all reads of variables, reads of attributes, and calls to functions. If the given code accesses an initialized value or calls an existing function, the instrumented code continues with the regular execution. If, instead, the code accesses an uninitialized value or calls a non-existing function, i.e., the code would usually crash, the instrumented code queries the machine learning model and injects the returned value into the running program. By injecting a (return) value that is likely realistic for the given code, the execution can continue and the code can be executed in isolation.

Prior work has proposed learning-guided execution as a general concept to execute individual code snippets [32]. Our work is the first to adapt and systematically apply this concept to pairs of code snippets and the practically relevant task of reasoning about code changes.

2.3 Pairwise Learning-Guided Execution

The following presents the core technical contribution of this work: pairwise learning-guided execution. The basic idea is to execute two versions of a code snippet side-by-side, while predicting and injecting any missing values into the execution. Like learning-guided execution, the approach

Table 1. Abstract values and their concretizations.

Abstract value	Concrete value(s)
None	`None`
Boolean	`True`, `False`
Integer	`-100`, `-10`, `-1`, `0`, `1`, `10`, `100`, and all integer literals in the code
Float	`-100.0`, `-10.0`, `-1.0`, `0.0`, `1.0`, `10.0`, `100.0`, and all float literals in the code
String	`""`, `"a"`, and all string literals in the code
List	List with random size and elements of a randomly selected type
Tuple	Tuple with random size and elements of a randomly selected type
Dictionary	Dictionary with random size that maps strings to values of a random type
Set	Set with random size and elements of a randomly selected type
Callable	Class of a versatile object with various default methods
Resource	Versatile object with various default methods
Object	Versatile object with various default methods

is designed to execute possibly incomplete code snippets, i.e., code snippets that would likely crash due to missing values when being executed as-is. Unlike existing work on learning-guided execution, we here target a pair of code snippets, instead of a single code snippet. Specifically, we focus on the task of comparing two versions of a function, f_{old} and f_{new}, to determine whether the code change is semantics-preserving or semantics-changing. The following describes the challenges that arise during pairwise learning-guided execution and how ChangeGuard addresses them.

2.3.1 Merging Functions into a Comparison Program. To prepare the given function versions for pairwise learning-guided execution, we first merge them into a single program. The motivation for this step is to enable the execution of the two functions on the same values and to enable the approach to directly compare their return values. Given two functions f_{old} and f_{new}, we construct a program that consists of three parts. First, the program contains the two function definitions, where each function is given a unique name, e.g., `f_old` and `f_new`. The approach removes the formal parameters of the functions, so the functions can be called without arguments. Any usage of a formal parameter in the function body will be handled by injecting values during the execution, following the concept of learning-guided execution. Second, the program contains a call to the old function, `f_old()`, followed by a call to the new function, `f_new()`. Finally, the program contains code to compare the execution behavior of the two functions, which we describe in Section 2.4.

2.3.2 Injecting Diverse, Project-specific, and Realistic Values. Learning-guided execution is based on predicting and then injecting otherwise missing values. The nature of these values is crucial for the success of the execution. However, as we show in our evaluation, using the existing learning-guided execution technique [32] out-of-the-box fails to cover many of the changed code lines, and hence, fails to provide a meaningful comparison of many function pairs. A crucial reason for this limitation is that the values injected by the existing learning-guided execution approach are sampled from only 23 concrete values, such as `1`, `"a"`, `True`, `False`, `None`, an empty list, or a list consisting of a single, simple object. While these values are sufficient to execute some code snippets without crashing, they often fail to cover branches guarded by non-trivial conditions or cause the function to crash before reaching the changed code locations.

ChangeGuard addresses this limitation by extending the existing learning-guided execution approach with a set of techniques to inject diverse, project-specific, and realistic values, as presented

in the following. The basic idea is to let the neural model predict one of twelve abstract values to inject at a specific code location, and to then concretize the predicted abstract value to a concrete value sampled from a (theoretically infinitely) large pool of possible values. Table 1 shows the twelve abstract values and their concretizations. The abstract values correspond to the "coarse-grained abstraction" in prior work [32]. Unlike the prior work, which concretizes the abstract values into one of only 23 fixed values, ChangeGuard samples from many more values. This pool contains a diverse set of hard-coded primitive values, as shown in the table. More importantly, the pool contains project-specific literals, a versatile object with rich default behavior, and randomly constructed data structures, as presented in the following.

The architecture and training process of the neural model is the same as in prior work [32], i.e., a fine-tuned CodeT5 [38] model (Section 2.2). Section 4.1.1 provides more details on the training data.

Project-specific literals. To increase the diversity of values and enable our approach to inject values that are specific to the project under test, we extract literals from the code under test. To this end, ChangeGuard parses the old and the new function and extracts all integer, float, and string literals. For each of these three types, the extracted literals are added to the pool of concrete values of that type. When executing code and injecting a value of a specific type, ChangeGuard randomly samples from all values in the pool of concrete values of that type. For example, suppose a function contains a check `if val == "new"`, then the approach will add the string `"new"` to the pool of strings to concretize to, which enables ChangeGuard to execute the branch guarded by this check.

Generating diverse data structures. A commonly missing kind of value are complex data structures, such as lists, sets, or dictionaries. To vary the size and the content of these data structures, ChangeGuard randomly generates complex values in the following way. At first, the approach determines the size of the data structure by randomly sampling a value between zero and four. The rationale for this heuristic choice is to support sequence unpacking[3] and the fact that unpacking sequences of sizes larger than four is uncommon in practice. Next, ChangeGuard randomly selects the type of values to add into the data structure, where objects are created with 50% probability and otherwise, the approach randomly decides between integers, floats, strings, boolean, and `None`. The reason for biasing the selection toward objects is that, as explained below, ChangeGuard injects objects with a rich set of default behaviors. Finally, the approach fills the data structure with concrete values created by recursively invoking the value concretization algorithm.

For example, consider a code snippet that reads a variable `my_dict` for which the neural model predicts the abstract value "dictionary". ChangeGuard will create a dictionary with a randomly selected size and with randomly initialized key-value pairs, e.g., `{"a": 42, "special": set()}`.

Versatile objects with rich default behavior. Whenever the neural model predicts the abstract value to be a resource or an object, ChangeGuard injects an instance of a special class designed to provide a default object with rich behavior, which we call the *versatile object*. The goal of this class is to mimic the behavior of a wide range of real-world objects, such as built-in types, user-defined classes, or objects from third-party libraries. To this end, the versatile object implements most of Python's special methods[4], such as `__add__`, `__getitem__`, `__xor__`, and `__enter__`, with a default behavior designed to not stop the execution. For example, let `o` be a variable predicted to be an object, which is then used in a statement `x = o + 3`. The class of the versatile object implements the `__add__` method so that it determines that the statement is supposed to be an integer addition (as opposed to, e.g., a string concatenation) based on the value of the other operand, i.e., `3`, and then uses its internal integer representation, e.g., the value `1` to perform the addition. As a result, the

[3] https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences
[4] https://docs.python.org/3/reference/datamodel.html#specialnames

statement will assign `4` to `x`, and the program continues to execute. Based on such implementations of the special methods, a versatile object can be used in a wide range of scenarios, such as addition statements, indexing operations, and when being used as a resource.

2.3.3 Ensuring Consistency and Non-Interference. One of the key challenges for pairwise learning-guided execution is to ensure that the executions of the two functions are consistent without interfering with each other. Consistency here means that both functions should operate on the same values, which is essential to compare their behavior. Non-interference here means that the execution of one function should not affect the execution of the other function, which is important to ensure that any observed behavioral difference is due to the code change and not due to the interaction between the two functions. ChangeGuard ensures these two properties by maintaining a consistency map and by handling renamed variables, as described in the following.

Consistency map with deep copies. To ensure both consistency and non-interference, ChangeGuard maintains a *consistency map* that maps each predicted variable or attribute to a pair of values. The keys of the map are the access path used to refer to the variable or attribute, e.g., `data.ages`. The access path uniquely identifies the variable or attribute, unlike prior work [32], which tries to ensure consistency using the unqualified name of attributes only. Each key in the consistency maps to a pair (v_{old}, v_{new}) of values, where the second element v_{new} is a deep copy of the first element v_{old}. For example, suppose a function is reading an undefined attribute `data.ages` and ChangeGuard injects a mutable list created via `[23, 42]`. The consistency map would then contain the entry `data.ages` $\rightarrow$ (`[23, 42]`, `deepcopy([23, 42])`), where `deepcopy` is a function provided by the builtin `copy` module of Python. The first value of the pair is injected into the execution of the f_{old} function, whereas the second value is injected into the execution of the the f_{new} function. The reason for using a pair of values is to prevent modifications of a value in one function from affecting uses of the value in the other function. Without this separation, the execution of f_{old} could modify the value, e.g., by appending an element to a list, which would then affect the execution of f_{new}.

Handling renamed variables. The consistency map uses access paths, i.e., fully qualified identifier names, to keep the injected values consistent across both executions. However, this approach falls short if an identifier has been renamed as part of the analyzed code change. Naively handling a renamed variable as two separate variables would cause the approach to inject different values, which could lead to detecting diverging behavior even though nothing but the name has changed. For example, consider a code change that renames the variable `data` to `store`. Because the old and the new variable names are different, the approach would create and inject different values for the two variables, which may then cause the two functions to appear to behave differently. To avoid such spurious changes in behavior, we enhance the consistency map to handle renamed variables. Given a mapping of old names to new names, ChangeGuard merges the old and the new variable name into a new unique name and then uses the merged name as the key in the consistency map. For the above example, the consistency map would contain an entry `data_renamed_store` $\rightarrow$ (v_{old}, v_{new}). When the old name is encountered during the execution of the old function f_{old}, the approach looks up the merged name in the map and returns the first element v_{old}, and likewise for the new function. For our experiments, we manually create the mapping of old names to new names, but we envision this to be automated [36] for a real-world deployment.

2.3.4 Handling Calls to External Functions. Because ChangeGuard executes the two functions in isolation, it needs to handle calls to other functions, e.g., functions imported in a third-party project. Our approach follows the general idea of learning-guided execution, which is to predict and inject return values for calls to external functions. We improve this idea in three ways.

```
try:
    parse(code)
except SyntaxError as e:
    print("Caught exception:", e)
```

Fig. 3. Example of handling exceptions triggered by external calls.

Inject exceptions thrown by external functions. The first change is motivated by the fact that calls to external functions may raise exceptions, and that the analyzed function may have code to handle these exceptions. With the existing learning-guided execution approach [32], calls to external functions never raises an exception, and hence, code that handles such exceptions would never be executed. Instead, ChangeGuard tries to mimic exceptional behavior that external functions may trigger. To this end, the approach statically extracts the types of exceptions that are caught in a try-except statement and associates them with function calls in the corresponding try block. Whenever the approach reaches a call to an undefined function, then with a configurable probability (default: 15%), the approach triggers the corresponding exception. The exception is then caught by the surrounding try-except statement, allowing ChangeGuard to reason about any differences in how the two functions handle exceptions. For example, consider the code in Figure 3, where calling `parse` may raise a `SyntaxError`, and assume that the `parse` function is not defined within the analyzed function. ChangeGuard associates the `parse` function with the `SyntaxError` exception, and when the approach reaches the call of `parse`, it will probabilistically raise a `SyntaxError`. As a result, the code in the except block will be executed, and the approach can reason about any differences in the output printed by the two functions.

Type checks. The second change is to replace calls to `isinstance` with a call to a custom function. To illustrate the motivation, consider an expression `isinstance(x, ClassA) and isinstance(x, ClassB)` in the analyzed function. Assuming `ClassA` and `ClassB` are not in a subtype relationship, the expression should always evaluate to false. However, if the approach injects a versatile object for `x` and injects the class of the versatile object both for the `ClassA` and `ClassB`, then the expression would evaluate to true. In other words, because the isolated code execution does not have access to the actual classes, the `isinstance` checks may not behave as expected. To address this problem, ChangeGuard statically extracts all classes that appear in any `isinstance` call in the analyzed functions and randomly assigns one of these types each time one of our versatile objects is created. Our custom replacement function then checks whether the given object is an instance of the pre-assigned type. As a result, each object injected by the approach behaves consistently across all `isinstance` checks and may pass this check even for project-specific classes.

Super calls. Finally, the third change is to replace `super` calls with a call to a custom dummy function. The rationale is that the analyzed function may actually be a method of a class, and the `super` call would then refer to the superclass of that class. However, since we analyze functions in isolation, the surrounding class is not available, and the `super` call would always fail. Instead, our custom dummy function returns the versatile object, and any calls made on the returned object are handled by the standard logic for handling external functions, i.e., ChangeGuard will inject a realistic return value for them.

2.3.5 Handling Indexing Operations. The existing learning-guided execution approach [32] intercepts variable reads, attribute reads, and function calls to inject predicted values. In contrast, the existing approach does not handle indexing operations, e.g., `x[0]` or `x["key"]`. As a result, indexing operations on values predicted by the neural model often fail, causing the execution to crash.

To avoid such crashes, ChangeGuard extends the existing instrumentation and the neural model to handle indexing operations. The change in the instrumentation is to intercept any indexing operation, and in case it would usually crash, to query the neural model to predict a value to inject as the result of the indexing operation. The change in the neural model is to train the model not only on the existing three kinds of values, but also on values that are the result of indexing operations. As an example, consider a code snippet that reads `x["key"]`, where `x` was predicted to be a dictionary. Since the dictionaries injected by the approach are randomly constructed, the `"key"` is unlikely to be present in the dictionary, and hence, the code would most likely raise a `KeyError`. To avoid this crash, ChangeGuard queries our updated neural model to predict a value to inject as the result of the indexing operation.

2.3.6 Covering Different Execution Paths. The overall goal of ChangeGuard is to find behavioral differences between two versions of a function. To achieve this goal, it is essential to cover as many different execution paths as possible. Because the values injected during learning-guided executions are sampled randomly, different executions may follow different paths and trigger different behaviors. The approach randomizes the injected values in two ways. At first, the approach picks one of the abstract values in Table 1 by sampling from the probability distribution produced by the neural model. Then, the approach concretizes the selected abstract value as described in Section 2.3.2. We exploit the randomized nature of the injected values by repeating the pairwise learning-guided execution, which probabilistically, causes different values to be injected. ChangeGuard continues to execute the two functions until it has either identified a behavioral difference, or until reaching a configurable limit of $k = 300$ executions.

2.4 Comparing Execution Behavior

To determine whether the analyzed code change alters the semantics of the function, ChangeGuard compares the execution behavior of the old and the new version of the function. Intuitively, the approach aims at finding differences in the input-output behavior of the two functions. More precisely, the approach compares the behavior of the two functions in four ways: by comparing argument and return values, output written to the console, functions that get called, and exceptions.

2.4.1 Comparing Argument and Return Values. One important aspect in the behavior of a function is the side-effects on the function arguments and on the return values. ChangeGuard compares these values across the two functions by serializing the value for the old and new function, and by checking whether the serialized values are equal. If any of the values differ, then the two functions are classified as semantics-changing. For values that are a versatile object injected by the approach, we adopt an existing approach [34] for recursively flattening an object as a sequence of its attributes, where each attribute is serialized by using its string representation. For values of other types, the approach uses the string representation of the value. To allow the comparison of values not only from regular functions but also from generator functions or asynchronous functions, the approach tries to unwrap the return value before the comparison in case it is a coroutine or generator object. Additionally, if a return value is a regular functions itself, then the approach tries to execute the function without any arguments, in an effort to compare the behavior of the returned function.

2.4.2 Comparing Output. Beyond the return value, the behavior of a function may also be reflected in the output written to the console. To compare the output of the two functions, ChangeGuard captures the standard output and the standard error of the two functions, compares the captured outputs, and classifies the two functions as semantics-changing if the outputs differ.

2.4.3 Comparing Called Functions. To accurately identify a change in behavior between the two versions of the function, the approach compares their potential side effects performed by calling

external functions. Because learning-guided executions abstract away the actual behavior of external functions, the approach cannot directly compare these side effects. Instead, ChangeGuard records a log of all function calls for which the approach injects a return value, along with the arguments passed to these calls. After both versions of the function have executed, the approach compares their logs of calls to external functions. If the logs differ, e.g., in the order of called functions or the arguments passed to the functions, the approach classifies the code change as semantics-changing.

2.4.4 Comparing Exceptional Behavior. Executing the two functions may cause an exception to be raised in none, one, or both of the functions. A simple approach could consider the two functions to be semantically equivalent if either both raise an exception or none raises an exception. However, such an approach would reduce ChangeGuard's ability to reason about subtle differences in the exception-raising behavior of the two functions. Specifically, exceptions can be raised for two kinds of reasons. On the one hand, raising an exception may be part of the intended behavior of the executed code, e.g., to signal an error condition. Such exceptions are part of the behavior that our approach should compare to determine whether the code change preserves the way in which exceptions are raised. On the other hand, some exceptions may be caused by the learning-guided execution itself, e.g., when ChangeGuard injects an unrealistic value that the code is not supposed to handle. Because unrealistic values violate the assumptions of the code, the code may expose arbitrary, possibly exceptional behavior, which should not be compared across the two functions.

To distinguish between exceptions intended by the developer and exceptions caused by our approach itself, we consider two kinds of exceptions to be *intended*. First, ChangeGuard wraps all exceptions created by a `raise` statement with a special class `IntentionalException` and modifies any `except` clauses around the raise statement to catch `IntentionalException`. Second, the approach considers `AssertionError` exceptions as intended exceptions because developers often use them to signal an error condition.

In case at least one of the two functions raises an exception, ChangeGuard compares the raised exceptions by considering the following three scenarios. (1) One function raises an intended exception, whereas the other does not raise any exception. In this case, both versions of the function exhibit different semantics and ChangeGuard classifies the code change as semantics-changing. For example, such a semantic difference may result from a code change that introduces a new error condition or changes the way in which an existing error condition is signaled. (2) Both functions raise an intentional exception. In this case, we compare the type of the raised exception and the arguments passed to the exception. If the exceptions differ, the code change is classified as semantics-changing. (3) At least one unintended exception is raised, which likely is caused by the approach itself. In this case, we refrain from drawing any conclusions about this execution.

2.4.5 Classification of Code Changes. If any of the above comparison steps reveals a difference in the behavior of the two functions, the approach reports the code change to be *semantics-changing*. Otherwise, ChangeGuard continues to analyze the code change until exceeding the budget of k repetitions. If, after exceeding this budget, the approach has not found any behavioral differences, it classifies the code change as follows. If none of the executions have reached any of the changed code lines, or if all executions finished prematurely due to an unintended exception, then the approach classifies the code change as *inconclusive*. Otherwise, i.e., ChangeGuard has successfully exercised the changed code but has not found any behavioral differences, the approach classifies the code change as *likely semantics-preserving*. The reason for saying "likely" is that the approach cannot guarantee that the two functions are semantically equivalent, e.g., because the approach may have missed some execution paths.

3 Implementation

ChangeGuard builds upon, and significantly extends, the open-source tool LExecutor [32].[5] To instrument code, we use the libCST library.[6] For training and querying the neural model, we build upon the Transformers library.[7] The code changes to analyze are automatically extracted from git commits: Given two commit hashes, the approach extracts individual functions that differ between the two commits, and writes the old and new version into a JSON file. Instead of extracting the changes from commits, one could easily create such a JSON file from other code changes, e.g., changes that are not yet committed to any repository. The output of the approach is the verdict, i.e., "semantics-changing", "likely semantics-preserving", or "inconclusive", along with a detailed trace of any observed differences in behavior. For example, if the two executions return different values for the same inputs, the approach prints the inputs and the differing return values.

4 Evaluation

Our evaluation addresses the following research questions:

- RQ1: How effective is ChangeGuard at classifying code changes?
- RQ2: How does ChangeGuard compare to regression testing?
- RQ3: How accurate is the neural model underlying the approach?
- RQ4: How effective is ChangeGuard at successfully executing the changed code?
- RQ5: How efficient is ChangeGuard?

4.1 Experimental Setup

4.1.1 Dataset for Fine-Tuning. Following prior work [32], the neural model underlying ChangeGuard is a fine-tuned CodeT5 model [38]. Because ChangeGuard supports a wider range of values to inject (Section 2.3.5) than prior work [32], we fine-tune our own model. To gather a dataset for fine-tuning, we build upon DyPyBench [1], a benchmark of 50 executable, open-source Python projects. We use 48 out of the 50 projects, excluding Flask-API and Black to avoid overlap with the projects we evaluate on (Section 4.1.2). Using DyPyBench, we manage to collect 250,046 training samples, including 7,502 for the newly added indexing operations (Section 2.3.5). We shuffle and split the collected data, using 95% for fine-tuning and 5% to answer RQ2. We fine-tune the model for ten epochs.

4.1.2 Datasets of Code Changes to Analyze. We apply ChangeGuard to four datasets of code changes, which are extracted from ten popular open-source Python projects, listed in Table 2.

Manually annotated commits. To validate the approach against a ground truth, we manually inspect and annotate 299 code changes, as summarized in Table 2. As a first step, we filter all commits made before November 1, 2023, based on the following criteria: (i) the commit is not a merge commit; (ii) the change affects a single function; (iii) the commit modifies the function, as opposed to adding or removing an entire function; (iv) the changed file is a Python file and does not contain "test" in its name; and (v) the old and the new function parse into different ASTs, even after removing comments, decorators, and type annotations. In an attempt to identify code changes meant to be semantics-preserving, we then filter for commit messages that contain the keyword "refactor", "simplify", "cleanup", or "optimize", which yields a total of 149 code changes. To balance the dataset, we also collect 150 (15 per project) code changes that do not contain the above keywords, and are likely semantics-changing. Because commit messages alone are not a

[5] https://github.com/michaelpradel/LExecutor/
[6] https://github.com/Instagram/LibCST
[7] https://github.com/huggingface/transformers

Table 2. Manually annotated code changes.

Project	Code changes				
	By commit message		Manually annotated		
	Sem.-preserving	Sem.-changing	Sem.-preserving	Sem.-changing	Unclear
Airflow	31	15	15	25	6
Black	3	15	4	13	1
FastAPI	9	15	8	12	4
Flask	3	15	5	10	3
HTTPie	6	15	7	14	0
Pandas	27	15	5	13	24
Poetry	3	15	2	11	5
Scikit-Learn	37	15	23	16	13
Scrapy	20	15	15	12	8
TheAlgorithms	10	15	9	5	11
Total	149	150	93	131	75

reliable way of identifying semantics-preserving changes [22], we then manually inspect each code change and annotate it as either "semantics-preserving", "semantics-changing", or "unclear". This process results in a ground truth of 93 and 131 code changes annotated as semantics-preserving and semantics-changing, respectively.

Rule-based refactorings. In addition to developer-created code changes, we also apply ChangeGuard to validate changes made by automated tools. To this end, we use RIdiom [40], a refactoring tool that transforms Python code into a more idiomatic version. We applying RIdiom to the newer version of the 299 functions in the above dataset and use all of the resulting 165 code changes as an additional dataset. As RIdiom is designed to make the code more idiomatic, while preserving its behavior, we expect these code changes to be semantics-preserving.

Refactorings created by GPT-3.5 and GPT-4. Finally, we apply ChangeGuard to code changes created by large language models (LLMs). LLMs have the potential to support developers by refactoring their code, and we evaluate to what extent ChangeGuard could be used to validate the correctness of such LLM-generated code changes. Similar to the above setup, we ask two of OpenAI's recent models (gpt-3.5-turbo-0125 and gpt-4-turbo-2024-04-09) to refactor the newer version of the functions from Table 2. We use the following prompt: "You are a Python expert. Improve the quality of this Python code while preserving its behavior and without renaming variables, adding comments, adding a docstring, or adding imports: <code>" We discard any code changes that do not modify the AST and code changes that split the given function into multiple functions. This process results in 187 code changes by the GPT-3.5 and 258 code changes by GPT-4.

4.1.3 Hardware. All experiments are performed on a machine with an Intel Xeon Silver 4214 CPU (2.0GHz, 12 cores) running Ubuntu 20.04. We use an NVIDIA Tesla P100 (16GB GPU) and a NVIDIA Tesla T4 (16GB GPU) for fine-tuning and inference, respectively.

Table 3. Effectiveness of ChangeGuard and regression testing on manually annotated ground truth.

Ground truth		ChangeGuard			Regression testing		
	Total	Changing	Preserving	Inconclusive	Changing	Preserving	Inconclusive
Changing	131	91	12	28	10	29	92
Preserving	93	27	48	18	2	18	73
Total	224	118	60	46	12	47	165

```
else: raise AssertionError(
-       "Unexpected IPython magic {node.value.func.attr!r} found. "
+       f"Unexpected IPython magic {node.value.func.attr!r} found. "
        "Please report a bug on https://github.com/psf/black/issues.") from None
```

Fig. 4. Code change intended to change the semantics, which ChangeGuard confirms.

4.2 RQ1: Effectiveness

The following evaluates the effectiveness of our approach at identifying code changes to be semantics-changing or semantics-preserving. We first present the results on the manually annotated code changes, then discuss the results on the code changes created by RIdiom and LLMs.

4.2.1 Results on Manually Annotated Code Changes. Table 3 (center) shows the results of applying ChangeGuard to the 224 code changes that are manually annotated as either semantics-preserving or semantics-changing. For the task of identifying semantics-changing code changes, the precision is $91/(91+27) = 77.1\%$ and the recall is $91/(91+12+28) = 69.5\%$. The overall accuracy across all cases where the approaches gives an answer, i.e., excluding cases where ChangeGuard responds with "inconclusive" is $(91+48)/(118+60) = 78.1\%$. That is, ChangeGuard's predictions are correct for the vast majority of code changes, offering both high precision and high recall.

Figure 4 shows an example where the developers intend to change the semantics, which ChangeGuard correctly confirms to be true. The exception raised by the old code had an incorrect message. The new code fixes this problem by turning the string into an f-string. Because our approach reaches the exception-triggering code and compares the messages, it confirms that the code change is indeed semantics-changing.[8] Our motivating example from Figure 1 is another code change that ChangeGuard classifies as semantics-changing. However, in this case the developers did not intend any change in behavior. This and other examples like it demonstrate that our approach can be useful for detecting unintended changes in behavior.

While the majority of code changes is classified correctly, ChangeGuard also has 27 false positives, 12 false negatives, and 46 code changes classified as inconclusive. We manually inspect these cases to better understand the limitations of our approach. The main reason for false positives, i.e., code changes classified as semantics-changing even though the change preserves the behavior, is external function calls. For 21 out of 27 false positives, the code change moves, removes, or changes a function call. Since our approach reasons about the changed code in isolation, it does not know what side effects, if any, external functions have, and instead classifies any difference in the observed calls as a behavioral difference (Section 2.4.3). Each of the remaining six false positives has a unique reason, e.g., related to how ChangeGuard compares the argument and return values of the

[8] https://github.com/psf/black/commit/72a84d4099f2930979bd1ca1d9e441140b0a304d

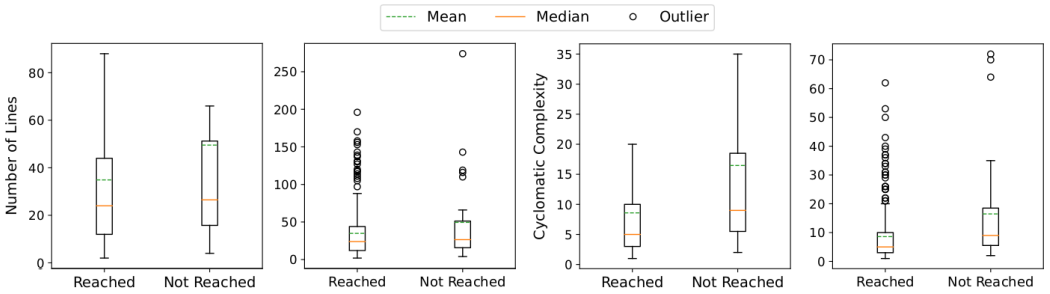


Fig. 5. Impact of function size and complexity on ChangeGuard's ability to reach the changed code lines.

Table 4. Effectiveness on automatically refactored code.

Refactoring tool	Prediction		
	Inconclusive	Changing	Preserving
RIdiom	38	5	122
GPT-3.5	31	87	69
GPT-4	38	143	77

Table 5. Neural model accuracy.

	Accuracy
Top-1	95.13%
Top-3	96.17%
Top-5	97.52%

two functions (Section 2.4.1). False negatives occur when ChangeGuard executes at least parts of the the changed code, but nevertheless cannot observe any behavioral difference. The main reasons for such cases is the behavioral difference manifests only under specific values, or combination of values, but those values are not predicted by the neural model. One way to address this limitation could be to combine learning-guided execution with more systematic, constraint-based reasoning about execution paths. Finally, we also inspect those 46 cases where the approach refrains from making a prediction, but instead responds with "inconclusive". The main reason for such cases is that the approach cannot reach those lines in a large and complex functions that were changed. To corroborate this hypothesis, Figure 5 shows how the number of non-comment, non-empty lines of code in a function and the cyclomatic complexity of the function impacts the chance of reaching the modified code lines. As illustrated by the figure, code changes that are not reached tend to be in larger and more complex functions.

4.2.2 Results on Automatically Refactored Code. ChangeGuard cannot only reason about manually created code changes, but also about code changes created by automated refactoring tools. The following reports the effectiveness of the approach on refactorings created via an existing, rule-based approach and via LLMs (Section 4.1.2). Table 4 summarizes the results.

When applying ChangeGuard to the 165 code changes created by RIdiom[40], the approach confirms 122 code changes to be semantics-preserving, marks 38 code changes as inconclusive, and reports a behavioral difference for five code changes. As the transformations performed by RIdiom are meant to be semantics-preserving, this results mostly aligns with our expectation. One code change classified by ChangeGuard as semantics-changing is shown in Figure 6a. In this example, RIdiom accidentally removes lines during the transformation, presumably due to a bug in the tool's implementation.

The last two rows of Table 4 show the results of applying ChangeGuard to code changes created by LLMs instructed to refactoring the code. Perhaps surprisingly, many of the code changes created

```
- provider = Provider(self._pkg, self._pool, self._io)
- locked = {}
- for package in self._locked.packages: locked[package.name] = package
+ provider, locked = Provider(self._pkg, self._pool, self._io), {}
```

(a) Code change performed by RIdiom.

```
- if isinstance(arr_or_dtype, ExtensionType): return arr_or_dtype.name == "category"
- if arr_or_dtype is None: return False
- return CategoricalDtype.is_dtype(arr_or_dtype)
+ if isinstance(arr_or_dtype, ExtensionType) and arr_or_dtype.name == "category": return True
+ elif arr_or_dtype is None: return False
+ else: return CategoricalDtype.is_dtype(arr_or_dtype)
```

(b) Code change performed by GPT-3.5.

```
  def param_allowed(stat_name, include, exclude):
    if not include and not exclude: return True
-   for p in exclude:
-       if p in stat_name: return False
-   if exclude and not include: return True
-   for p in include:
-       if p in stat_name: return True
-   return False
+   if any(p in stat_name for p in exclude): return False
+   if include: return any(p in stat_name for p in include)
+   return not exclude
```

(c) Code change performed by GPT-4.

Fig. 6. Code changes performed by automated refactoring tools, which ChangeGuard finds to unexpectedly change the semantics.

by the LLMs are classified by ChangeGuard as semantics-changing. For the GPT-3.5-generated code changes, ChangeGuard identifies 87 out of 187 as semantics-changing and only 69 code changes as semantics-preserving. Manually inspecting 30 randomly sampled examples out of the 87 code changes flagged as semantics-changing shows that 21 are true positives, i.e., the LLM indeed changes the behavior of the code. Figure 6b shows an example, where the behavior differs if the `isinstance` check passes but the `arr_or_dtype.name` attribute does not equal `"category"`. For the GPT-4-generated code changes, ChangeGuard classifies 143 code changes, i.e., 54.4% of all code changes, as semantics-changing. We again inspected a random sample of 30 of these code changes and find 16 of them to indeed modify the behavior, despite the LLM being instructed to preserve the behavior. In the example in Figure 6c, GPT-4 tries to simplify the logic of the function. However, if an empty list is passed as an argument to the `state_name` parameter and the `include` parameter and a non-empty list is passed as an argument to the `exclude` parameter, the behavior changes. The remaining 14 code changes are false positives, mostly caused by code changes that modify if and how the code invokes external functions, which ChangeGuard cannot reason about. Overall, our results suggest that LLMs often fail to improve the code while preserving its semantics, and that ChangeGuard provides an effective means to identify such semantics-breaking code changes.

4.3 RQ2: Comparison with Regression Testing

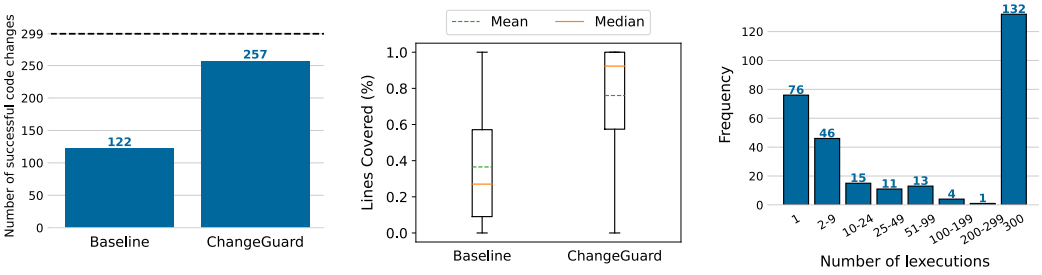
As regression testing currently is the most widely used approach to validate code changes, we compare ChangeGuard's effectiveness against existing regression test suites. Such test suites exist for all analyzed projects (Table 2), except for TheAlgorithms. We perform the comparison for all 224 code changes that are manually annotated as either semantics-preserving or semantics-changing, i.e., where we have a ground truth to use a reference. To check whether the existing regressions tests correctly identify a code change as semantics-preserving or semantics-changing, we take a two-pronged approach: First, we check whether the corresponding commit has any associated continuous integration logs on the GitHub Workflows platform. If such logs exists, we compare the test execution results for the commits of f_{old} and f_{new}. Second, if we cannot find any continuous integration logs, then we try to execute the tests locally as follows: 1) clone the project repository; 2) check out to the commit under analysis; 3) follow the project's instructions on installing dependencies, building, and running the tests. Finally, we compare the test execution results for the commits of f_{old} and f_{new}. Because all code changes affect a single, non-test function, the test cases executed for f_{old} and f_{new} are always the same. Hence, we can directly compare the number of passing and failing tests before and after the code change. If these numbers are the same, the regression tests consider the code change to be "semantics-preserving", and "semantics-changing" otherwise. For code changes without any test execution results, e.g., because the project fails to build, we consider regression testing to be "inconclusive".

Table 3 (right) shows the results of regression testing on the manually annotated ground truth. For those code changes that have regression testing results, the tests correctly identify semantics-changing code changes with a precision of $10/(10+2) = 83.3\%$, a recall of $10/(10+29+92) = 7.6\%$, and an accuracy of $(10+18)/(12+47) = 47.5\%$. Compared to ChangeGuard, regression testing provides a slightly higher precision (83.3% vs. 77.1%), which comes at the expense of a huge drop in recall though (7.6% vs. 69.5%). The reasons for this low recall include that the existing tests do not cover the changed code, the specific commit cannot be built correctly, or that there simply are no regression tests. Out of the ten cases that regression testing correctly identifies as semantics-changing, ChangeGuard also identifies seven of them. Overall, these results show that ChangeGuard is more effective than the existing regression tests at identifying semantics-changing and semantics-preserving code changes, adding benefit over the current state of the art in validating code changes.

4.4 RQ3: Accuracy of the Neural Model

The following two research questions study two important factors that contribute to the effectiveness of the overall approach, starting with the accuracy of the neural model underlying ChangeGuard. We evaluate the model, which is trained to predict an abstract value (Table 1) for a given code context, on a held-out subset of all available data (Section 4.1.1). We report top-k accuracy for different values of k, where a prediction is counted as correct if and only if the correct abstract value is among the top-k values predicted by a the model.

Table 5 shows the results. When considering only the top-most prediction, the model predicts the correct abstract value for 95.1% of the cases. The accuracy increases further when considering more predictions, with the top-5 accuracy reaching 97.5%. As a point of reference, the top-1 accuracy of prior work [32] has been 88.1% in their "coarse-grained" prediction task, which is based on the same twelve abstract values as in our work. There are two main differences. First, ChangeGuard expands the prediction task to also predict values for indexing operations (Section 2.3.5). Second, we use a larger and more diverse training dataset, made available by DyPyBench [1]. Because of these differences, a direct comparison of accuracy values is not meaningful. However, given that



(a) Successfully executed.

(b) Line coverage.

(c) Reaches decision.

Fig. 7. Robustness and coverage compared to the baseline learning-guided execution [32], and number of executions before ChangeGuard reaches a decision, across all 299 manually annotated code changes.

the accuracy of our model is high and higher than in the existing learning-guided execution work, we conclude that the neural model is overall successful at predicting realistic values.

4.5 RQ4: Robustness and Coverage

The following evaluates ChangeGuard's ability to execute the changed code, which is a prerequisite for reasoning about the code change. Specifically, we measure two properties: 1) We assess the robustness of the approach by measuring how often ChangeGuard *successfully executes* the comparison program, where we count an execution as successful if it does not raise any exceptions, except for intended exceptions and assertion violations. 2) We assess how much of the code in the analyzed functions ChangeGuard executes by measuring *coverage*, which we define as the number of executed lines over all code lines. As a baseline, we compare against LExecutor, i.e., the state-of-the-art learning-guided execution technique [32]. For a comparison with other baselines, e.g., executing code by deterministically or randomly injecting values into the code, or via unit-level test generation, we refer to previous results showing that LExecutor outperforms those baselines [32].

Figure 7 summarizes the results. As shown in Figure 7a, ChangeGuard successfully executes 257 out of 299 code changes, enabling it to meaningfully compare the large majority of all code changes. This result significantly improves upon the baseline, which can successfully execute only 122 out of the 299 code changes. The fact that ChangeGuard more than doubles the robustness of the approach compared to the baseline shows the importance of the improvements described in Section 2.3. As shown in Figure 7b, these improvements also increase the coverage achieved by the approach. With a median line coverage of 92%, ChangeGuard exercise the large majority of all code in the analyzed functions. Overall, the high robustness and coverage of our approach is an important ingredient for ChangeGuard's effectiveness.

4.6 RQ5: Efficiency

To quantify how efficient ChangeGuard is at determining whether a code change is semantics-preserving or semantics-changing, we measure the time it takes to instrument and execute the comparison program. Instrumenting the comparison program takes 1.15 seconds, on average. Once instrumented, ChangeGuard repeatedly executes the comparison program until finding a behavioral difference. Because our implementation caches requests to the neural model, most requests to the model happen during the first execution, and we hence measure the time taken by the first and later executions separately. On average, the first execution of the comparison program takes 1.67 seconds, and the remaining executions each take 1.01 seconds.

The overall time for classifying a code change depends on how often the approach repeats the pairwise learning-guided execution. Figure 7c shows that the number of executions required to reach a conclusion follows a bimodal distribution: either the approach quickly identifies that the old and the new function have diverging behavior (left end of the plot), or the approach keeps executing the comparison program without finding any difference (right end of the plot). More precisely, for 76 out of the 299 code changes the approach only needs one execution to identify a change in semantics. Increasing the number k of repetitions yields diminishing returns, with only a single behavioral difference found in executions 200 to 299. In other words, by reducing the number k of executions, one can trade efficiency for a small reduction in recall.

5 Limitations, Threats to Validity, and Future Work

Our approach has several limitations that affect the generalizability of our results. First of all, our implementation targets Python, which, as a highly dynamic language, is particularly attractive for a dynamic analysis approach, but also facilitates the implementation of learning-guided execution. Learning-guided execution has so far been implemented for Python only [32, 33]. Applying it to another highly dynamic language, e.g., JavaScript, seems relatively straightforward, whereas extending learning-guided execution, and by extension also ChangeGuard, to statically typed languages will pose interesting new research challenges.

Second, our work focuses on single-function changes. Reasoning about changes that span multiple functions will require defining "semantics-preserving" for more complex code and new techniques, e.g., to select an entry point for the execution. Going in the opposite direction, future work could also explore to adopt ChangeGuard to code snippets smaller than a single function. For example, if a single expression or a single statement is changed, ChangeGuard could be applied only to the changed code area to check whether its semantics has changed.

As a third limitation, our approach does not consider the side effects of external functions, which is the main cause of false positives (Section 4.2). This limitation could be addressed in the future either by running ChangeGuard in an environment with all dependencies installed or by modeling the side effects of external functions.

Fourth, our approach assumes the changed code to behave deterministically, which has been the case for all code changes in our evaluation.

Finally, our work inherits the general limitation of learning-guided execution to not guarantee executions to be realistic. ChangeGuard mitigates this limitation through several novel techniques that make learning-guided execution more robust (Section 2.3.2–2.3.5) and by training a highly accurate neural model (Section 4.4).

6 Related Work

Reasoning about code changes. Motivated by the importance of code changes, various approaches try to predict their risk [21, 29], e.g., via just-in-time defect prediction [9, 10, 39, 41]. While "risk" and "defect" are defined broadly in prior work, we focus on the orthogonal problem of validating whether a code change modifies the runtime behavior of the code. Another line of work tries to identify breaking API changes, e.g., via static analysis [2] or by observing type signatures during testing [20]. ChangeGuard differs by exercising the changed code in a targeted manner and by performing a detailed comparison of the runtime behavior. DiffSearch [3] is a search engine to find specific code changes. Our work differs from all the above work by using learning-guided execution to reason about code changes.

Checking code for equivalence. Even though the problem of deciding whether two pieces of code are equivalent is undecidable in general, many approaches try to address it approximately. Unlike

approaches that compare functions at the binary level [6, 23], our approach works on source code and executes the code to observe its behavior. Clone detection [25, 26, 28], especially approaches that can detect type-4 clones [8, 13, 15, 16] also rely on static analysis. EQMiner mines functionally equivalent code fragments via random testing [11]. ChangeGuard differs by using learning-guided execution instead of purely random testing, by reasoning about code changes, and by considering side-effects of external functions when comparing the execution behavior.

Refactorings. The ability of ChangeGuard to identify semantics-changing code changes can be used to validate the transformations performed by automated refactoring tools. We explore this usage scenario in our evaluation with code changes created by RIdiom [40] and by large language models. Another refactoring-related line of work is the automatic detection of refactorings [7], e.g., Ref-Finder [14], RefDiff [30, 31], RefactoringCrawler [4], and RefactoringMiner [35, 37]. These techniques are all based on static analysis and detect semantics-preserving changes by looking for specific patterns, whereas ChangeGuard dynamically executes the code before and after a potential refactoring to observe its behavior.

Testing and automatic test generation. The currently most common way to validate code changes in practice is through regression testing. We show in our evaluation that ChangeGuard significantly increases the ability to find behavior-changing code changes compared to the existing regression tests (Section 4.3). Learning-guided execution and automated test generation share the goal of executing code. Several test generators for Python have been proposed recently, e.g., Pynguin [18], CodaMosa [17], CoverUp [24], and SymPrompt [27]. Test generators typically assume that the code under test comes with all dependencies available, and they rely on a test oracle to validate the behavior. Instead, ChangeGuard executes incomplete code in isolation, and it compares the code before and after a change. We refer to prior work [32] for an empirical comparison of learning-guided execution with test generation.

7 Conclusion

This paper introduces ChangeGuard, a novel approach for identifying semantics-breaking code changes using pairwise learning-guided execution. Our evaluation on a diverse set of code changes in popular Python software shows high precision and recall in detecting unintended behavioral modifications, and it demonstrates that ChangeGuard enhances the robustness and coverage of the state-of-the-art existing learning-guided execution technique. We envision ChangeGuard to serve as a validation step both for code changes made by developers and by automated code transformation tools, to ensure that the behavior of the code is preserved.

Data Availability

Our code and data are available at https://github.com/sola-st/ChangeGuard/.

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832) and by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects.

References

[1] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A Benchmark of Executable Python Software. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

[2] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 507–511.

[3] Luca Di Grazia, Paul Bredl, and Michael Pradel. 2022. DiffSearch: A scalable and precise search engine for code changes. *IEEE Transactions on Software Engineering* (2022).

[4] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*. Springer, 404–428.

[5] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 303–317.

[6] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*. 303–317.

[7] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[8] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. 321–330.

[9] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. 34–45. https://doi.org/10.1109/MSR.2019.00016

[10] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[11] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 81–92.

[12] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[13] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. 301–310.

[14] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 371–372.

[15] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, 40–56.

[16] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings eighth working conference on reverse engineering*. IEEE, 301–309.

[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[18] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for python. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 9–24.

[19] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches.. In *ESEC/SIGSOFT FSE*. 235–245.

[20] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd european conference on object-oriented programming (ECOOP 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[21] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[22] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18.

[23] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering* (2022).

[24] Juan Altmayer Pizzorno and Emery D Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv preprint arXiv:2403.16218* (2024).

[25] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[26] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR* 541, 115 (2007), 64–68.

[27] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. *arXiv preprint arXiv:2402.00097* (2024).

[28] Neha Saini, Sukhdip Singh, et al. 2018. Code clones: Detection and management. *Procedia computer science* 132 (2018), 718–727.

[29] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[30] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.

[31] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.

[32] 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*. 1522–1534. https://doi.org/10.1145/3611643.3616254

[33] Beatriz Souza and Michael Pradel. 2025. Treefix: Enabling Execution with a Tree of Prefixes. In *International Conference on Software Engineering (ICSE)*.

[34] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 607–622.

[35] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2020), 930–950.

[36] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[37] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*. 483–494.

[38] 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

[39] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*. 17–26. https://doi.org/10.1109/QRS.2015.14

[40] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 696–708.

[41] Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2023. A Systematic Survey of Just-in-Time Software Defect Prediction. *ACM Comput. Surv.* 55, 10 (2023), 201:1–201:35. https://doi.org/10.1145/3567550

Received 2024-09-13; accepted 2025-01-14