TransMap: Pinpointing Mistakes in Neural Code Translation

Bo Wang
bo_wang@u.nus.edu
National University of Singapore
Singapore, Singapore

Ruishi Li
liruishi@u.nus.edu
National University of Singapore
Singapore, Singapore

Mingkai Li
t0927617@u.nus.edu
National University of Singapore
Singapore, Singapore

Prateek Saxena
prateeks@comp.nus.edu.sg
National University of Singapore
Singapore, Singapore

ABSTRACT

Automated code translation between programming languages can greatly reduce the human effort needed in learning new languages or in migrating code. Recent neural machine translation models, such as Codex, have been shown to be effective on many code generation tasks including translation. However, code produced by neural translators often has semantic mistakes. These mistakes are difficult to eliminate from the neural translator itself because the translator is a black box, which is difficult to interpret or control compared to rule-based transpilers. We propose the first automated approach to pinpoint semantic mistakes in code obtained after neural code translation. Our techniques are implemented in a prototype tool called TransMap which translates Python to JavaScript, both of which are popular scripting languages. On our created micro-benchmarks of Python programs with 648 semantic mistakes in total, TransMap accurately pinpoints the correct location for a fix for 87.96%, often highlighting 1-2 lines for the user to inspect per mistake. We report on our experience in translating 5 Python libraries with up to 1*k* lines of code with TransMap. Our preliminary user study suggests that TransMap can reduce the time for fixing semantic mistakes by around 70% compared to using a standard IDE with debuggers.

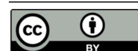
CCS CONCEPTS

• **Computing methodologies → Machine translation**; • **Software and its engineering → Imperative languages**.

KEYWORDS

Code Translation, Large Language Models, Semantic Mistakes

ACM Reference Format:
Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. 2023. TransMap: Pinpointing Mistakes in Neural Code Translation. 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/10.1145/3611643.3616322



This work is licensed under a Creative Commons Attribution 4.0 International License.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616322

1 INTRODUCTION

Automated code translation is the process of transforming code from one programming language to another. Such techniques can help developers express ideas in one language syntax and then easily transfer them to other languages without the encumbrance of learning a new language syntax. Automatic code translation can also lower the cost of migrating code bases from a legacy language to a modern language [22] or from one platform to another [1].

A new approach to program migration or translation has become possible based on neural network models [43, 46, 55]. These techniques were originally developed for natural language generation tasks [6, 30, 42] but also showed promising results in code-related tasks. For example, a state-of-the-art neural code generator Codex [9] is fine-tuned on a popular language model [30] and it backs the AI pair programmer Copilot [20] in Visual Studio Code. It is a general-purpose generative model trained on billions of lines of code from public repositories. It has recently been utilized for translating code between programming languages [4].

Neural code translation offers a promising approach for mostly automatic code translation. These translators are trained in an *unsupervised* manner from examples on a large corpus, saving manual effort in devising translation rules. The translated code is often natural-looking, unlike compiler-generated code. This is because neural code translators can capture the conventions seen in lots of training data derived from real code. Furthermore, state-of-the-art neural translators are derived from large natural language models and can take into account natural language hints like comments or variable naming conventions during translation.

In this work, our goal is to improve neural code translation for code written in scripting languages. We specifically target the translation of Python programs to JavaScript. Both these languages are popular and have many similarities being statically untyped, and therefore we believe offer a concrete starting point.

However, neural code translators can introduce errors in generated code. Prior work has observed that code produced from neural code generators can have syntax mistakes, operator precedence errors, misuse of APIs, incorrect data types, and so on [23, 43]. Mistakes are either syntactic or semantic. Syntactic mistakes violate the target language syntax and can be easily detected by syntax checkers. Semantic mistakes, on the other hand, can result in code that does not execute or executes but outputs wrong values. For many of these, a fix location is also not immediately visible from running

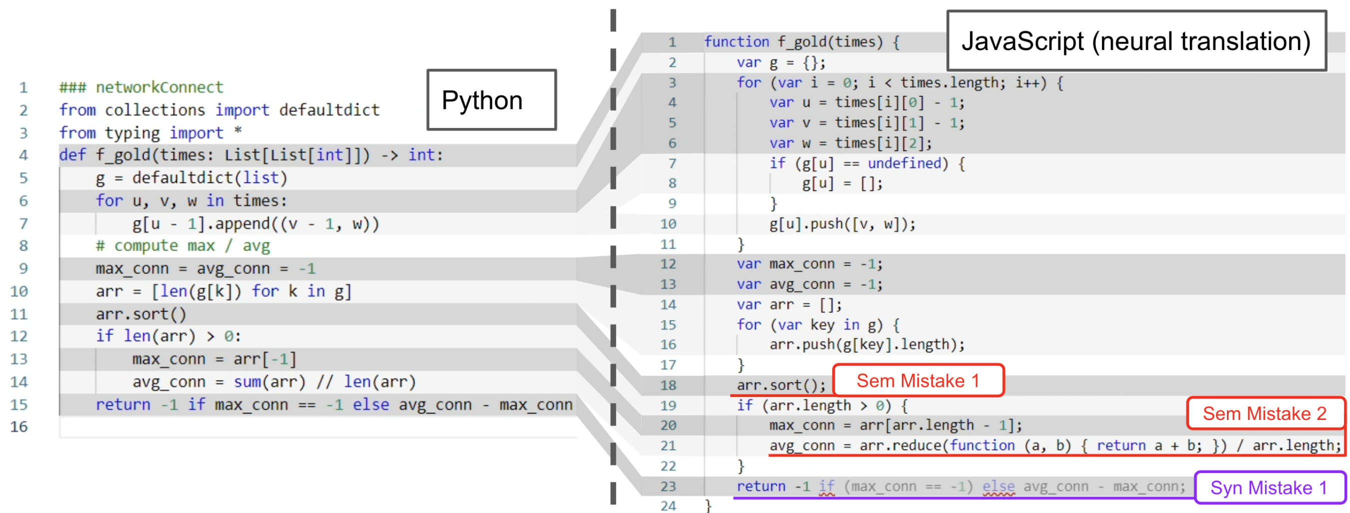


Figure 1: An example of translated code from a neural code generator. The code contains 3 translation mistakes: two semantic mistakes and one syntax mistake. The gray and white shading is the visualization of the source map generated by TransMap.

tests or syntax checkers. We call such mistakes *hidden*. In microbenchmarks discussed later in Section 6, 84.93% of all mistakes are semantic, and more than half of these are hidden mistakes.

In this paper, we therefore focus on the problem of automatically *pinpointing* hidden mistakes in the translated code. Given the source code, its translated code that contains semantic mistakes, and error-triggering tests, the goal is to automatically find the positions in the translated code where small modifications are sufficient to produce translated code that passes the tests. These pinpointed locations serve as an aid to human programmers: They can focus on 1-2 lines of code to fix, instead of the whole input program. While we believe our techniques can be combined with upstream tasks such as automated repair or test generation, we consider these analyses as orthogonal and retain the human developer in the translation loop, keeping with the notion that "the user knows best".

The challenge in pinpointing hidden mistakes when working with neural code generators is that their behaviour is difficult to interpret or explain. Access to state-of-the-art models is often black-box, and even with white-box access, neural networks with billions of parameters are challenging to analyze. Several mechanisms devised in the context of natural language translation tasks, from which neural code translators are derived, do not necessarily lend actionable insights for code translation tasks. For example, the neural attention mechanism can identify which part of the input code the model concentrates on to produce an output snippet [49], but this signal is too noisy and does not explain where the errors are introduced if they are. Furthermore, little modification in the source code or the decoding algorithms [58] of a modern neural generator like Codex can result in unpredictable changes in the translated code, making the process of pinpointing mistakes ad-hoc.

A key technical difficulty in pinpointing translation mistakes is that we have tests that run on two different programs, one written in the source language and the other in the target. This setup is unlike that in fault localization [5, 26, 28, 41, 44, 59] or repair [17, 18, 21, 34, 57] which works with the same program. Therefore, we propose to reconstruct a line-to-line mapping between the source and target program, analogous to "source maps" produced by traditional compilers. Such mappings allow us to compare execution traces run on the source and target programs and narrow down locations to focus on. Our solution requires only black-box access to the neural code generator and avoids making many computationally expensive queries to it. This makes the approach compatible with updates in code generators and lightweight. It also avoids manually writing hard-coded rules for specific types of translation mistakes, heavy-weight analyses, or formal language semantics. Our techniques are embodied in a prototype tool called TransMap, which is short for **Trans**lation with source **Map**.

We create a micro-benchmark for quantitative evaluation, derived from 3 sources of real Python programs: LeetCode [15], HumanEvalX [10], and GeeksForGeeks benchmarks [43]. Each benchmark contains Python source code, translated JavaScript code obtained from querying Codex, passing/failing tests, and manually determined fixes to all the translation mistakes sufficient to pass the tests. There are 648 identified semantic mistakes with manually validated fixes in total. We evaluate TransMap on our quantitative benchmarks and it can successfully pinpoint 87.96% of all the 648 semantic mistakes and 84.44% of all the hidden mistakes. The final reported position ranges by TransMap have an average length of 1.23 lines, which highlights that the developer only needs to focus their attention on a very small range of the code to devise a fix.

We conduct a preliminary user study to use TransMap for debugging code translations with different lengths (10-200 lines) and find that TransMap can save around 70% of the time for fixing the code compared to using an industry-standard IDE (VS Code [32]) with Python and JavaScript debuggers. Furthermore, we report on a qualitative experience by authors in using TransMap to translate 5 larger Python libraries of about 120 to $1k$ lines of code.

2 PROBLEM OVERVIEW

We begin by showing an example for which Codex, a state-of-the-art neural generator, produces a useful initial Python to JavaScript translation. It has several mistakes, the locations of which are not straightforward to pinpoint by just running the given tests and observing the runtime errors. This motivates the need for TransMap.

2.1 Motivating Example

The left part of Figure 1 is an executable Python code that has one function with a list as the input and returns an integer. The right part is the JavaScript code translated from the Python code by Codex. There are some complex tasks during the translation but Codex manages to correctly translate them. Firstly, the Python code creates a `defaultdict`-typed[1] dictionary and uses it on line 5, line 7, and line 10. As JavaScript does not have the equivalent built-in data type, Codex translates the `defaultdict` data type in Python to the `Object` data type with a key-existence check before access, e.g., at lines 7-9 in JavaScript. This type of check is correctly omitted in Codex output when the key is present in the dictionary, e.g., at lines 15-16. Secondly, the Python code has a list comprehension on line 10, which does not natively exist in JavaScript. Codex translates the list comprehension into statements with the same semantics: a `for` loop in JavaScript at lines 14-17. Thirdly, Python APIs, such as `len(arr)`, `sum(..)`, and data type operations, such as `arr[-1]`, are correctly translated into `arr.length`, `arr.reduce(...)`, and `arr[arr.length - 1]` in JavaScript.

While Codex translates most parts of the code correctly, the full translated code is not syntactically correct—a syntax checker can discover a syntax mistake on line 23. Even if the syntax mistake is fixed, the translated code still outputs different results from the source under the same test cases. A careful reader will notice that there are at least two other semantic mistakes, for which small modifications to the JavaScript code are sufficient to make it correct. The 3 mistakes highlighted in Figure 1 are as follows:

- **Syntactic Mistake 1**: the `if-else` to encode a ternary expression violates JavaScript syntax on line 23. The correct translation is `return max_conn==-1 ? -1 : avg_conn-max_conn;`.
- **Semantic Mistake 1**: the `arr.sort()` has different semantics in Python and JavaScript. The corresponding Python code `arr.sort()` on line 11 sorts `arr` in ascending numerical order whereas line 18 of JavaScript sorts elements alphabetically as strings. For example, the sort of an array `[2, 11, 5]` will return `[2, 5, 11]` after executing line 10 of the Python code and `[11, 2, 5]` after executing line 18 of the JavaScript code. The correct translated code should be `arr.sort((a,b)=> a-b);`.
- **Semantic Mistake 2**: the `/` operation in Javascript has a different meaning with `//` operation in Python. Line 15 of Python code means integer division, e.g., `12//5 = 2` but line 21 of JavaScript code is floating point division, e.g., `12/5 = 2.4`. The correct translated code should be `avg_conn = Math.floor(../..);`.

Figure 2 briefly summarizes our evaluation on more benchmarks presented later in Secion 6.1. We find that 84.93% of all mistakes are semantic among which more than half are hidden mistakes. For such mistakes, running the program either throws no runtime errors but produces wrong output values, or it throws runtime errors but the error locations are not where the right fix is needed.

[1]It provides a default value for non-existing keys, instead of raising errors.

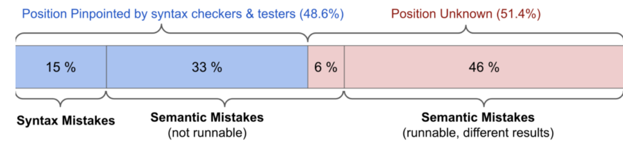


Figure 2: Distribution of translation mistakes in our microbenchmarks presented later in Section 6.1.

2.2 Problem Setup and Challenges

In our setup, we are given the source and target programs generated from the black-box neural generator, together with unit tests. It is straightforward to translate test inputs and outputs across languages; therefore, assume the availability of such tests. Tests under which the outputs of the source and target programs are the same are deemed as *passing*, whereas those on which they differ are deemed as *failing*. Our goal is to pinpoint locations in the translated code where fixes are sufficient to have all given test cases pass.

Black-box neural code generators are neither explainable nor stable to input perturbations. Unlike compilers of which the behaviors can be explained in deterministic transpilation rules, neural code generators can introduce unpredictable changes in the output on small changes in input. For instance, Codex can produce totally different results and mistakes for the same input even with small changes in its auxiliary inputs, such as in sampling methods or sampling temperatures. Expressing the behavior of the generator in deterministic rules for further analysis is difficult.

Another challenge comes from the complexity of the context-dependent semantics of statements in the program. For example, in untyped languages such as Python and JavaScript, the semantics of a statement may depend on types and possible value ranges of related variables, and how a piece of code should be translated would also depend on the earlier translation of related variables. For example, while line 7 of Python (`g[u - 1].append(...)`) is semantically equivalent to lines 7-10 in JavaScript, such equivalence is under the context that `g`'s data type is mapped from `defaultdict` (Python) to `Object` (JavaScript), and the value of `u` in JavaScript is equal to `u - 1` in Python due to the translation at line 4. If `g` were a `dict` in Python and `u` were translated to represent the same value, `g[u-1].append(...)` in Python would be equivalent to `g[u-1].push(...)` in JavaScript instead of lines 7-10. One possible solution to this challenge is to model shared semantics across the two languages with context awareness. But, such specifications are labor-intensive and change as the concerned languages evolve.

In this paper, we aim to provide an **automatic and lightweight** procedure to pinpoint semantic mistakes in neural code translation that is largely agnostic to the internals of the neural generator. The fewer the pinpointed locations, the better—we hope the user's attention can be drawn to as few locations to fix as possible.

Our solution works on the following key principle: Rather than trying to reconstruct externally why the generator produces a certain output, we simply ask the neural code generator to explain itself, i.e., to produce a mapping between the source code and its

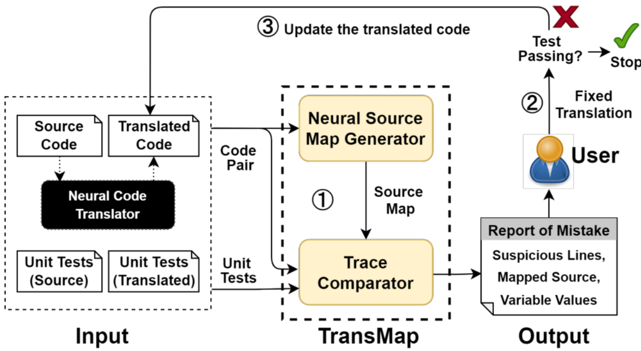


Figure 3: Workflow of TransMap to pinpoint and fix the translation mistakes. TransMap has two key components: the neural source map generator and the trace comparator.

own translated code in a format we designed. We show that such mapping, coupled with an execution trace comparison technique, can automatically pinpoint mistakes with high fidelity.

3 TRANSMAP: OVERVIEW

Figure 3 shows the workflow of TransMap that works in a loop with a user to pinpoint and fix the translation mistakes. The translated code is pre-translated by the neural code generator from the source and they form a code pair. ① The code pair is given as input to our neural source map generator to generate its source map. Given the code pair, its source map, and the unit tests of the source and the translated code, the trace comparator reports the location of the first semantic mistake in the translated code to the user. ② The user fixes the translated code based on the mistake report of TransMap. If the fixed code does not pass the unit tests, it means that the translation has other semantic mistakes. Thus, the translated code would be updated (③) and the next iteration of this process would start from ① to pinpoint the next mistake. The process stops when the translated code has no mistakes remaining, i.e., all tests pass.

Source Map Generation. The first component in TransMap generates the source map, a line-to-line mapping between the given source and translated code pair. Source map creation is standard in compiler-based translators [33, 51], but it has not been demonstrated for neural code translation yet, to the best of our knowledge. Our main insight to solving this challenge is that we can ask the neural code generator to explain itself. We propose a pre-designed prompt to guide the neural code generator to output the mapping. Our observation stems from the *in-context learning* capability of many modern large transformer-based generative models [6, 7, 45], which allows the model to learn a new task from only a few examples provided as "prompts"[6]. We call them *prompt examples*. In-context learning does not need to change the generative model and only requires black-box access to it. For our source mapping task, the in-context learning prompt starts with a *fixed* prompt example of the mapping task that we designed, followed by the pair of programs (S, T) for which we want to create a source map. Our fixed prompt example has a hard-coded pair of source and target programs and the corresponding source map between them

A
```
##### Translate this code from Python to JavaScript
### Python
  from typing import *
  def f_gold(s: str) -> int:
  ...

### JavaScript
"use strict";
  var f_gold = function(s) {
    ...

##### Match the above translation statement by statement
### Python
  ...
    for j, c in enumerate(s):   # --- py stmt 6
      if c == ' ':   # --- py stmt 7
        continue   # --- py stmt 8
  ...

### JavaScript
"use strict";
  ...
    for (var j = 0; j < s.length; j++) { // --- py stmt 6
      if (s[j] === ' ') continue;   // --- py stmt 7, py stmt 8
  ...
```

B
```
##### Translate this code from Python to JavaScript
### Python
  ...
    max_conn = avg_conn = -1                  } S: the given source
    arr = [len(g[k]) for k in g]
  ...
### JavaScript
"use strict";
  ...
    var max_conn = -1;
    var avg_conn = -1;
    var arr = [];                             } T: the given translation of S
    for (var key in g) {
      arr.push(g[key].length);
  ...
##### Match the above translation statement by statement
### Python
```

C
```
  ...
    max_conn = avg_conn = -1   # --- py stmt 8
    arr = [len(g[k]) for k in g]   # --- py stmt 9
  ...
### JavaScript
"use strict";
  ...
    var max_conn = -1;   // --- py stmt 8
    var avg_conn = -1;   // --- py stmt 8
    var arr = [];   // --- py stmt 9
    for (var key in g) {   // --- py stmt 9
      arr.push(g[key].length);   // --- py stmt 9
  ...
```

Figure 4: One-shot in-context learning prompt template (part A and B) for source map generation and its output (part C).

created manually. This example essentially illustrates or "teaches" the model how to create a source map. When we provide this fixed prompt example, together with the programs given as input to TransMap (S, T) to the neural code generator, it mimics the task demonstrated in the fixed prompt example on (S, T). Figure 4 briefly shows the prompt template and Section 4.1 details the mechanism.

One may naturally ask why does this approach to source map generation work. The explanation stems from why in-context learning works [60]: in-context learning succeeds when the neural network model is able to infer some shared latent concepts in the

provided prompt examples. The latent concept learnt for translation is the mapping between semantically similar statements in the two languages. Such latent concepts, if learnt for translation, should also be able to provide source line maps. The fixed prompt example teaches the model how to explicitly represent it in outputs.

When designing the prompt, computational costs become a bottleneck. While in-context learning works better with more prompt examples, the computational cost of the attention mechanism in modern transformer-based neural code generators increases non-linearly with prompt length. The model takes longer to translate longer inputs and therefore, the model query interface limits the number of lexical tokens. For example, Codex can only process $8k$ tokens per query at most for the prompt plus the generated text. To reduce the length of the prompt and save more tokens for the generated source map, we only use one fixed prompt example.

With a single prompt example, we cannot "teach" very complex tasks to neural code generators since their limited reasoning and arithmetic computation capability come into play [14]. Thus, even though our prompt example is short, it is empirically selected to minimize ambiguity in the source mapping task—it only requires the neural code generator to copy existing code and annotations to show the line correspondence, without generating any new code or performing complex arithmetic computations. Additionally, we also set the sampling temperature to 0 since the task requires no creativity [36]. With these design choices, we empirically observe that the noise in the generated source maps is quite low, as shown in Section 6.2. Figure 1 visualizes the source map automatically generated using our approach with white and gray shading.

Trace Comparison. After we get the source map, the next step in TransMap is to execute the source and target programs with the given test inputs, which results in execution traces. TransMap compares these execution traces to pinpoint a location where the execution states, i.e., values of traced program variables, *differ.* A simplistic approach to trace comparison would be to trace the source and translated program statement by statement. This approach leads to a large amount of trace data. Furthermore, the intermediate states at each individual statement in a larger block of statements can turn out to be different between the source and target traces, while still *producing the same output* at the end of the block. As an example, after line 14 in JavaScript, the variable `arr` is bound to an empty array, but `arr` is never empty in Python. However, the value of `arr` is changed multiple times before line 17 where it finally has the same value as in Python after line 10.

We can improve the above simplistic approach by using our source map. The source map gives us (possibly noisy) line-to-line mapping which is a natural segmentation in the programs to perform comparisons. Code segments which cannot be sub-divided based on the source map are called *atomic pieces.* We can now record and compare program states between corresponding atomic pieces of the source and the translated code. However, this naive approach can identify spurious locations to inspect.

To illustrate why, consider the translation example in Figure 1 again. Let us assume that the syntax error on line 23 has been fixed by the user so that only semantic mistakes remain. If we trace per atomic piece, we will insert a pair of tracepoints between line 6 and line 7 in both Python and JavaSript. The Line 7 of the Python code maps to lines 7-9 of the JavaScript code, but the program values differ—variable `u` on the Python side is not equal to the variable `u` on the JavaScript side, and that is because the translated code is assigning `times[i][0]-1` rather than `times[i][0]` to `u`. Thus, the naive approach would report a trace discrepancy at lines 3-6 as variables `u` and `v` have different values under almost all executions. However, if we carefully inspect the behavior of the whole loop, lines 3-6 are unnecessary to inspect and would waste user attention. This is because lines 7-10 in the translated code consistently use `u` instead of `u-1`, so the values of local variables after the combined code block (lines 3-10) produce the same result as the Python code. Further, this approach creates large traces, as trace length grows with algorithmic complexity of the code (e.g., with nested loops).

We propose a simple solution we call *dynamic granularity tracing* to address this challenge. Dynamic granularity tracing compares execution states at the boundaries where the semantics of the source and the translated code reach an agreement, hoping to skip past differences in intermediate states. The intuition is to mimic the debugging process of a programmer to some extent where the granularity of "breakpoints" is increased iteratively while we are getting closer and closer to the exact position of the mistake. For example, on the translation in Figure 1, the tracing starts with statements in the function scope at first to trace the program at lines 5, 6, 9, etc. of Python and 2, 3, 12, etc. of JavaScript. Notice that it will not trace inside the first for loop but only compare the program states before and after it. Thus, the trace comparator concludes that the first `for` loop is correct and moves on to spot mistake 1 at line 18. Once the user fixes that, it will report mistake 2 which is in a deeper block scope. The translation passes all tests after mistake 2 is fixed by the user in our example.

4 TRANSMAP: COMPONENTS

Two main components of TransMap, the source map generator and trace comparator, work together to pinpoint locations where the user can focus their attention.

4.1 Source Map Generator

Given the source and the translated code (S, T) by the neural code generator, the goal of the source map generator is to output a mapping between atomic pieces in the source and the translated code. An atomic piece is a tuple $(A[\ell_s], A[\ell_t])$, where $A[\ell_s]$ and $A[\ell_t]$ are ordered lists of statements corresponding to the line number ℓ_s in the source program and ℓ_t in the translated program, respectively. The *source map* is an ordered list of atomic pieces.

As briefly explained, we use in-context learning with prompts to the neural code generator for creating source maps. Figure 4 shows the prompt template that is used for all input programs. Conceptually, this prompt can be divided into two parts. The first part is a *fixed* pair of code fragments that is used to demonstrate the task of creating source maps. Specifically, it has a fixed Python code and its JavaScript equivalent with the source map annotations in comments, as seen in Figure 4, part A. This is the "one-shot" example of the task we want to teach the generator to perform [6]. Note that part A of the prompt does not change for different (S, T).

The second part of the prompt template consists of the given Python source program S and the given JavaScript translated code T,

but without any source map annotations. The neural code generator is expected to learn how to perform the task demonstrated by part A of the prompt template and auto-complete it. The prompt completion if successfully done, would copy the programs (S, T) and add annotations to them which serve as the source mapping. The output section of Figure 4 shows the part completed by the neural code generator. It can be seen that the model has added code comments line-by-line to both the Python program S and JavaScript program T provided to it in part B of the template. From this output it is straightforward to parse the programs, create a list of statements corresponding to each line number specified in the comments, and obtain the source map.

In order to arrive at this specific prompt template, we performed prompt engineering empirically of two kinds: prompt paraphrasing [25] and prompt scoring [13]. For prompt paraphrasing, we mutated both the source mapping instruction (`##### Match ...`) and the comments before statement numbers (`--- py stmt`) to create prompt variations to select from. They are highlighted in green in Figure 4. We tried 32 variations only since each query to the code generator incurs a cost.

We then compared prompt variations using prompt scoring[13]. We define the score of a prompt output as the log probability of filling all the statement numbers correctly (red circles in Figure 4). Specifically, let $C_i = 1$ be the event that the ith statement in the output is correctly mapped (else $C_i = 0$), then we are interested in the event that all C_i equal to 1. The score $F(P, X)$ for a prompt variation P on a specific source map example X is computed as:

$$\begin{aligned} F(P,X) &= \log \Pr\left[\left(\prod_i \mathbf{C}_i\right) = 1\right] \\ &= \log \prod_i \Pr\left[\mathbf{C}_i = 1 \mid \left(\prod_{j<i} \mathbf{C}_j\right) = 1\right] \quad \text{(chain rule)} \\ &= \sum_i \log \Pr\left[\mathbf{C}_i = 1 \mid \left(\prod_{j<i} \mathbf{C}_j\right) = 1\right] \\ &= \sum_i \texttt{token_logprob}(N_i, \texttt{full_correct_output}) \end{aligned}$$

The quantities in the last step are provided by the neural code generator and N_i is the ith statement number token. Multiple values `token_logprob(token, output)` for the same output can be obtained from a single query to Codex. We test each of the 32 prompt variations on 10 distinct source map examples that we manually created for validation. For each source map example, we obtain scores for all prompt variations. For each example X_t, we compute the ranking of candidate P_k as $\text{Rank}(P_k, X_t)$ where higher $F(P_k, X_t)$ is better. We choose the prompt with the highest average rank across the 10 source map examples, computed as $\frac{1}{10}\sum_{t=1}^{10} \text{Rank}(P_k, X_t)$.

4.2 Trace Comparator

Once we have the source map, the trace comparator instruments the two given programs (S, T). It then runs the test cases on them and compares their execution traces. For execution tracing, both programs (S, T) are instrumented with 2 kinds of logging: *tracepoints* and *line coverage*. The tracepoint instrumentation logs program states at the beginning statement of each atomic piece. The tracepoint is thus a 3-tuple (ℓ_s, ℓ_t, d), where ℓ_s is the line number of the first source statement of an atomic piece, ℓ_t is the line number of the first target statement of the same atomic piece, and d is the

Algorithm 1 Dynamic Granularity Tracing Algorithm

```
1:  procedure DynamicGranularityTracing(P_t, P_s, M, T, L_max)
2:      L ← 1                                   ▷ L is the current tracing level
3:      S_s ← {1, 2, ...#Line(P_s)},   S_t ← {1, 2, ...#Line(P_t)}
4:      while L ≤ L_max do
5:          T' = Filter(T, x → x.l ≤ L ∧ x.s ∈ S_s ∧ x.t ∈ S_t)
6:          (Γ_s, Γ_t) ← RunCollectTrace(P_t, P_s, T')
7:          k ← FindDivergingStep(Γ_s, Γ_t)
8:          if k is None then
9:              return ∅, ∅            ▷ No divergence, code is correct
10:         C_s ← GetCoveredLinesInBetween(Γ_s, k − 1, k)
11:         C_t ← GetCoveredLinesInBetween(Γ_t, k − 1, k)
12:         C'_t ← MapSrcLinesToTgt(M, C_s),
13:         C'_s ← MapTgtLinesToSrc(M, C_t),
14:         S_t ← C_t ∪ C'_t,    S_s ← C_s ∪ C'_s
15:         L = L + 1                           ▷ Increase tracing level
16:     return S_s, S_t          ▷ Suspicious lines (source and target)
```

level of the tracepoint. The level of the tracepoint is its lexical scope depth and all local variables accessible in that scope are logged. For example, line 7 in Python is at level 2 and line 3 is at the lowest level 0 (global scope). For a tracepoint to be valid, the scope level for the source statement and the target statement are expected to be the same—if not, we do not trace those statements.

The instrumented logging statements before each tracepoint record all local variables accessible in that scope and their values. However, this is not sufficient to know the set of executed lines after hitting one tracepoint and before hitting another. The executed lines in between are not necessarily the code between two tracepoints due to control-flow transition statement such as `break`, `continue`, and `throw`. We therefore use line coverage instrumentation to get all the executed lines in between, and hence the control flow.

After the code instrumentation, we obtain a pair of instrumented programs that are executed under given tests. As briefly discussed earlier in Section 3, we employ dynamic granularity tracing to minimize spurious locations being reported to the user. Our proposed algorithm for such dynamic granularity tracing and trace comparison is shown in Algorithm 1. It performs multiple rounds of tracing on instrumented programs, with each round increasing the tracing granularity. Line coverage information is small and is always turned on.

The input to Algorithm 1 includes the instrumented source program P_s, the instrumented translated program P_t, the source map M, static metadata T for selectively turning tracepoints on or off, and the maximum tracing level L_{max}.

Specifically, in the very first round, the algorithm starts at tracing level 1 ($L = 1$) with all the lines marked "suspicious", i.e., the set of suspicious lines X_s and X_t are initialized to all lines in the programs (lines 2-3 in Algorithm 1). In this round, all the tracepoints at level 1 are enabled for logging but tracepoints at higher levels are disabled (checked in line 5). For example, this means all local variables at function scope are traced in JavaScript. The tracing is performed by running the instrumented program on unit tests and collecting the logs. The enabled tracepoints in the program will record program states to logs (line 6). The trace logs, represented by Γ_s and Γ_t, are

sequences of log items where each log entry item consists of the tracepoint index and values of local variables. The trace logs Γ_s and Γ_t will then be compared step by step. If no mismatch is found in the traces, it implies that the programs behave the same thus the algorithm halts. Otherwise, the algorithm will find the first mismatching log item and its corresponding diverging tracepoint (line 7). It then obtains the executed lines between the diverging tracepoint and the previous tracepoint where the program states have not diverged from the line coverage information (line 10). Only those lines are now marked suspicious, therefore, the updated X_s and X_t are reduced. This finishes one round of tracing and the algorithm goes on to perform the next round of tracing with an increased tracing level 2 (line 15). In the round with tracing level 2, all the tracepoints outside of the suspicious line set will be disabled. For tracepoints that are still within the reduced suspicious lines set, tracepoints up to level 2 will be enabled (line 5). The second round of tracing can further reduce the set of suspicious lines with more tracepoints concentrated around the unknown mistake. This process repeats until the maximum tracing level is reached, and the final sets of suspicious lines are returned as the result (line 16).

When deciding if the source and translated code diverge at one tracepoint, we resort to the notion *observational equivalence* (up to the given tests) rather than semantic equivalence. We log the data types to their canonical forms and compare the canonical forms instead of the original data types. The idea is to project various data types into a minimum set of simplest data types. For example, `List`, `Array`, `Deque`, and other similar data types can be mapped to JSON array. `int`, `float`, and other numeric types can be mapped to JSON number. This tracing and comparison process does lose some information, but it reconciles the difference in data types across the source and target languages. While our tracing and value comparator implementation has been sufficient for our reported evaluation, it can be improved in future versions of TransMap.

5 IMPLEMENTATION

TransMap uses Codex as the neural code generator with the translation prompt shown in Figure 5. It starts with a fixed pair of minimalistic code fragments that demonstrate the translation task. The next part is the given Python source code that needs to be translated (left-bottom). The neural code generator completes the prompt with translated JavaScript code (right-bottom) as the output.

The source map generator queries Codex for the source map and we use the standard greedy decoding with the temperature set to 0 and p to 1. It parses the generated text from Codex to compute atomic pieces. It assumes that every line in the translation belongs to some atomic piece and that the atomic pieces order the line numbers in Python source and the JavaScript target in the same order, with no discontinued atomic piece that is mixed with other atomic pieces. This assumption enables simpler implementation for our trace comparator and empirically it is satisfied most of the time. The source map generator will abort if it cannot output a valid source map. When updated programs (S, T) are processed, only few lines are updated and so the updated source map can be computed from cached source maps without querying Codex.

The trace comparator is implemented as source-level code instrumentation and offline analysis on trace files. The instrumentation

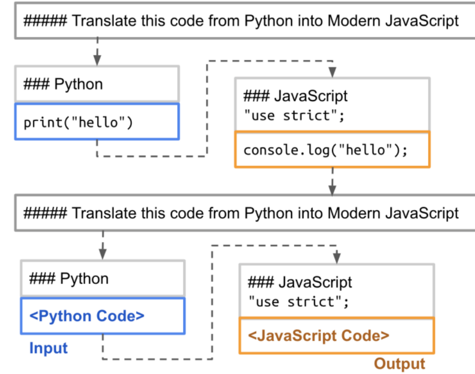


Figure 5: One-shot prompt for the neural code generator to translate Python code into JavaScript code.

requires simple static analysis to get the list of local variables at the position of each tracepoint. This is implemented with the help of tree-sitter [50] AST queries and Python's built-in support for reflection at runtime (e.g. `local()`).

TransMap reports the suspicious lines as given by Algorithm 7 and the variables with diverging values. The trace implementation includes runtime type information which is considered during value comparisons[2]. When a trace mismatch is found, TransMap provides both the suspicious variable and the "Jump-to-definition" utility to the user. The user can fix the variable type declaration, its type definition, or its use at the suspicious lines.

In total, TransMap is implemented in around 5k lines of Python and JavaScript and another 5k lines of UI code for easier interaction.

6 EVALUATION

We evaluate TransMap for identifying mistakes in neural code translation from Python to JavaScript. Our evaluation focuses on the following four aspects:

(1) **Motivation (Section 6.1):** What kinds of translation mistakes does Codex make for which TransMap is of value?
(2) **Effectiveness (Section 6.2):** How effective is TransMap at pinpointing translation mistakes?
(3) **Case Study (Section 6.3)** How much human effort does it take to translate real-world programs using TransMap?
(4) **User Study (Section 6.4)** How helpful is TransMap for users to pinpoint and fix the translation mistakes?

Micro-benchmarks for Pinpointing Neural Translation Mistakes. Pinpointing mistakes in neural code translation across languages is relatively new. We thus created a set of benchmarks that extends those used in recent works [52]. Each sample program in the micro-benchmarks contains the following:

- source program,
- a neural translated program from Codex with mistakes,
- test cases for the source and translated code,
- a list of mistakes in the code with line locations and fixes,
- the fixed translated program that passes the tests.

[2]Similar types like array, queue, and list are clustered as the same "simple type" for comparisons in our implementation.

We create our micro-benchmarks based on 3 popular benchmarks on code related tasks: LeetCode Python-to-JS benchmarks (1067 programs) [52], GeeksForGeeks benchmarks (699 programs) [43], and HumanEvalX benchmarks (164 programs) [10]. Taking the LeetCode benchmarks as an example, the process to create our micro-benchmarks is shown below. First, we query Codex to translate all programs in the LeetCode benchmarks into JavaScript using the translation prompt in Figure 5. The provided test cases for the source programs are simple, thus we obtained their corresponding versions for translated code using straightforward string substitution. Next, we run the translated code on unit tests and collect the failing programs. This gives us 424 programs. Then we manually check these failing programs and filter out those (132 of them) where the translated code is either *unfixable* or not corresponding to the source at all. We consider a program as unfixable if completely rewriting it or implementing non-existing functions and data types used by the translation is necessary to make it pass the tests. Further discussion on these is presented in Section 6.5. After the filtering, we manually check the remaining 292 programs, pinpoint their mistakes, and write a fix for each mistake. In the end, we validate 459 mistakes from 292 programs on LeetCode benchmarks.

Similarly, we collect 235 mistakes from 136 programs on GeeksForGeeks benchmarks and 69 mistakes from 51 programs on HumanEvalX benchmarks. In total, we have 479 programs and 763 mistakes with fixes (115 syntax mistakes and 648 semantic mistakes). The translated programs are about 17.53 lines on average as shown in column ALC_{js} of Table 1 and the longest program has 66 lines. The detailed characteristics of the micro-benchmarks are provided in the supplementary materials [2].

Evaluation Metrics. To evaluate the effectiveness of TransMap, we check if TransMap can pinpoint the semantic mistakes in our micro-benchmarks. For the program with more than one semantic mistake, we generate its partially-fixed variant programs in which only one semantic mistake exists. This gives us as many variant programs as semantic errors in this program. We look at the suspicious lines highlighted at the end. If the suspicious lines contain the mistake, we count it as successful. We also compute the average number of suspicious lines that the user needs to inspect, as well as the average ratio between the suspicious lines and the total lines in the program.

Baseline. The baseline approach is simply to run the given unit tests without TransMap. If a runtime error appears, we check if that line number is the location where the fix is needed. If not, it is counted as a failure. Note that for hidden mistakes, where program states differ but no runtime errors are thrown, the baseline approach cannot pinpoint them.

System Specification. We run all experiments on a desktop with 32GB of RAM and an i7 9700 8-Core CPU. To query the Codex model for generating translations and source maps, we directly use OpenAI API without local computation [37].

6.1 Quantifying Mistakes in Codex Translations

It is useful to understand the kind of errors made by Codex before we can quantify where TransMap provides most value. Figure 6 shows the distribution of all the 763 translation mistakes in all our micro-benchmarks. Among them, syntax mistakes and semantic mistakes account for 15.07% (115) and 84.93% (648), respectively.

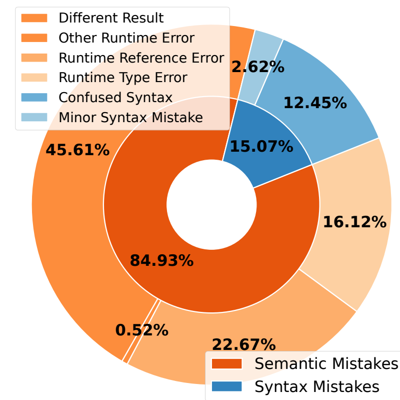


Figure 6: Distribution of mistakes in our micro-benchmarks

A syntax checker spots syntax mistakes even without TransMap. Most syntax mistakes (82.61%) are due to Codex confusing JavaScript syntax with Python syntax and producing Python-like expressions in JavaScript. For example, the expression `[int(x)for x in y]` is wrongly translated into the expression `[parseInt(x)for x in y]`. There is no list comprehension syntax support in JavaScript thus the correct translation should be `y.map(x => parseInt(x))`. These mistakes are not minor lexical edits, but can often be resolved with an online search of the error description [38]. A minority of the syntax mistakes are due to minor errors (e.g., missing parentheses). For example, Codex translates for loops `for a, b in arr` into `for (let a,b of arr){...}` but misses a pair of square brackets around `a,b` (i.e., `for(let [a,b] of ...`).

Around half of the semantic mistakes (45.61% out of 84.93%) cause no runtime errors but result in different results from their corresponding Python source. For the other half of the semantic mistakes, most of them cause either reference errors or type errors. The detailed distributions of mistakes in each micro-benchmark are shown in the supplementary materials [2].

Semantic mistakes can be more subtle and can result in runtime errors (39.32% of all mistakes). We divide runtime errors caused by semantic mistakes into three types: runtime reference error, runtime type error, and other runtime errors. They account for 22.67%, 16.12%, 0.52% of all mistakes, respectively, as shown in Figure 6. An example of runtime reference error is using undeclared variables or non-existing functions. Misuse of API and operators or accessing a numeric value as if it is an object will result in runtime type errors. Occasionally, the translated program might not terminate and exceed the maximum call stack. Most of such mistakes that cause runtime errors can be fixed by changing the API calls, operators, or adding variable declarations at the runtime error location. However, the locations of 14.67% runtime errors (6.79% of all semantic mistakes) do not match the correct fix locations.

The other 53.70% of the semantic mistakes (45.61% of all mistakes) cause no runtime errors but make the translated code differ in output of the unit tests from the source. This type of mistake can be further divided into several sub-types:

Table 1: Distributions of mistakes on micro-benchmarks and performance of TransMap compared to the baseline approach

Micro-Benchmarks	Mistaken Prog.		Mistakes		Baseline	TransMap				
	Count	ALC_{js}	Syntax	Semantic	S_{sem}	S_{sem}	S_{hid}	S_{dif}	L_{sus}	R_{sus}
Leetcode	292	20.26	86 (18.7%)	373 (81.3%)	42.09%	**87.67%**	82.41%	81.96%	1.31	7.23%
GeeksForGeeks	136	13.05	16 (6.8%)	219 (93.2%)	39.27%	**88.58%**	86.47%	88.50%	1.11	7.44%
HumanEvalX	51	11.49	13 (18.8%)	56 (81.2%)	23.21%	**87.50%**	88.37%	87.80%	1.18	10.90%
Total*/Average	479*	17.53	115*	648*	39.51%	**87.96%**	84.44%	84.77%	1.23	7.62%

- **Semantic confusion**, i.e., confusing similar APIs and data types. Such as translating from `int(x)` into `Math.floor(x)` or `Math.trunc(x)` (different when x < 0), `a[i]` to `a.at(i)` or `a.get(i)`.
- **Wrong assumptions** about the JavaScript runtime and APIs. Such as translating `a,b = c.popleft()` to `a = c.shift()[0]; b = c.shift()[1];` (`shift` is not pure and will change c), translate `not arr` to `!arr` (The equivalent expression should be `arr.length === 0` when `arr` is an array).
- **Others**, i.e., mistakes that seem to be difficult to cluster or categorize. Such as missing a function call in the translation, assigning absurd values (such as `[[0], [0]]`) to variables, and introducing a temporary variable with the same name as a local variable.

> 6.79% of the semantic mistakes in our micro-benchmarks cause runtime errors at locations different from the mistakes. 53.70% of the semantic mistakes cause no runtime errors but give different results. TransMap is aimed at pinpointing such *hidden* mistakes.

6.2 Effectiveness of TransMap

We evaluate TransMap on the number of lines that the user has to inspect to fix each mistake in our micro-benchmarks. If the lines highlighted by TransMap contain the location to fix, the pinpointing is deemed successful and a failure otherwise. Our results on the micro-benchmarks are shown in Table 1, where S_{sem}, S_{hid}, S_{dif} represent the ratio of successfully pinpointed mistakes to the total semantic mistakes, hidden mistakes, and mistakes that cause different output results from the source respectively. The quantities L_{sus} and R_{sus} represent the number of highlighted suspicious lines and its ratio to the program size, averaged over the semantic mistakes successfully pinpointed by TransMap.

Among the set of semantic mistakes, TransMap can pinpoint 87.96% of them successfully. In contrast, the baseline successfully pinpoints only 39.51% of the semantic mistakes. Thus, TransMap significantly improves over the baseline. Its success ratio can achieve 95% satisfaction rate among developers according to this work [29]. The suspicious code lines L_{sus} by TransMap are 1.23 lines, which is 7.62% of the program code lines on average. This means that the user often only needs to focus on 1-2 lines to understand the mistake and to fix it. It shows that TransMap can efficiently pinpoint the semantic mistakes for short code fragments with high accuracy.

TransMap performs well on the hidden mistakes that the baseline approach cannot find at all. As shown in S_{hid} column of Table 1, 82.41%- 88.37% of hidden mistakes can be pinpointed among three micro-benchmarks. For semantic mistakes that do not cause any runtime errors but different results from the source code, TransMap is able to successfully diagnose 84.77% mistakes (S_{dif}) on average.

We also look at the quality of the generated source map specifically. We find that 93.8% of the generated source maps are correct. Failing cases are discussed in Section 6.5.

> 87.96% of semantic mistakes in our micro-benchmarks are pinpointed successfully by TransMap, compared to 39.51% by baseline. The user inspects only 1.23 lines on average per mistake.

6.3 Case Studies

How much does TransMap help in translating larger real-world Python programs to JavaScript? We report on the qualitative experience of translating 5 Python libraries using TransMap as an aid. Their LoC (excluding tests) and the number of semantic mistakes are shown in Table 2. We select those modules because they are standalone, have no external dependencies, and their code can be segmented and translated function-by-function without significant loss of context. We had no apriori familiarity with these libraries.

We explain our methodology in detail using the first module (`strsimpy`) as an example. The `strsimpy` library implements many string functions to compute similarity and distance measures. It also comes with unit tests. It has 800+ stars on GitHub and around 20k downloads per week[3]. It contains around 1k lines of code distributed in 35 Python files and has no third-party dependencies.

Before translating this Python library to JavaScript, we first merge the related Python program files into standalone programs and convert unit tests to JavaScript. More specifically, we manually merge the 35 files with their unit tests into 5 self-contained program files according to their dependency relation. We first convert unit tests in Python into JavaScript using Codex and manually post-process to ensure the tests are translated correctly. This is fairly easy since the unit tests are mostly function calls and assertions that are straightforward to convert. It is worth noting that all these manual efforts on merging files and converting unit tests can be further reduced with sufficient engineering when integrating TransMap into modern IDEs or in other build environments.

Each of the 5 self-contained program file is around 200 lines. If we translate one file and create its source map in one attempt, it would exceed the token limit of the Codex API. We observe that 50 is a feasible line number of the source code to be translated and mapped in one query. Thus, we split the code into segments of around 50 lines while preserving the boundaries of classes and functions. Then, each segment is translated with Codex, source-mapped separately by TransMap, and then merged back after finishing all segments. These steps are fully amenable to complete automation.

[3] https://github.com/luozhouyang/python-string-similarity. The download statistics were obtained from https://pypistats.org/packages/strsimpy on 28 January 2023.

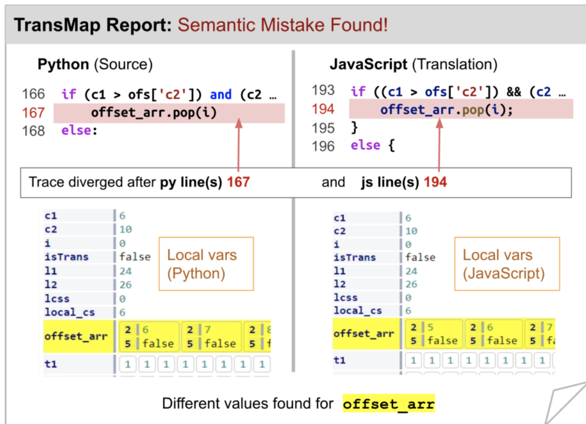


Figure 7: An example report of a hidden semantic mistake pinpointed by TransMap. It contains the mistake location and variables with differing values in the source and the translated code.

We then run programs and fix the syntax mistakes caught by syntax checkers so that only semantic mistakes are remaining. After that, we start TransMap to work with the user iteratively to pinpoint (by TransMap) and fix (by the user) semantic mistakes following the workflow shown in Figure 3. In each round, TransMap outputs the suspicious lines that contain the first semantic mistake found in the translated code, together with the detailed information of the diverging tracepoint (e.g., the mismatching between the expected value in the source and the actual value in the translated code of variables), and the corresponding source lines in Python according to the source map. For example, Figure 7 shows the information reported by TransMap on one of the semantic mistakes in this case study. TransMap pinpoints the mistake at the JavaScript code line 194 with the corresponding Python code line 167. Besides, it gives the local variable values in two code and highlights the inequivalent variable `offset_arr`. With this report, the human user is expected to figure out that the mistake is caused by the semantic confusion of `pop(...)` in JavaScript and provide the fix: Line 194 in JavaScript should be `offset_arr.splice(i, 1)` that removes the i-th element from `offset_arr`.

TransMap correctly pinpoints the location of all of the 13 hidden mistakes in the translated code automatically. With those mistakes highlighted from TransMap, one of the authors of this work with no apriori experience with the library, fixed all of them in about 1 hour. The final JavaScript translation passes all the given tests.

In addition to `strsimpy`, results of 4 more case studies are shown in Table 2. TransMap correctly pinpoints 128 of the 130 hidden mistakes and has 12 false positives (FPs). FPs are due to type differences of some variables. The user can disable tracing them to continue debugging. The supplementary material gives more details.

> TransMap pinpoints 128 of the 130 hidden mistakes in translations of 5 Python libraries, with 12 false positives.

Table 2: Python libraries for case studies: LoC means Lines of Code, Sem. M. means the number of semantic mistakes, #Hid. means the number of hidden mistakes, #TP and #FP mean true positives and false positives of TransMap respectively.

Module	LoC	Sem. M.	#Hid. / #TP / #FP
strsimpy	926	42	**13 / 13 / 0**
mathgen	791	100	**95 / 95 / 5**
colorsys	121	3	**3 / 3 / 0**
heapq	184	15	**9 / 8 / 3**
html	776	25	**10 / 9 / 4**

Table 3: The number of unfinished tasks and the median time for users to pinpoint hidden mistakes and finish tasks

Group	Short Program		Long Program		N_{unfin}
	T_{spot}	T_{total}	T_{spot}	T_{total}	
G_c	$452s$	$739s$	$1020s$	$1255s$	7
G_e	$60s$	$235s$	$286s$	$365s$	2

6.4 User Study

We conduct a preliminary user study on the usefulness of TransMap in the debugging *task* of pinpointing and fixing hidden mistakes. Evaluation metrics are the time users spend pinpointing and fixing bugs, and the number of unfinished tasks. We invite 24 computer science graduates[4], who have no prior familiarity with TransMap, and randomly divide them into a control group G_c (without TransMap) and an experimental group G_e (with TransMap). Two groups are given the same set of programs consisting of randomly sampled 8 short programs (< 20 lines) from our micro-benchmarks and 4 long code segments (> 150 lines) from `strsimpy` library. Each participant is randomly assigned 3 debugging *tasks*: 2 short programs and 1 long program, and we make sure every program is assigned to three users per group. Each program contains one mistake and only a one-line fix is needed to pass tests. If they spend more than 15 minutes, they have the option to abandon that program, which would be regarded as an unfinished task. The results are presented in Table 3. The N_{unfin} column shows the number of unfinished tasks out of 36 tasks per group. After removing these samples, the median of the time to confirm the line of mistake and the median of the total time to finish this task is presented as T_{spot} and T_{total}. We can see that the median time to pinpoint hidden mistakes reduces 87% on short code and 72% on long code when using TransMap compared to using the VS Code debugger. Moreover, the total time to pinpoint and fix mistakes also reduces by 68% on short code and 71% on long code. Additional results of the user study are presented in the supplementary material [2].

6.5 Discussion

Unfixable Cases When Creating Micro-benchmarks. We removed unfixable translations due to a known issue referred to as hallucination in large language models [24, 31]. Such unfixable translations typically have a heavy reliance on non-existent APIs, data types, and operators lacking JavaScript equivalents. Although

[4] All participants reported having at least some familiarity with Python or JavaScript.

Table 4: Accuracy of source mapping and breakdown of failures under different styles of the translated code

Scenario	Accuracy	Failure Breakdown		
		NEQ.	OOB.	DISO.
Fixed Translation	94.5%	2.1%	2.1%	1.4%
Buggy Translation	92.5%	3.8%	1.7%	2.1%
Added Comments	93.8%	1.4%	2.7%	2.1%
Renamed Variables	91.4%	2.1%	5.8%	0.7%
Multiple Functions	89.4%	4.8%	3.1%	2.7%

TransMap can identify these mistakes, fixing them is complicated, and we found it generally simpler to entirely rewrite the translation. TransMap cannot help much in such instances, examples of which are provided in the supplementary material [2].

The Types of Failures in Source Mapping. In our evaluation of the micro-benchmarks, 6.2% of the source mapping are counted as failures. There are three types of failures:

- **Code Not Equal (NEQ)**: 3.4%. Some code lines in the mapping output are different from the input (i.e., Part C and B in Fig. 4).
- **Out of Bound (OOB)**: 1.5%. Some annotated statement numbers in the translation (JavaScript) do not exist in the source (Python).
- **Disorder (DISO)**: 1.2%. Some statements are mapped wrongly.

We count the first two types as invalid and the third type as inaccurate. Most of the failure cases can be fixed with manual modifications to several lines of the mapping.

Robustness of Source Mapping. To assess how varying styles of the translated code affect the performance of source mapping, we conduct additional experiments[5] on the same LeetCode programs from our evaluation. We introduced changes to the fixed JavaScript translations in various ways: comment insertion, variable renaming, and function concatenation. Table 4 shows that these transformations degrade the accuracy of source mapping by about 1-5%. The supplementary material [2] has a more detailed discussion.

Possible Limitations of In-context Learning using LLMs. LLMs can typically generalize from a few examples [19, 60], yet the context window restricts the complexity of tasks that they contextualize. Further, their reasoning capability is token-limited, leading to a subpar performance on some few-shot reasoning tasks as indicated by Brown et al. However, methods like chain-of-thoughts [56] can be used to enhance outcomes with increased output length.

TransMap vs. Conventional Code Alignment Approaches. To our knowledge, most of the conventional approaches to code and trace alignment [41, 54, 61] focus on different versions of the code in the same language. We observe neural code translation can make unpredictable transformations, such as breaking, merging, reordering statements, transforming loop structure, and so on. Thus, an LLM-based self-explanatory approach is potentially easier to implement and more accurate for code translations by the LLM itself. However, the approach has some pitfalls, including noisy output and the limiting context length inherent in LLMs. Enhancements could potentially come from imposing output restrictions on language models [40] and expanding the context window [47].

[5]We used ChatGPT 3.5 [35] instead of Codex for this experiment and 4 case studies (except `strsimpy`) since Codex became publicly unavailable to us.

7 RELATED WORK

This is the first work on automatically pinpointing mistakes for neural code generator outputs when attempting to translate across programming languages. This problem is related to non-neural code translation and fault localization in different ways.

Code Translation. Traditional compilers and transpilation tools are the most common approaches used today. The readability of the generated code is often limited and the effort to write compilers is significant. To address these issues, a recent line of work uses natural language processing tools for code generation and translation. TransCoder [43] proposed unsupervised code translation using back-translation. Szafraniec et al. proposed to train neural decompilers to translate between languages that can be compiled to LLVM IR. Larger state-of-the-art models such as Codex [9] are also trained in an unsupervised manner and can translate between multiple languages. However, all of these approaches suffer from noisy outputs with mistakes. Our work on TransMap is motivated from these observations and aims to augment existing neural code translators. Another line of work with a longer history is to use symbolic approaches for code translation. TXL [11] is a domain-specific language for writing code translators. A more recent work introduces DuoGlot [52], a system that combines rule-based translation with rule synthesis from user-provided examples. This approach has the benefits of being predictable, but generally requires human effort and expertise to write rules. TransMap is an incomparable alternative: it directs the user to a fix in the translated program, rather than general transformation rules across languages.

Fault Localization. Fault location is a rich area of prior research which also pinpoints errors in buggy programs given tests [59]. The key difference from these works is that tests in our problem setup run on two programs in different languages. The novel challenge is therefore in mapping execution semantics between the programs especially after they have undergone a black-box neural translation. TransMap introduces generic capabilities, such as source maps, that can act as an important aid to adapt existing fault localization techniques to neural code generators in the future. Prior techniques have focused on different aspects: improving statistical scoring functions [3, 27], ranking mechanisms [5, 44], better reasoning under mutations [62], increase the quality of test suites [39, 48], improving trace analysis [12], symbolic reasoning for root cause analysis [8, 16, 28, 41], and more.

8 CONCLUSION

We provide the first systematic approach to pinpointing errors in code translated from one program language to another by modern neural translators. Our TransMap identifies mistakes with high fidelity in short JavaScript fragments translated from Python.

9 DATA AVAILABILITY

Our code and datasets are available on Zenodo [53]. The latest version and supplementary materials [2] can be found on Github.

ACKNOWLEDGMENTS

We thank the anonymous reviewers. This research is supported by grants given by the Ministry of Education in Singapore: Tier-2 grant MOE-T2EP20220-0014 and Tier-1 grant T1 251RES2023.

REFERENCES

[1] 2002. Chapter 15 - Microsoft Says JUMP—Java User Migration Path. In *C# For Java Programmers*, Brian Bagnall, Philip Chen, Stephen Goldberg, Jeremy Fairdoth, and Harold Cabrera (Eds.). https://doi.org/10.1016/B978-193183654-8/50019-0

[2] 2023. *Supplementary Material*. https://github.com/HALOCORE/TransMap

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[4] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[5] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 14, 18 pages.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Stephanie C. Y. Chan, Adam Santoro, Andrew K. Lampinen, Jane X. Wang, Aaditya Singh, Pierre H. Richemond, Jay McClelland, and Felix Hill. 2022. Data Distributional Properties Drive Emergent In-Context Learning in Transformers. http://arxiv.org/abs/2205.05055 arXiv:2205.05055 [cs].

[8] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/1985793.1985811

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] CodeGeeX. 2023. HumanEval-X: A new benchmark for Multilingual Program Synthesis. https://github.com/THUDM/CodeGeeX.

[11] James R Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210. https://doi.org/10.1016/j.scico.2006.04.002

[12] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 17–32.

[13] Joe Davison, Joshua Feldman, and Alexander Rush. 2019. Commonsense Knowledge Mining from Pretrained Models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 1173–1178. https://doi.org/10.18653/v1/D19-1109

[14] Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A Ortega. 2023. Neural Networks and the Chomsky Hierarchy. https://openreview.net/forum?id=WbxHAzkeQcn

[15] doocs. 2023. LeetCode solutions in any programming language. https://github.com/doocs/leetcode.

[16] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error invariants. In *FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings 18*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[17] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[18] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program Repair. *arXiv preprint arXiv:2211.12787* (2022). https://doi.org/10.48550/arXiv.2211.12787

[19] Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. What can transformers learn in-context? a case study of simple function classes. *arXiv preprint arXiv:2208.01066* (2022). https://doi.org/10.48550/arXiv.2208.01066

[20] OpenAI Github. 2023. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot/.

[21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[22] Anna Irrera. 2017. Banks scramble to fix old systems as IT 'cowboys' ride into sunset. https://www.reuters.com/article/us-usa-banks-cobol-idUSKBN17C0D8.

[23] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2021. Jigsaw: Large Language Models meet Program Synthesis. http://arxiv.org/abs/2112.02969 arXiv:2112.02969 [cs].

[24] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248 (mar 2023), 38 pages. https://doi.org/10.1145/3571730

[25] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. 2020. How Can We Know What Language Models Know? *Transactions of the Association for Computational Linguistics* 8 (07 2020), 423–438. https://doi.org/10.1162/tacl_a_00324 arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00324/1923867/tacl_a_00324.pdf

[26] Wei Jin and Alessandro Orso. 2013. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/2483760.2483763

[27] James Jones, Mary Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. *Proceedings - International Conference on Software Engineering*, 467– 477. https://doi.org/10.1145/581339.581397

[28] Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. *SIGPLAN Not.* 46, 6 (jun 2011), 437–446. https://doi.org/10.1145/1993316.1993550

[29] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[30] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Phrase-Based & Neural Unsupervised Machine Translation. , 5039–5049 pages. https://doi.org/10.18653/v1/D18-1549

[31] Junyi Li, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2023. HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models. *arXiv e-prints* (2023), arXiv–2305. https://doi.org/10.48550/arXiv.2305.11747

[32] Microsoft. 2023. Visual Studio Code. https://code.visualstudio.com/.

[33] mozilla. 2023. source-map: Consume and generate source maps. https://github.com/mozilla/source-map

[34] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[35] OpenAI. 2023. ChatGPT. https://openai.com/blog/chatgpt.

[36] OpenAI. 2023. Code completion - OpenAI API. https://platform.openai.com/docs/guides/code/best-practices.

[37] OpenAI. 2023. OpenAI API. https://openai.com/api/.

[38] Stack Overflow. 2023. Does JavaScript support array/list comprehensions like Python? https://stackoverflow.com/questions/31353213/does-javascript-support-array-list-comprehensions-like-python.

[39] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[40] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. http://arxiv.org/abs/2201.11227 arXiv:2201.11227 [cs].

[41] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. DARWIN: An Approach to Debugging Evolving Programs. *ACM Trans. Softw. Eng. Methodol.* 21, 3, Article 19 (jul 2012), 29 pages. https://doi.org/10.1145/2211616.2211622

[42] 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, 1, Article 140 (jan 2020), 67 pages.

[43] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 20601–20611. https://proceedings.neurips.cc/paper_files/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf

[44] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 537–549. https://doi.org/10.1145/3433210.3437528

[45] Richard Shin and Benjamin Van Durme. 2022. Few-Shot Semantic Parsing with Language Models Trained on Code. In *Proceedings of the 2022 Conference of the*

North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, Seattle, United States, 5417–5425. https://doi.org/10.18653/v1/2022.naacl-main.396

[46] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code Translation with Compiler Representations. (July 2022). http://arxiv.org/abs/2207.03578 arXiv:2207.03578 [cs].

[47] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2022. Efficient Transformers: A Survey. 55, 6, Article 109 (dec 2022), 28 pages. https://doi.org/10.1145/3530811

[48] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. NET. In *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2*. Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[49] Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2022. StructCoder: Structure-Aware Transformer for Code Generation. *arXiv preprint arXiv:2206.05239* (2022). https://doi.org/10.48550/arXiv.2206.05239

[50] tree sitter. 2023. Tree-sitter | Introduction. https://tree-sitter.github.io/tree-sitter/.

[51] typescriptlang. 2023. TSConfig Reference: Source Map. https://www.typescriptlang.org/tsconfig#sourceMap

[52] Bo Wang, Aashish Kolluri, Ivica Nikolić, Teodora Baluta, and Prateek Saxena. 2023. User-Customizable Transpilation of Scripting Languages. 7, OOPSLA1, Article 82 (apr 2023), 29 pages. https://doi.org/10.1145/3586034

[53] Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. 2023. *TransMap: Pinpointing Mistakes in Neural Code Translation (Artifact)*. https://doi.org/10.5281/zenodo.8283633

[54] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jinsong Dong, Qinghua Zheng, and Ting Liu. 2021. Explaining Regressions via Alignment Slicing and Mending. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2421–2437. https://doi.org/10.1109/TSE.2019.2949568

[55] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022). https://doi.org/10.48550/arXiv.2201.11903

[57] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[58] Sean Welleck, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho. 2020. Consistency of a Recurrent Language Model With Respect to Incomplete Decoding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 5553–5568. https://doi.org/10.18653/v1/2020.emnlp-main.448

[59] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[60] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2022. An Explanation of In-context Learning as Implicit Bayesian Inference. http://arxiv.org/abs/2111.02080 arXiv:2111.02080 [cs].

[61] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 238–248. https://doi.org/10.1145/1375581.1375611

[62] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning Dynamic Slices with Confidence. *SIGPLAN Not.* 41, 6 (jun 2006), 169–180. https://doi.org/10.1145/1133255.1134002

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