CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models

Caroline Lemieux
University of British Columbia, Canada†
clemieux@cs.ubc.ca

Jeevana Priya Inala
Microsoft Research, USA
jinala@microsoft.com

Shuvendu K. Lahiri
Microsoft Research, USA
shuvendu.lahiri@microsoft.com

Siddhartha Sen
Microsoft Research, USA
sidsen@microsoft.com

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00085

***Abstract*—Search-based software testing (SBST) generates high-coverage test cases for programs under test with a combination of test case generation and mutation. SBST's performance relies on there being a reasonable probability of generating test cases that exercise the core logic of the program under test. Given such test cases, SBST can then explore the space around them to exercise various parts of the program. This paper explores whether Large Language Models (LLMs) of code, such as OpenAI's Codex, can be used to help SBST's exploration. Our proposed algorithm, CODAMOSA, conducts SBST until its coverage improvements stall, then asks Codex to provide example test cases for under-covered functions. These examples help SBST redirect its search to more useful areas of the search space. On an evaluation over 486 benchmarks, CODAMOSA achieves statistically significantly higher coverage on many more benchmarks (173 and 279) than it reduces coverage on (10 and 4), compared to SBST and LLM-only baselines.**

I. INTRODUCTION

The goal of *automated test case generation* is to generate test cases covering all the different behaviors of the program under test. Test cases that cover behaviors a developer did not think about may help the developer find bugs or otherwise improve the quality of their program.

One approach to automatically generate test cases is *search-based software testing* (SBST) [1]–[5]. Most state-of-the-art SBST tools build on some form of *evolutionary algorithm*. At a high-level, these algorithms start by creating a set of random test cases, repeatedly mutating these test cases, and keeping those with higher *fitness* for further mutation. Often, higher fitness relates to higher coverage of the program under test.

SBST works well when mutating a test case has a non-negligible likelihood of increasing fitness. This is not the case for the Python module in Fig. 1. The main function under test, `bump_version`, increments program version strings, e.g. bumping `'1.2.2'` to `'1.2.3'`. As shown in Fig. 1, an SBST tool can generate test cases where `bump_version` is called on random strings like `'a\!sUo~AU'`. From here, SBST has difficulty making further testing progress because the probability of mutating `'a\!sUo~AU'` to a reasonable version string—necessary to exercise any more of the code under test—is nearly zero. Unfortunately, studies have found that fitness plateaus like these—where mutated test cases are unlikely to increase coverage—are quite common [6], [7].

† Most implementation/evaluation work conducted at Microsoft Research.

One core problem leading to these plateaus is that SBST has difficulty exercising the program under test (PUT) in an *expected manner* [7]. For instance, in our running example, `bump_version` *expects* its first argument to be a string of a particular format. Or, a program might *expect* a particular sequence of function calls to correctly construct an object [8]. We hypothesize that if we provided SBST with test cases exercising the PUT in an expected manner, then SBST's search could escape these plateaus. The difficulty is in *how* to produce these expected test cases.

Recently, large language models (LLMs) applied to code—such as Codex [9]—have shown impressive results in producing natural-looking code. In this paper, we investigate whether LLMs can be selectively invoked to produce test cases that can help SBST escape its coverage stalls.

Our proposed technique, CODAMOSA, starts SBST and monitors its coverage progress. When CODAMOSA notices a stall in coverage, it identifies callables in the PUT that have low coverage. Then, it queries Codex to generate tests for these callables. To build upon these generations, CODAMOSA deserializes the raw character sequences generated by Codex into SBST's internal test case representation. This allows it to leverage SBST's mutation operations and fitness function to mutate and evaluate the quality of the Codex-generated tests.

In building this workflow, there are some subtleties in *when* and *how* to ask Codex for a test case. Deserializing test cases presents a core technical challenge: to make the search tractable, SBST restricts the set of statements that can be present in a test case. Codex, on the other hand, generates arbitrary Python code. Our deserialization procedure enables tractable expansion of SBST's search space: calls to new functions or new syntactical constructs suggested by Codex are integrated into test cases if they help increase coverage.

We build CODAMOSA for Python on top of the Pynguin [10] test suite generation framework. We conduct a large-scale evaluation of CODAMOSA on 486 Python modules. We find that CODAMOSA achieves statistically significantly higher coverage on many more benchmarks (173 and 279) than it reduces coverage (10 and 4) on compared to our SBST and LLM-only baselines. The average magnitude of the coverage increase is also higher (10% and 9%) than the average magnitude of decreases (-4% and -3%). In case studies, we find that Codex's ability to generate special string primitives, call new functions, and introduce syntactical

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00085

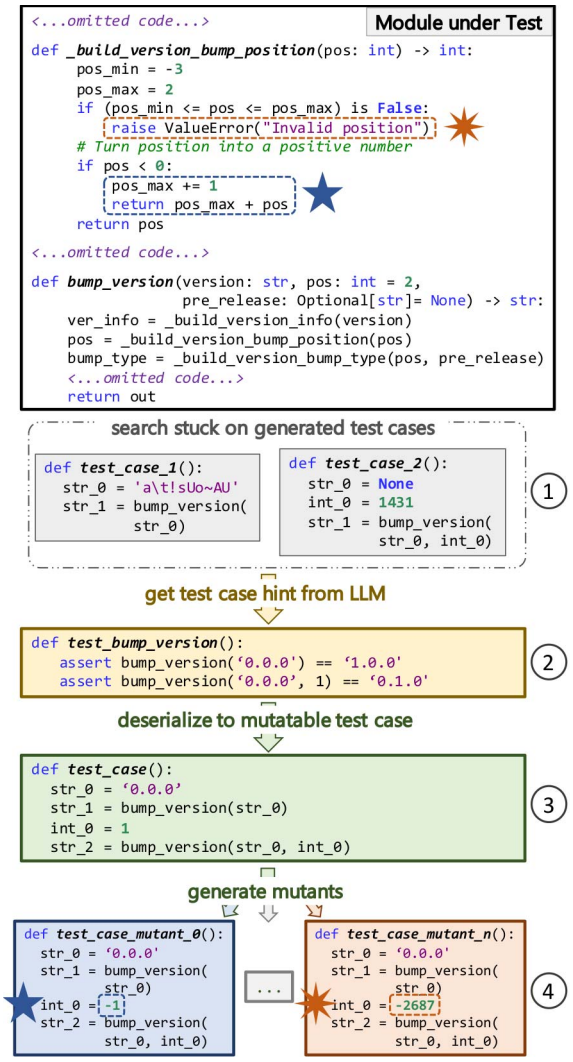


Fig. 1: Example run of CODAMOSA on a module under test. When search-based testing hits a coverage stall (1), CODAMOSA queries Codex for a test case (2), and deserializes this test (3) into a form that enables the search to progress (4).

constructs contribute the most to CODAMOSA's improvements over SBST. In summary, our contributions are as follows:

- We propose CODAMOSA, which integrates LLMs of code with SBST. It includes techniques to integrate arbitrary Python test cases into SBST, regardless of their source.
- We conduct a large-scale evaluation of CODAMOSA, its design decisions, and baselines, across 486 benchmarks.
- We release our open-source implementation of CODAMOSA, and data to help replicate the experiments in this paper: https://github.com/microsoft/codamosa.

II. MOTIVATION

Consider the code fragment at the top of Fig. 1. It is based off of the `packages` module of the `flutils` project [11]. The core function provided by this module is `bump_version`, which increases the version in a version string, e.g. from `'1.2.2'` to `'1.2.3'`. The `pos` argument specifies the position at which to increase the version, and the `pre_release` argument creates alpha or beta versions: `bump_version('1.2.2', 1, 'a')` yields `'1.3a0'`. Our code fragment mostly highlights a helper function, `_build_version_bump_position`, which handles the integer value passed in the `pos` argument.

Suppose we want to automatically create a test suite for this module. Most state-of-the-art tools for test suite generation are *search-based*. These tools use meta-heuristic optimization algorithms [12]–[14] to generate high-coverage test suites for the module under test. State-of-the-art algorithms such as MOSA [3] and MIO [5] are at their core *evolutionary algorithms*. They produce test cases by (a) starting with a set of randomly-generated test cases and (b) repeatedly mutating these test cases, and keeping only the fittest test cases, as defined by some—often code-coverage-related—fitness metric. Section III-A provides more background on these algorithms.

Suppose we run the SBST algorithm MOSA on the module under test in Fig. 1. Initially, MOSA will achieve some coverage of this module. But soon, its coverage improvements cease: even after repeated mutation iterations, none of the mutated test cases it generates cover new code in the module. We say that the algorithm has reached a *coverage stall*.

The reason for this stall is clear to any human analyst looking at the test cases generated by MOSA. Fig. 1 shows two such test cases, labeled (1). These test cases pass unexpected values to the `version` argument of `bump_version`: `'a\t!sUo~AU'` and `None`. These values cause an exception to be raised immediately upon invoking `bump_version`, and so little code in the `packages` module is exercised.

Our proposed technique, CODAMOSA, overcomes such stalls as follows. CODAMOSA starts by running MOSA. Then, when it reaches a coverage stall, it asks for a "hint". Precisely, it asks for a test case targeting an under-covered function in the module under test. For instance, it may get the test case labeled (2) in Fig. 1. Although the right-hand-side of this test case's first assert is not correct, it calls `bump_version` on a valid version string. CODAMOSA extracts the core functionality of the test case into a mutatable test case format: (3) in Fig. 1. It observes that this test case increases coverage of the module under test, and adds it to its test case set.

However, this test case consists of normal invocations of `bump_version`, that do not exercise the subtlety of, say, the `pos` argument. The test case neither exercises the code for handling an out-of-bounds position (highlighted with a ✶ in Fig. 1), nor the code for handling negative positions (highlighted with a ★ in Fig. 1).

Luckily, exploring different integer values for an argument is one of the strengths of mutation-based algorithms like MOSA, on which CODAMOSA is built. Thus, CODAMOSA will generate many mutants from the deserialized test case, labelled (4) in Fig. 1. Some, like `test_case_mutant_0`, may have valid negative values for `pos`, and exercise the code labeled ★. Others, like `test_case_mutant_n`, may have out-of-bounds values for `pos`, exercising the code labeled ✶.

Algorithm 1 Abstract SBST algorithm with archiving for test generation (*SBST-A*).

Input: *module* to test, population size *N*, search time T
Output: a set of test cases exercising *module*

```
1: covPts ←GETCOVERAGEPOINTS(module)
2: callables ← GETCALLABLES(module)
3: testCases ← RANDOMTESTCASES(callables, N)
4: archive ← SMALLESTCOVER(∅ ∪ testCases, covPts)
5: while timeElapsed < T do
6:   newTests ← MUTATE(testCases, callables, N)
7:   archive←SMALLESTCOVER(archive∪newTests,covPts)
8:   testCases ← FITTEST(newTests ∪ testCases, covPts)
9: return archive
```

The *key takeaway* from this example is the following. Providing the search with an example of *expected* use of the module under test allows SBST to *escape its coverage stall*.

The problem, of course, is how to get such examples of expected uses of module code. In order to do this, we need a system that understands what code *"should look like"*. Conveniently, large language models for code such as Codex [9] promise to do just this. Thus, CODAMOSA uses Codex to generate test cases hints like (2) in Fig. 1.

The example in Fig. 1 is taken from a real example in our evaluation on which CODAMOSA greatly outperforms our baselines. Over 16 runs, MOSA achieved 18.4% average total coverage of this module. CODAMOSA, in contrast, achieved 91.8% average total coverage, an absolute percentage-point increase of 73.4% over MOSA. CODAMOSA also outperforms a Codex-only baseline, which achieves only 64.5% average total coverage on this module.

Section IV describes CODAMOSA in more detail. We first provide some background on SBST and LLMs.

III. BACKGROUND

A. Search-Based Software Testing

The goal of test-suite generation is to generate a *test suite*—that is, a set of test cases—that covers the diverse behaviors of the program under test. A *test case* consists of a sequence of statements, including call statements to the callables (functions, methods, constructors) in the module under test. *Search-based software testing* (SBST) uses meta-heuristic optimization [12]–[14]—such as evolutionary algorithms or simulated annealing—to generate test suites.

A large class of SBST tools use evolutionary or genetic algorithms [1]–[5] in order to generate test suites. At their core, these tools work by (a) randomly generating a set of test cases to start from; and (b) repeatedly mutating the test cases to increase coverage of the program under test. In this paper, we build on SBST algorithms with an archive, such as MOSA [3]. Algorithm 1 outlines the high-level view of such algorithms: we refer to this generalized algorithm as *SBST-A*.

First, *SBST-A* collects a set of coverage points (lines and branches) in the module under test (Line 1). Then, it collects the callables—functions, constructors, and generators—from the module under test (Line 2). These callables are split into two groups. First, *test objects*: the public callables in the module. Second, *dependent callables*, which are used to construct or modify arguments to the test objects. Implementations may collect these dependent callables by following type signatures (in Java [15]) or available type hints (in Python [10]).

SBST-A then initializes the population, *testCases*, by generating N random test cases (Line 3). Each test case consists of a sequence of calls to test objects, and any primitive statements or calls to dependent callables necessary to generate arguments for the test object calls. In addition to *testCases*, the population mutated during the main search, *SBST-A* keeps track of an *archive* (Line 4). The separation of these two sets allows *SBST-A* to keep track of the test cases that most succinctly cover the module under test (*archive*), but also keep those that are found to be promising for future coverage (*testCases*). The function SMALLESTCOVER (Line 4) returns the smallest test cases from its first argument that together cover all coverage points in *covPts*.

While time remains in the search, *SBST-A* mutates its test case population (Line 6). If any mutated test case covers a new coverage point, or does so with fewer statements than before, *SBST-A* updates the archive (Line 7). Then, it updates the population with the fittest of the original population and the mutated test cases (Line 8). Finally, when the time budget expires, *SBST-A* returns the test suite *archive*.

B. Large Language Models of Code

Large language models (LLMs) have achieved impressive performance on natural language tasks, including text generation [16]–[18], conversation [19], and reasoning [20]. They are trained in an *auto-regressive* manner, i.e. trained to predict the next token given some prefix of text. This allows LLMs to be trained on vast quantities of text available on the web, with no need to label data. For instance, GPT-3 [16] was trained on almost 700 GB of data collected from CommonCrawl [21].

Following the success of LLMs at natural language tasks, there has been tremendous interest in applying LLMs to code. This led to the development of several code generation models such as Codex [9], AlphaCode [22], Google's program synthesis model [23], PolyCoder [24], InCoder [25] and CodeGen [26]. Codex is the largest state-of-the-art code generation model publicly available for querying through an API. It is built on top of GPT-3, with additional training on code from 55 million GitHub repositories, totaling 160GB of data. Codex is most proficient in Python, but it supports several other languages including Go, JavaScript, and TypeScript.

We control LLM generations via *prompts*. A *prompt* is the input text that is passed to the LLM at inference time. The LLM generates text following the prompt until it generates a predetermined stop word or exceeds its maximum word limit. By carefully crafting prompts, i.e. *prompt engineering*, researchers have applied Codex to a variety of new tasks—solving coding competition and interview questions [9], [22], code completion [27], code explanation [28], and code re-

Algorithm 2 CODAMOSA. Parts of the algorithm that are the same as Algorithm 1 are greyed out.

Input: *module* to test, population size *N*, search time T, maximum stall length *maxStallLen*, *model* to query for test cases, and number of times to query per iteration M.
Output: a set of test cases that maximizes the coverage of *module*

```
1:  covPts ←GETCOVERAGEPOINTS(module)
2:  callables ← GETCALLABLES(module)
3:  testCases ← RANDOMTESTCASES(callables, N)
4:  archive ← SMALLESTCOVER(∅ ∪ testCases, covPts)
5:  stallLen ← 0
6:  while timeElapsed < T do
7:    coverageBefore ←COVERAGE(archive)
8:    testCasesBefore ← testCases
9:    if stallLen > maxStallLen then
10:     wasTargeted ← True
11:     newTests, fns ← TARGETEDGEN(archive, callables)
12:     callables ← callables ∪ fns      ▷ expand the callables
13:     mutatedTests ← MUTATE(newTests, callables, N)
14:     newTests ← newTests ∪ mutatedTests
15:     stallLen ← 0
16:   else
17:     wasTargeted ← False
18:     newTests ← MUTATE(testCases, callables, N)
19:   archive←SMALLESTCOVER(archive∪newTests,covPts)
20:   testCases ← FITTEST(newTests ∪ testCases, covPts)
21:   if wasTargeted and testCases = testCasesBefore then
22:     maxStallLen ← 2 ∗ maxStallLen                ▷ back off
23:   if COVERAGE(archive) = coverageBefore then
24:     stallLen ← stallLen + 1
25:   else
26:     stallLen ← 0
27: return archive
```

pair [29], [30]—*without* having to retrain Codex. In this paper, we use Codex to generate test cases via prompt engineering.

IV. TECHNIQUE

Recall the motivation from Section II. SBST has a tendency to periodically reach coverage stalls; these stalls may be overcome if we provide SBST with an example of an expected test case. In CODAMOSA, we investigate whether LLMs can effectively provide such examples.

A. High-Level Walkthrough

Algorithm 2 shows CODAMOSA's high-level algorithm. We have greyed out the statements identical to those in Algorithm 1. In addition to *module*, population size N, and search time T, CODAMOSA expects a maximum stall length *maxStallLen*, LLM to query *model*, and number of model queries per iteration *M*. CODAMOSA starts by initializing the set of coverage points, the callables, the population, and the archive as in Algorithm 1 (Lines 1-4). It also initializes the coverage stall length to 0 (Line 5)

Algorithm 3 The TARGETEDGEN function.

Input: a set of test cases *archive*, a set of functions that can be called *callables*. From Alg. 2: *module* to test, *model* to query, and number of times to query per iteration M.
Output: a set of test cases generated by *model* to exercise low-coverage test objects in *callables*, and any new function objects that were called in these test cases.

```
1:  function TARGETEDGEN(archive, callables)
2:    testFns ← OBJECTSUNDERTEST(callables)
3:    covPerFn ← COMPUTECOVERAGE(archive, testFns)
4:    newFns ← ∅
5:    newTests ← ∅
6:    for 1 ≤ i ≤ M do
7:      targetFn ← SAMPLELOWERCOV(covPerFn, testFns)
8:      prompt ← GENERATEPROMPT(targetFn, module)
9:      modelOutput ← QUERY(model, prompt)
10:     testCase, fns ← PARSETOTESTCASE(modelOutput)
11:     newFns ← newFns ∪ fns
12:     newTests ← newTests ∪ {testCase}
13:   return newTests, newFns
```

At the start of each mutation iteration, CODAMOSA keeps track of the coverage achieved by the current archive (Line 7) and keeps a copy of the current population (Line 8). If the coverage stall length does not surpass *maxStallLen*, CODAMOSA behaves just like *SBST-A*: creating a set of mutated test cases, and updating the archive and population with these mutants (Lines 18-20). If the coverage increases, the stall length is set to zero (Line 26). If it does not increase, the coverage stall length is incremented by one (Line 24).

CODAMOSA behaves differently when it notices the coverage stall has reached the maximum length (Line 9). In this iteration, rather than simply mutating the existing population, CODAMOSA calls TARGETEDGEN (Sec. IV-B), which invokes *model* to generate test cases targeting low-coverage test objects (Line 11). The Codex-generated tests may cause CODAMOSA to increase its search space (Line 12); Section IV-C discusses this in detail. CODAMOSA applies one round of mutation on these generated test cases (Line 14). The union of the generated test cases and their mutants is then used to update the archive and population, as in a regular *SBST-A* iteration (Line 19-20). Finally, since Codex queries are time-consuming, if the targeted generation step does not yield an update in the population, CODAMOSA "backs off", doubling the maximum stall length (Line 22).

B. Targeted Generation

Algorithm 3 describes the TARGETEDGEN function. The goal of this function is to generate M tests invoking callables in *callables* that are not well-covered by the tests in *archive*.

First, TARGETEDGEN identifies the *test objects* in *callables* and stores these in *testFns* (Line 2). As described in Section III-A, test objects are all the public callables in *module*; these form the backbone of test cases. The other callables are *dependent callables*, necessary to generate arguments for

the test objects. Then, TARGETEDGEN computes the coverage achieved by *archive* for each of the test objects in *testFns* (Line 3). Each callable in *testFns* gets a score from 0 to 1, where 1 means fully covered.

After this, TARGETEDGEN moves on to querying the model. It samples a callable to target with probability inversely proportional to the coverage of each callable (Line 7). That is, if c^* is a callable with coverage score $cov(c^*)$, the probability of sampling c^* is $\frac{1-cov(c^*)}{\sum_{c \in testFns} 1-cov(c)}$. Then, we generate a prompt to target this particular callable (Line 8), query *model* with this prompt (Line 9), and deserialize *model*'s output into a test case (Line 10). Finally, TARGETEDGEN returns the deserialized generated test cases (Line 13).

1) Prompt Generation: Let X be the callable being targeted. The first part of our prompt is the source code of the module under test. If the source code is longer than the LLM's maximum prompt length, we take as much of the source code as possible, including the definition of the callable X. Then, we add to the prompt a function header of the appropriate form depending on whether X is a function (`# Unit test for function X\ndef test_X():`), method (`# Unit test for method X of class C\ndef test_C_X():`), or constructor (`# Unit test for constructor of class X\ndef test_X():`).

C. Deserializing Generated Test Cases

In Line 10 of Algorithm 3, CODAMOSA deserializes a Codex-generated test case into the search algorithm's internal representation. This representation simplifies specialized operations such as type-aware mutations or test case reduction.

CODAMOSA is built on the Pynguin framework for Python test suite generation [10]. Pynguin's implementation includes a test case deserialization procedure, which we refer to as *Pynguin-deserialize*. Its purpose is to deserialize test cases generated by prior Pynguin runs, in order for a new run to start from these test cases rather than random ones. Thus, it cannot deserialize arbitrary Python code. To take maximal advantage of Codex-generated code, we build an augmented deserialization procedure: PARSETOTESTCASE.

1) Rewriting Codex Generations: Pynguin assumes test cases are a sequence of assignment statements. The left-hand-side of the assignment must be a single variable name and the right-hand-side must be: (1) a primitive constant; (2) a list or dictionary where all elements are variables names, or (3) a call where all arguments are variables names.

Thus, PARSETOTESTCASE first transforms the arbitrary Python code generated by Codex into a series of assignment statements. In particular, (1) we store values of standalone expression statement in a variable, and (2) we remove *nested subexpressions* as allowed by variable scoping rules, so that all right-hand-side subexpressions are variable references. For instance, `z = foo(bar(2))` becomes 3 statements: `int_0 = 2`, `var_0 = bar(int_0)` and `z = foo(var_0)`.

2) Partial Parsing: Pynguin-deserialize discards any test cases where one statement could not be parsed. For instance, suppose we give *Pynguin-deserialize* the following test case:

```
1. x = 3
2. y = UNKNOWN_FUNCTION(x)
3. z = foo(y)
4. w = bar(x)
```

When it fails to parse Line 2, *Pynguin-deserialize* discards the entire testcase. In contrast, our approach visits every assignment statement and tries to parse it. It discards unparseable statements (e.g., Line 2) and those that depend on them (e.g., Line 3), but turns the parseable ones into a testcase, e.g:

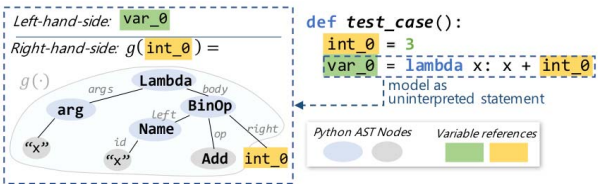
```
x = 3
w = bar(x)
```

Additionally, PARSETOTESTCASE visits assignment statements within code blocks of complex statements (e.g. `if`, `with`, `class`), but discards the complex statements themselves. Thus, the deserialized test cases consist of only straight-line code.

3) Callables Expansion: Pynguin-deserialize only parses call statements that invoke a callable in *callables* (Line 2 in Algorithm 1). Recall from Section III-A that *callables* includes only test objects and dependent callables. In contrast, CODAMOSA keeps track of all callables that are reachable via import statements in the module under test, in a *backup callables* set. Then, if Codex outputs a call that cannot be resolved, CODAMOSA checks if the callable is in its backup set. If it is, its deserializer parses the call and promotes the callable to *callables* (Line 12 in Algorithm 2). These promoted callables can later be used in mutations.

Concretely, this means that if Codex notices a commonly imported module, (e.g., `import ast`), and calls a real function from this module (e.g. `ast.parse(str_0)`), this call will be parsed and taken up into the test case. However, if Codex invents a new function call not defined in an imported module, (e.g. `ast.foo(str_0)`), the deserializer discards the statement. Expanding the space of callables only if they are suggested by Codex prevents the huge explosion of the search space that would come with considering *all* the backup callables.

4) Uninterpreted Statements: Recall that *Pynguin-deserialize* supports a limited number of expressions on the right-hand-side of assignment statements. In initial experiments, we noticed that sometimes right-hand-sides with different syntactical constructs, e.g. `var_0 = lambda x : x + int_0`, were crucial to increasing coverage. We added a new type of statement to Pynguin's test case model, *uninterpreted statements*, to integrate these new syntactical constructs into test cases. These statements take the form $lhs = g(vr_0, \ldots, vr_n)$, where some g operates over variable references $vr_0, \ldots, vr_n$. We model g as a modified Python AST, which keeps track of any use of a variable previously defined in the test case. For instance:



Tracking variable references $vr_0, \ldots, vr_n$ allows us to use Pynguin's test case pruning and analysis operations out-of-the-box. We implement a single mutation operator for these statements: replace an existing variable reference with another one. Unlike the mutation operators for other Pynguin statements, this mutation is unaware of g's semantics or type constraints—i.e., g is *uninterpreted*. Thus, if a test case defines variables `int_0` and `str_0`, CODAMOSA may mutate `a=b[int_0]` $\rightarrow$ `a=b[str_0]`. At test case execution time, g is turned back into Python code, and the new syntax will be executed.

This novel representation allow us to incorporate a variety of syntactical constructs into SBST without having to write custom generators and mutators for each construct, which is error-prone. For instance, while building CODAMOSA, we noticed that *Pynguin-deserialize* did not handle calls to builtins such as `list` correctly [31]: it deserialized these to list *statements*, which have different semantics (`list("ab")!=["ab"]`). Fixing this is non-trivial, as builtin functions are not in the set *callables*. Thus, they cannot be modeled with Pynguin's existing call statements. In CODAMOSA, we easily fixed this issue by modeling builtin calls as uninterpreted statements.

V. EVALUATION

Our evaluation investigates the following questions.

RQ1. How does CODAMOSA compare to our baselines on our benchmark set? (Section V-B)

RQ2. How do our design decisions (uninterpreted statements, Codex hyper-parameters, low-coverage targeting, prompting) affect test effectiveness? (Section V-C)

RQ3. Why, qualitatively, does CODAMOSA achieve different coverage results than MOSA? (Section V-D)

RQ4. Are Codex generations copied from out-of-prompt files in the module under test's codebase? (Section V-E)

A. Experimental Setup

The version of CODAMOSA used in all our experiments was built on top of Pynguin 0.19.0. We built CODAMOSA specifically on top of the MOSA implementation in Pynguin, though it can be easily ported to algorithms that share the structure of Algorithm 1. We add 3.7k lines and modify around 700 lines of main functionality code for our implementation. We also add 1.8k lines of tests of our implementation.

1) Baselines: Our first baseline is Pynguin's implementation of MOSA. The other obvious baseline would have been DynaMOSA [3]: unlike MOSA, it prioritizes only those coverage objectives that are not dominated by existing unmet objectives. For instance, if a function entry is not yet covered, the branch statements in that function are not prioritized as coverage objectives: first, the function entry must be covered. DynaMOSA was found to be more effective than MOSA in the original work [3]. We use MOSA rather than DynaMOSA because in Pynguin 0.19.0, only MOSA supported both line and branch coverage. We want both measures because Python code, when called with invalid arguments, throws many exceptions. In the presence of exceptions, branch coverage does not supersede line coverage.

Our second baseline, CODEXONLY, uses only Codex to generate test cases. CODEXONLY spends its entire search budget calling TARGETEDGEN for randomly selected test objects, and calling PARSETOTESTCASE on the Codex-generated tests to get Pynguin-format test cases. This enables us to see whether the search in CODAMOSA confers any advantage over using Codex alone. Note that since the test cases are deserialized back into the Pynguin format, this baseline may have lower coverage than running the raw Codex output (see "Unparseable Constructs" in Section V-D2).

2) Benchmark Collection: We identified 35 projects from which to source modules, from the evaluation of Pynguin [32] and BugsInPy [33]. We use `pipreqs` [34] to automatically identify each project's dependencies, and Python's builtin `setuptools.find_packages` utility to automatically identify modules in each project. For each of these modules we ran a preliminary test: two runs of MOSA for one minute each. We removed modules that failed to produce results, and those on which 100% coverage was achieved within a minute. We also down-sampled the number of modules that shared the same parent module. This left us with 486 benchmark modules over 27 projects; Table I provides the details.

3) Algorithm Parameters: During implementation, after observing runs of CODAMOSA on a small test benchmark, we chose a maximum stall length of 25 iterations, and a maximum number of queries to Codex of 10. We used these values for evaluation to prevent overfitting to our benchmarks. For population size, we use the Pynguin default of 50.

4) Experimental Parameters: We ran each technique on each test module 16 times, for 10 minutes each time ($T = 10$ minutes). We chose 10 minutes as a search time as it is the time used in the evaluation of Pynguin [32], and is longer than the 5-8 minute search times used in the evaluation of MOSA [3]. For CODAMOSA and its variants, the 10-minute time includes the time to query Codex via OpenAI's API.

B. Comparison to Baselines

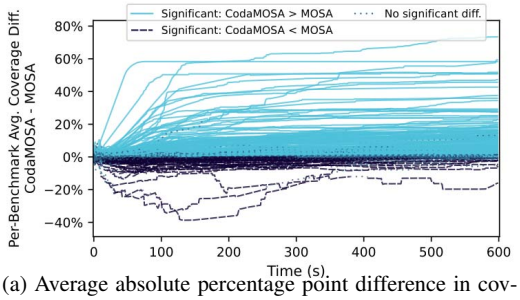
We ran MOSA, CODEXONLY, and CODAMOSA on our 486 benchmarks. Following Arcuri and Briand's [35] guidelines, we use a Mann-Whitney U-Test to compare the significance of coverage differences between techniques, at $p = 0.05$.

Fig. 2 shows the average coverage *difference* through time between CODAMOSA and the baselines. Each line represents a separate benchmark module. The differences are the *absolute percentage point differences* in line + branch coverage. For instance, if CODAMOSA has 80% coverage and the baseline has 20% coverage, this is 60% on the y-axis. We highlight the average difference when CODAMOSA's coverage at that time was significantly higher or lower than the baseline's.

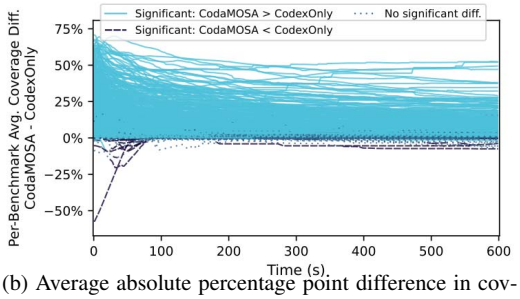
Fig. 2a shows the average difference in coverage through time compared to MOSA. Since CODAMOSA waits until a coverage stall to start invoking Codex, we do not see an immediate bump in coverage. Contrast this with Fig. 2b, where CODAMOSA starts with higher coverage than CODEXONLY. This reflects the fact that it is initially faster to generate test cases using random test generation than by querying Codex.

TABLE I: Characteristics of benchmark modules used in the evaluation. Src is the project source: "Pyn" for Pynguin, "BIP" for BugsInPy. **# Mod.** is the number of benchmark modules from the project. Min, mean, and max of **Size (LoC)**, **Test Objects** is over benchmark modules in the project.

Project	Revision	Src	# Mod.	Size (LoC)			Test Objects		
				min	*mean*	*max*	*min*	*mean*	*max*
apimd	f32841b	Pyn	2	111	291	471	3	16	29
codetiming	a7ad85a	Pyn	1	45	45	45	12	12	12
dataclasses-json	3dc59e0	Pyn	4	55	194	286	3	9	17
docstring_parser	a5dc2cd	Pyn	5	14	120	198	1	6	15
flutes	49647e4	Pyn	3	13	99	216	1	9	20
flutils	df0f84e	Pyn	9	21	127	263	1	4	9
httpie	bb36897	Pyn	19	9	129	393	2	7	16
isort	a6222a8	Pyn	2	109	110	111	13	14	15
mimesis	310092c	Pyn	18	25	75	188	1	10	30
py-backwards	8be3c44	Pyn	19	8	50	189	1	3	16
pyMonet	f132cfa	Pyn	10	23	51	75	3	13	27
pypara	7d705a5	Pyn	6	7	195	743	1	41	183
semantic-release	3689157	Pyn	6	21	84	254	1	8	28
string-utils	d903db3	Pyn	3	55	157	230	4	16	27
pytutils	9813bb3	Pyn	12	4	56	243	1	5	14
sanic	93a0246	Pyn	11	19	139	505	2	8	21
sty	f99e918	Pyn	2	19	56	94	2	8	13
thonny	fb389f4	Pyn	3	24	217	521	3	11	24
typesystem	6a9590c	Pyn	10	32	197	603	1	14	37
black	2354126	Both	6	52	271	555	3	19	53
ansible	f00f123	BIP	237	8	132	1110	1	7	73
cookiecutter	1c0b5b1	BIP	5	18	60	111	1	4	9
PySnooper	31bfc63	BIP	4	68	144	336	2	9	14
thef**k	0949d2e	BIP	32	6	36	156	1	3	18
tornado	2047e7a	BIP	13	100	288	600	4	19	33
tqdm	18d7aa4	BIP	9	16	75	202	1	5	11
youtube-dl	b224cf3	BIP	35	24	203	873	1	5	32



(a) Average absolute percentage point difference in coverage between CODAMOSA and MOSA.



(b) Average absolute percentage point difference in coverage between CODAMOSA and CodexOnly.

Fig. 2: Per-benchmark difference in average coverage through time. Times where coverage is significantly higher for one technique are highlighted as indicated in the legend.

We see that the increases compared to MOSA get larger over time (repeated injections of Codex test cases give coverage boosts), while those over CODEXONLY decrease over time (mutating test cases gives a fixed boost over Codex test cases).

Figs. 3a and 3b summarize the end-of-search-time average coverage achieved by CODAMOSA (on the y-axis) and the baselines (on the x-axis). The circles along the $x = y$ line on these figures highlight that on many benchmarks, CODAMOSA's average coverage does not differ from the baselines'. Looking at the crosses above the $x = y$ line, we see that CODAMOSA achieves a higher magnitude of coverage increases over MOSA (Fig. 3a) than CODEXONLY (Fig. 3b). This is consistent with the flattening in Fig. 2b.

A question which emerges from these results is whether the synergy between SBST and Codex demonstrated in Fig. 1 exists beyond that example, or whether CODAMOSA returning the union of MOSA and CODEXONLY test suites.

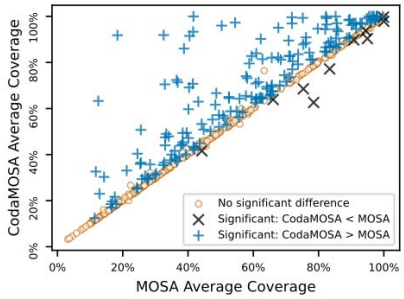
To evaluate this, we re-ran the generated test suites through a third-party coverage tool, `coverage.py` [36]. We created *Union* test suites by taking the union of test cases generated by the i^{th} MOSA run and those generated by the i^{th} CODEXONLY run. This is a pessimistic comparison for CODAMOSA, as it gives *Union* $2\times$ the search time allotted to CODAMOSA.

We were able to successfully replay the CODAMOSA and *Union* testcases through `coverage.py` for 429 of our benchmarks.[1] Of those benchmarks, CODAMOSA's coverage was significantly better than *Union* coverage on 72 benchmarks, not significantly different on 314 benchmarks, and significantly worse on 43 benchmarks. The fact that CODAMOSA performs significantly better than *Union* on 17% of these benchmarks, and matches its performance on 73% of benchmarks—despite having only half the search time of *Union*—suggests the meaningful combination of SBST and LLMs seen in our motivating example is not a one-off.

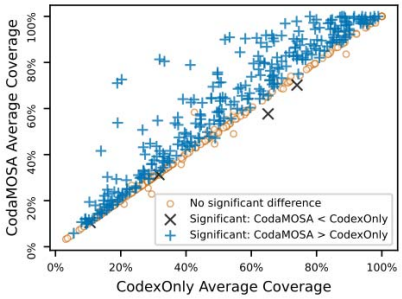
Further, these coverage increases do not seem to come at the cost of test suite bloat. On average over all benchmarks, CODAMOSA created test suites containing 11 tests, while MOSA's contained 10 tests. The difference in test suite size appears to correlate to coverage differences. On those benchmarks where CODAMOSA had significantly higher coverage than MOSA, its average test suite size was 18, compared to MOSA's 15. Meanwhile, on the benchmarks where MOSA had significantly higher coverage, MOSA's average test suite size was 18 compared to CODAMOSA's 17. CODEXONLY's test suites are smaller, containing on average 7 tests. But, as discussed previously, they achieve consistently lower coverage.

Takeaway. CODAMOSA achieves significantly higher coverage on many more of our benchmarks (173 vs MOSA, 279 vs CODEXONLY) than it reduces coverage on (10 vs MOSA, 4 vs CODEXONLY). On 17% of benchmarks, CODAMOSA outperformed the union of CODEXONLY and MOSA, even when allocated only half the search time. CODAMOSA's tests suites are not much larger than MOSA's.

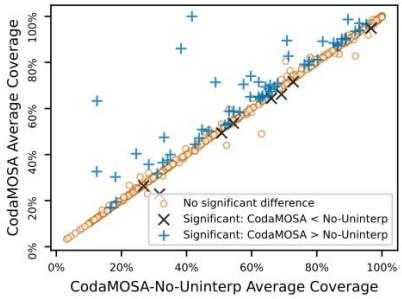
[1]The reasons for replay failures were often having to forcibly timeout test cases, especially for benchmarks from `youtube_dl.extractor`.



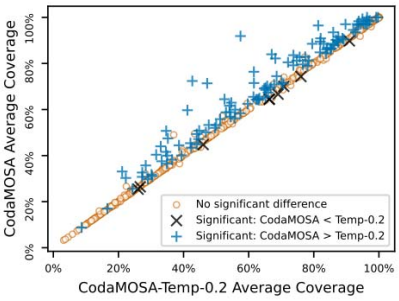
(a) Comparison to MOSA baseline. CODAMOSA has significantly higher coverage on 173 benchmarks; lower on 10.



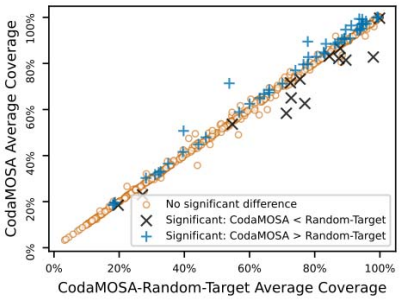
(b) Comparison to CODEXONLY baseline. CODAMOSA has significantly higher coverage on 279 benchmarks; lower on 4.



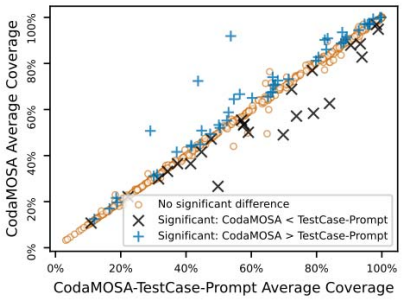
(c) Effect of uninterpreted statements. Using them, CODAMOSA has significantly higher coverage on 57 benchmarks; lower on 8.



(d) Effect of temperature. Default temp. 0.8 achieves significantly higher coverage than temp. 0.2 on 113 benchmarks; lower on 9.



(e) Targeting low-coverage functions has significantly higher coverage than targeting random ones on 50 benchmarks; lower on 14.



(f) Default CODAMOSA has significantly higher coverage than test-case-prompted version on 49 benchmarks; lower on 24.

Fig. 3: Average coverage achieved by CODAMOSA vs. baselines at the end of search time. Modules where CODAMOSA gets significantly ($p < 0.05$) higher coverage are marked $+$; those where it gets significantly lower coverage are marked $\times$.

C. Effects of Design Decisions

In building CODAMOSA, we made several design decisions. We evaluate the effect of each of these in detail.

1) Uninterpreted Statements: CODAMOSA's *uninterpreted statements* (ref. Section IV-C4) allow it to produce assignment statements with arbitrary right-hand-side expressions.

We compare the performance of CODAMOSA to a version of CODAMOSA with uninterpreted statements disabled (CODAMOSA-NOUNINTERP). Fig. 3c shows the average coverage achieved per benchmark for CODAMOSA on the y-axis, compared to CODAMOSA-NOUNINTERP on the x-axis. Overall, the use of uninterpreted statements results in significant coverage increases on 57 benchmarks, and significant coverage decreases on 8 benchmarks. As we will see in Section V-D1, uninterpreted statements can exercise function behaviors that can only be exercised by complex constructs, e.g., iteration. However, uninterpreted statements may also lead to the inclusion of useless statements in test cases. This makes it harder for the genetic algorithm to make progress via mutation, thereby reducing its achieved coverage.

Takeaway. Uninterpreted statements yield large coverage increases on a non-negligible fraction of benchmarks—those whose behaviors can only be exercised via certain program constructs. But on most benchmarks, the same improvements over MOSA can be achieved without them.

2) Temperature: Codex, like other LLMs, learns to generate code by learning a probability distribution over next tokens, given a prefix. There are several strategies for generating code from these probability distributions. We use *temperature*, which controls the greediness of sampling. With a temperature of 0, Codex always returns the most likely next token, while with a temperature of 1, it returns a next token sampled according to the learned probability distribution.

By default, we use temperature 0.8 to sample Codex. Fig. 3d shows a comparison between the coverage achieved with this default (on the y-axis), and CODAMOSA with temperature 0.2 (on the x-axis). CODAMOSA had significantly higher coverage than CODAMOSA-TEMP-0.2 on 113 benchmarks, and significantly lower coverage on 9 benchmarks.

A lower temperature means Codex is more likely to sample the *most* expected test case given a particular prompt. However, it also results in less variety. Thus, if Codex's most expected test cases are not useful to CODAMOSA, repeated queries that return similar expected test cases will not increase coverage. A higher temperature may require more queries to get the most expected test case, but is more robust in scenarios where this most expected test case is not useful.

Takeaway. Of all our evaluated design decisions, sampling Codex with a higher temperature has the most consistently positive effect on achieved coverage.

3) Targeting Low-Coverage Functions: Recall that CODAMOSA prompts Codex to generate tests for lower-coverage test objects. We evaluate the utility of this sampling by comparing to CODAMOSA-RANDOM, which prompts Codex with a randomly-selected test object instead.

Fig. 3e shows CODAMOSA's average coverage on the y-axis, and CODAMOSA-RANDOM's on the x-axis. On 50 benchmarks, CODAMOSA has significantly higher coverage than CODAMOSA-RANDOM, but CODAMOSA-RANDOM has significantly higher coverage on 14 benchmarks. Further, compared to the use of uninterpreted statements or high temperature, targeting low-coverage callables does not yield large magnitudes of coverage increases: in Fig. 3e all but three points above the $x = y$ line lie *just* above the line.

We expect CODAMOSA-RANDOM to behave like CODAMOSA when low-coverage prompting degrades to random prompting. For instance, when there are very few test objects, or when all test objects are equally covered. Further, CODAMOSA-RANDOM could perform better than CODAMOSA if a low-coverage function's coverage cannot be increased (e.g., "Unparseable Constructs" in Section V-D2).

Takeaway. Targeting low-coverage functions gets consistent, but low-magnitude, coverage increases over targeting random functions. The contribution of low-coverage targeting to CODAMOSA's performance is less than that of uninterpreted statements and high temperature sampling.

4) Prompting: The prompts CODAMOSA sends to Codex consist only of the source code under test and a test function header. We observed in preliminary experiments that sometimes Codex repeatedly outputs code that is too far from Pynguin's test case format to be of use to CODAMOSA. For instance, Codex sometimes generates test cases that are mainly Python testing framework boilerplate. A natural question is whether prompting Codex with an example of a well-formatted test case could improve these results.

In CODAMOSA-TESTCASEPROMPT, we add an example of an already-generated test case in the prompts to Codex. We choose the smallest non-empty test case in the archive, because we found that Codex imitates the Pynguin format too much when given a longer test. On 49 benchmarks, CODAMOSA had significantly higher coverage than CODAMOSA-TESTCASEPROMPT. CODAMOSA had significantly lower coverage on 24 benchmarks. Fig. 3f shows that the coverage differences on those 24 benchmarks are generally of larger magnitude. This includes a benchmark on which CODAMOSA-TESTCASEPROMPT outperforms MOSA while CODAMOSA performs worse than MOSA (see Section V-D2).

Takeaway. More complex prompting yielded better results in some cases, but was less consistent than our simple prompting. Further prompt engineering could improve results.

D. Case Studies

In this section, we analyze why MOSA and CODAMOSA's results differ, on the benchmarks where they differ the most.

1) Reasons for Coverage Improvement: Recall that CODAMOSA achieved significantly higher coverage than MOSA on 173 benchmarks, too many to analyze manually. Thus, we analyze the 20 benchmarks on which CODAMOSA had the largest average coverage increases over MOSA (from 24.2% to 73.4% absolute percentage points).

Special strings. On 15 of the 20 benchmarks, string-valued data was key to the observed increases. These "special strings" range from single argument values (as in our motivating example in Fig. 1), to longer text being parsed (`json` and `docstring`), to special-case dictionary keys. While Pynguin can extract constant strings from the source code, Codex-generated strings of particular formats were not explicitly present in the module under test. Further, Codex-generated test cases seem to be more likely to put strings in the right place, while MOSA needs to explore which strings go to which arguments.

Backup callables. On 7 of the 20 benchmarks, CODAMOSA was able to correctly "set up" the target functions by invoking backup callables. Recall that MOSA only considers test objects and dependent callables. This sometimes prevents MOSA from setting up tests correctly, notably when type hints are missing. Codex could call backup callables that were important to set up: CODAMOSA integrated these into tests with its callable expansion. A 2019 study [37] found that only 2.6k of 70.8k collected Python repositories had any type annotations, so the ability to support code without type hints is significant.

Uninterpreted statements. On 5 of the 20 benchmarks, uninterpreted statements increased coverage. Three benchmarks had `yield` generators, which must be exercised via iteration. On those, Codex produced calls to `list` and `next`, or list comprehensions. In another case, comparison operators (e.g., `<=`) were necessary to exercise methods. MOSA is unable to generate these syntactical constructs on its own. Interestingly, on some benchmarks, a large number (e.g., 57.3%) of accepted Codex-generated tests had uninterpreted statements, but CODAMOSA performed just as well without them. In such cases, it has added "useless" uninterpreted statements into tests.

2) Reasons for Coverage Decreases: CODAMOSA suffered significant coverage decreases compared to MOSA (from -0.2% to -15.9%) on 10 benchmarks; we analyze them all. The core reason for coverage decreases is *wasted exploration time*. On average over all benchmarks, CODAMOSA issues 60 queries to Codex and spends 413 seconds waiting for these queries. This rises to 480 seconds on the worst-performing benchmarks. If we could reduce Codex querying time, CODAMOSA might perform better. For instance, if we reduced the query time on the worse-performing benchmark from 494s to 49s, CODAMOSA would get 62.6% coverage in the same time that MOSA gets 43.5% coverage (155s).

Wrong signature. For CODAMOSA's worst benchmark, the constructor of the main class is implicitly defined via inheritance, and requires an argument in order to run without error. Codex usually failed to correctly call this constructor. Another benchmark had a similar issue, where Codex rarely filled optional arguments for one of the test callables.

Unparseable constructs. On average over all runs and

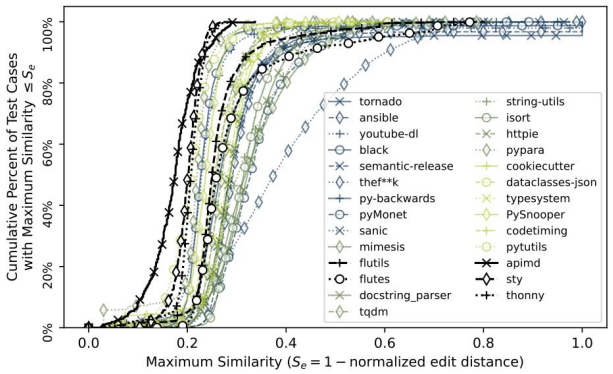


Fig. 4: Cumulative percent of Codex-generated test cases with maximum similarity less than what is designated on x-axis. Max is over all out-of-prompt test cases in the repository.

benchmarks, CODAMOSA parses 59.3% (min. 10.9% and max. 98.7%) of Codex-generated statements. Usually, these unparsed statements are not key to coverage increases, but on two of the worst-performing benchmarks, they would have been. On `sanic.cookies`, Codex generated many expressions of the form `cookies[key] = value`. On `flutils.objutils`, many Codex-generated tests included in-test class definitions.

Token limitations. We cut off Codex completions at 200 tokens—this helped reduce query time and seems like a reasonable length for a test case. However, on one benchmark, Codex spends all its budget on import statements and setup code, not managing to call the target function before the cutoff.

E. Similarity of Codex Generations to Out-of-Prompt Code

As mentioned in Section III-B, Codex was trained on a large corpus of code sourced from Github. As we do not have access to Codex's training set, we cannot confirm whether the code in our benchmark modules is part of this set. A natural question is whether Codex is overfitted to our benchmarks. To evaluate this, we compare the Codex-generated tests for each module to test functions present in the module's source repository *but not in the module's definition file*. These test functions are not in the prompt CODAMOSA sends to Codex, so we should not expect Codex to generate them.

Metrics for code plagiarism detection [38], which erase function names and primitive values to focus on code structure, did not suitably capture the notion of test case similarity. For instance, `copydetect` [39] would output that `ast.parse("x")` is highly similar to `collections.Counter("abb")`.

Edit distance [40] gives a more intuitive measure of similarity. E.g., the Codex-generated test for `flutes.iterator`:

```
assert list(scanl(operator.add, [1,2,3,4], 0)) == [0,1,3,6,10]
assert list(scanl(lambda acc, x: x + acc, ['a', 'b', 'c', 'd']))
                            == ['a', 'ba', 'cba', 'dcba']}
```

has low normalized edit distance (0.287) to a test from elsewhere in the `flutes` project:

```
check_iterator(flutes.scanl(operator.add, [1,2,3,4], 0), [0,1,3,6,10])}
check_iterator(flutes.scanl(lambda s, x: x + s, ['a', 'b', 'c', 'd']),
                            ['a','ba', 'cba', 'dcba'])}
```

Thus, we define maximum similarity as follows. For a Codex-generated test case t^* for module m in project P, and T_P, the set of test functions defined in P outside of m, the maximum similarity is $\max_{t_p \in T_P}\left(1 - \frac{dist(t^*,t_p)}{\max(len(t^*),len(t_p))}\right)$. We use `editdistance` [41] to calculate $dist$.

Fig. 4 shows the cumulative percent of Codex-generated test cases, per project, with maximum similarity less than the x-axis threshold. We see that the majority of generated test cases have similarity ≤ 0.4. For some projects, no testcases are remotely similar (`apimd`, `sty`, `thonny`), while for others (e.g., `flutes`, mentioned above) there is a long tail of test cases with higher similarity. The vertical line segments on the far right of the figure indicate some exactly-copied test cases (e.g. `tornado`): but 99.1% of these are simply the statement `pass`.

One of our test projects, `flutils`, is hosted on GitLab rather than GitHub, so is unlikely to be in Codex's training set. This allows us to evaluate how CODAMOSA works on "new" code. Of the 9 `flutils` modules we evaluated, CODAMOSA had higher coverage than MOSA on 7 ($p\text{-}value = 8 \times 10^{-3}$ to 2×10^{-8}), lower coverage on 1 ($p\text{-}value = 4 \times 10^{-3}$), and no coverage difference on 1 module ($p\text{-}value = 0.51$). While it is a small sample size, this 7/1/1 split compares favorably to the 166/9/296 split seen over the rest of our benchmarks.

Takeaway. Most Codex-generated tests are not very similar to out-of-prompt test cases. On benchmarks likely outside of Codex's training set, CODAMOSA performed well.

VI. THREATS TO VALIDITY

Internal: While we have made a best-effort attempt to make sure that test cases generated by Codex are deserialized correctly, it is possible that there are bugs remaining and that more code generated by Codex could be parsed.

External: While CODAMOSA had coverage increases over a large number of our benchmarks, these results may not hold for any arbitrary Python module. As seen in the evaluation, for many benchmarks, there was no significant coverage differences between our baselines and CODAMOSA. We automated our benchmark identification and filtering process in order to reduce the effect of bias in benchmark selection.

Construct: We use coverage to judge the goodness of test cases. There is debate on the relation between coverage and bug finding ability. A study from the fuzz testing space finds that although coverage and bug-finding ability are correlated, they do not agree on the ranking of different testers [42]. However, an industrial study of SBST [43] found it had difficulty finding bugs that required the construction of complex objects and generation of specific primitives. CODAMOSA outperformed MOSA on these points in our case studies.

VII. DISCUSSION

Contribution of Large Language Model: While the use of LLMs is central to our presentation of CODAMOSA, LLMs are not the solution to every problem in test case generation. In integrating LLMs into SBST, we faced challenges, and the fixes to these challenges may improve SBST more broadly.

Uninterpreted statements, for instance, could improve the use of human-written tests as seeds in SBST.

More Generic Representation of Test Cases: While uninterpreted statements in CODAMOSA conferred advantages on a number of benchmarks, we observed during our case studies that these statements sometimes cluttered test cases. A more generic representation of test cases—for instance, an augmented AST—could better capture the LLM output, but may also make the search less effective.

Test oracles: In order to reveal bugs, each test case must also contain a *test oracle*. A test oracle is a (set of) assertion(s) that should pass when the behavior of the module under test is correct, and fail otherwise. In this paper, we *do not consider* the test oracle generation problem. Given a test suite generated by CODAMOSA, different approaches for generating oracles can then be applied to each test case, including regression oracles [44], mutation-testing-based oracles [1], [2], and deep-learning-based approaches [45]–[48]. Because CODAMOSA is built on top of Pynguin, it inherits Pynguin's support for regression or mutation-testing-based oracle generation [49].

Data availability: Our source code is available at: https://github.com/microsoft/codamosa. The repository includes a replication folder containing: (1) a docker container to clone projects and filter modules as described Section V-A2; (2) a docker container to run CODAMOSA; (3) data from our similarity analysis; (4) scripts to generate the plots in this paper. It also contains information on how to access our raw evaluation data, notably the Codex-generated test cases that can be used to replay CODAMOSA runs.

VIII. RELATED WORK

ATHENATEST [50] is a deep learning approach for generating test cases. They train a transformer to generate tests from a large corpus of focal methods and test cases. In contrast, we use an LLM as an opaque black-box to generate tests without training, and incorporate these tests into a search algorithm. Developers liked ATHENATEST tests better than EvoSuite tests. As CODAMOSA imports test cases into Pynguin's format, its tests lose some readability. TICODER [51] also mutates LLM-generated tests, but with the goal of formalizing natural language intent. This work is complementary to ours: CODAMOSA does not need natural language intent to create tests, but requires the code under test to be fully implemented.

EvoSuite [1], [2] popularized the use of evolutionary algorithms for test suite generation. It starts from a random test suite, then repeatedly mutates it, saving mutated test suites that have higher coverage than the original. Mutations at the test suite level include mutating the individual statements in a test case, adding or removing entire test cases, or crossing over test cases or suites. The EvoSuite platform [15] now supports multiple search algorithms for test suite generation. These include generic multi-objective search algorithms like SPEA2 [52] and NSGA-II [53], as well as algorithms tuned specifically for test case generation, such as MOSA [3], DynaMOSA [3], and MIO [4], [5]. All of these algorithms include phases where test cases are randomly generated. MIO, for instance, directly trades off from an exploring phase where it mostly generates random test cases, to an exploiting phase where it mostly mutates promising test cases. An LLM-based hinting strategy in the style of CODAMOSA could be incorporated into these algorithms at these random test case generation points.

A study by Rojas et al. [54] studying the effects of seeding EvoSuite supports this idea. They find that including existing hand-written test cases in EvoSuite's original population of test cases increases coverage on 37% of their benchmarks. This finding is consistent with the results of this paper.

Several works aim to improve the performance of SBST without using LLMs. Galeotti et. al [55] use Dynamic Symbolic Execution to mutate test cases in which primitive values affect fitness. Lin et. al [56] propose a richer method to calculate branch distance for boolean branch conditions. EvoObj [8] uses static analysis of a target branch condition to build an object construction graph, which is then used to create better seeds for EvoSuite. All these improvements are complementary to CODAMOSA's hints-based strategy.

We believe CODAMOSA-style hinting can be applied to other automated test generation approaches. Randoop [44] is a technique for feedback-directed random test case generation: it generates test sequences by adding a new call to previously-generated function call sequences that run without throwing exceptions. TestMiner [57] extracts string literals from human tests suites to use as string constants in Randoop. When special strings are present in the test codebase, this technique could confer some of CODAMOSA's advantages. We could also add CODAMOSA-style test case hinting when growing sequences no longer increases coverage, or prompt Codex to extend sequences directly. Sapienz [58] uses reverse-engineering to extract strings in order to create more human-like interactions in Android tests; LLMs have the potential to create more generalized human-like interaction strings.

Bareiß et al. [59] compare the performance of Randoop to a Codex-only approach. To generate tests, they query Codex with a prompt containing one example of a method-test pair, then the body of the method to be tested. On an evaluation over 18 Java methods, they find this approach outperforms Randoop in terms of code coverage. But testing is not the only code-related application domain for LLMs. Bareiß et al. [59] also evaluate the use of Codex to generate code mutants and test oracles. Compared to baselines, Codex is relatively more performant at test oracle generation than code mutation. Jigsaw [60] uses LLMs to synthesize code that performs data transformations, then leverages user-provided test cases to post-process the LLM-outputted code. Their approach significantly outperforms the task-specific neural-backed generators in AutoPandas [61]. Kharkar et al. [62] use LLMs to filter out false positives in static analysis. To check null pointer warnings, they ask the LLM to complete the code before the flagged code. If the LLM adds a null check, they consider the warning to be a true positive. This yields 17.5% improvements in precision for null pointer warnings. Overall, these results suggest that strategic use of LLMs can improve the performance of many software engineering tools.

References

[1] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 416–419, ACM, 2011.

[2] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[3] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[4] A. Arcuri, "Many Independent Objective (MIO) Algorithm for Test Suite Generation," in *International Symposium on Search Based Software Engineering*, pp. 3–17, Springer, 2017.

[5] A. Arcuri, "Test suite generation with the Many Independent Objective (MIO) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.

[6] A. Aleti, I. Moser, and L. Grunske, "Analysing the Fitness Landscape of Search-Based Software Testing Problems," *Automated Software Engg.*, vol. 24, p. 603–621, sep 2017.

[7] N. Albunian, G. Fraser, and D. Sudholt, "Causes and Effects of Fitness Landscapes in Unit Test Generation," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, (New York, NY, USA), p. 1204–1212, Association for Computing Machinery, 2020.

[8] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-Based Seed Object Synthesis for Search-Based Unit Testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, (New York, NY, USA), p. 1068–1080, Association for Computing Machinery, 2021.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021.

[10] G. F. Stephan Lukasczyk, "Pynguin: Automated Unit Test Generation for Python," *CoRR*, vol. abs/2202.05218, 2022.

[11] flutils contributors, "flutils." https://gitlab.com/finite-loop/flutils, 2022. Accessed June 22nd, 2022.

[12] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[13] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[14] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, dec 2012.

[15] EvoSuite contributors, "evosuite." https://github.com/EvoSuite/evosuite, 2022. Accessed August 18th, 2022.

[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[17] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[18] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[19] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[20] K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[21] Sara Crouse, Sebastian Nagel, "Common Crawl." https://commoncrawl.org/, 2022. Accessed August 22nd, 2022.

[22] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[23] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

[25] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[26] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[27] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 21–29, 2022.

[28] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*, pp. 27–43, 2022.

[29] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?," *arXiv preprint arXiv:2112.02125*, 2021.

[30] J. A. Prenner and R. Robbes, "Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs," *arXiv preprint arXiv:2111.03922*, 2021.

[31] Pynguin Contributors, "`try_generating_specific_function` Implementation." https://web.archive.org/web/20230209202307/https://github.com/se2p/pynguin/blob/083c49677c55b5f94af619c25f727b5ff6ed8d63/src/pynguin/analyses/seeding.py, 2022. Line 751. Accessed February 9th, 2023.

[32] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for Python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, 2023.

[33] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (New York, NY, USA), p. 1556–1560, Association for Computing Machinery, 2020.

[34] Pipreqs Contributors, "pipreqs." https://github.com/bndr/pipreqs, 2022. Accessed June 15th, 2022.

[35] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 2011.

[36] Ned Batchelder, "coverage.py." https://github.com/nedbat/coveragepy, 2010. Accessed January 25th, 2023.

[37] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby, "Python 3 Types in the Wild: A Tale of Two Type Systems," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, (New York, NY, USA), p. 57–70, Association for Computing Machinery, 2020.

[38] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), p. 76–85, Association for Computing Machinery, 2003.

[39] I. R. Bryson Lingenfelter, "copydetect." https://github.com/blingenf/copydetect, 2022. Accessed August 18th, 2022.

[40] G. Myers, "A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming," *J. ACM*, vol. 46, p. 395–415, may 1999.

[41] H. Tanaka, "editdistance." https://github.com/roy-ht/editdistance, 2022. Accessed August 17th, 2022.

[42] M. Böhme, L. Szekeres, and J. Metzman, "On the Reliability of Coverage-Based Fuzzer Benchmarking," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1621–1633, 2022.

[43] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, 2017.

[44] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), pp. 75–84, May 2007.

[45] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Assert Statements for Unit Test Cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 1398–1409, Association for Computing Machinery, 2020.

[46] E. Dinella, G. Ryan, T. Mytkowicz, and S. Lahiri, "TOGA: A Neural Method for Test Oracle Generation," in *ICSE 2022*, ACM, May 2022.

[47] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, (New York, NY, USA), p. 54–64, Association for Computing Machinery, 2022.

[48] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using Transfer Learning for Code-Related Tasks," *IEEE Transactions on Software Engineering*, 2022.

[49] Pynguin Contributors, "Generating Assertions." https://web.archive.org/web/20220615155509/https://pynguin.readthedocs.io/en/latest/user/assertions.html, 2022. Accessed June 15th, 2022.

[50] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit Test Case Generation with Transformers," *CoRR*, vol. abs/2009.05617, 2020.

[51] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive Code Generation via Test-Driven User-Intent Formalization," 2022.

[52] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[53] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[54] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding Strategies in Search-Based Unit Test Generation," *Softw. Test. Verif. Reliab.*, vol. 26, p. 366–401, aug 2016.

[55] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 360–369, 2013.

[56] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 440–451, Association for Computing Machinery, 2020.

[57] L. D. Toffola, C.-A. Staicu, and M. Pradel, "Saying 'Hi!' is not enough: Mining inputs for effective test generation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 44–49, 2017.

[58] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-Objective Automated Testing for Android Applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 94–105, Association for Computing Machinery, 2016.

[59] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code," 2022.

[60] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large Language Models Meet Program Synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 1219–1231, Association for Computing Machinery, 2022.

[61] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "AutoPandas: Neural-Backed Generators for Program Synthesis," *Proc. ACM Program. Lang.*, vol. 3, oct 2019.

[62] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan, "Learning to Reduce False Positives in Analytic Bug Detectors," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 1307–1316, Association for Computing Machinery, 2022.