Genetic Fuzzing

Biologically inspired artificial intelligence – a.y. 2025/26

Luca Sartore, Jacopo Sbabo, Alessio Zeni

***Abstract*—Assessing code robustness is a fundamental aspect of many software applications, as it enables more resilient implementations while also enhancing security by identifying unintended or spurious behaviors.**
Robustness evaluation requires the definition of a structured testing suite, guided by one or more quantitative metrics, to systematically observe how a given codebase behaves under a variety of conditions.
A wide range of testing techniques exists, spanning from manual to fully automated approaches; these techniques differ in the aspects of the code they target (e.g., branch coverage, line coverage), in how inputs and outputs are generated or tracked (e.g., deterministic mappings versus random sampling), and in the feedback metrics they employ.
In this work, we focus on fuzzing, adopting Python as the reference programming language. Fuzzing is an automated testing technique that supplies random, malformed or unexpected inputs to a program with the goal of exposing erroneous or undefined behaviors.
The evolutionary component of our approach is realized through the use of novelty search, which drives the generation of inputs toward maximizing code exploration rather than optimizing for a specific failure condition.
To quantify the effectiveness of the proposed method, we rely on code coverage as the primary evaluation metric, leveraging the `coverage.py` library to measure both line and branch coverage.
Our results demonstrate that, although the proposed approach may perform comparably to simpler and faster techniques in worst-case scenarios, it is particularly effective when applied to deep and highly structured codebases. In such settings, it achieves superior coverage with fewer function calls compared to standard fuzzing strategies.

I. Introduction

CODE robustness is a cornerstone of programming; in many applications, especially security-dependent ones, reliability is fundamental to avoiding attacks and the exploitation of bugs. Standard applications also benefit from well-written and tested code, creating a smoother user experience and simplifying codebase maintenance.
For this reason, we must create test suites for our code, which generally aim for three different goals depending on requirements:

- **Bug-hunting:** looking for unwanted or anomalous behaviors
- **Requirements satisfaction:** ensuring the code satisfies the original constraints
- **Code coverage:** analyzing results from a wide variety of input combinations

Many testing techniques are available, divided into various categories depending on the level (single component, multiple components, complete software), approach (static, dynamic, passive), and the tester's point of view (black, white, gray box). It is also important to define metrics to evaluate the effectiveness of a test suite, though these are inherently application-dependent.
Generally, there is no free lunch; each application needs a proper evaluation strategy that provides a supervision signal and works in tandem with a tailored testing technique.

A. Main focus

In our case, we focused on *fuzzing*, an automated testing technique that involves the use of random, invalid, and unexpected data, typically employed when testing programs with structured inputs. During this process, the program is monitored for crashes, failing assertions, and memory leaks. Inputs that are at the limit of validity are particularly valuable, as they effectively test corner cases and help uncover bugs.

B. The genetic twist

To generate inputs for the code under test, various techniques may be employed, though standard approaches typically rely on random input generators.
However, random generators possess a fundamental flaw: as the input space expands, the number of possible combinations becomes intractable, making it difficult for the generator to produce valid or compatible inputs for effective testing.
To address this, we leverage a genetic algorithm—specifically *novelty search*. Instead of maximizing a traditional fitness function, this technique prioritizes novelty, which measures how "new" a specific individual is compared to the previously explored population.
Novelty search is an excellent candidate for software testing; unlike objective-based approaches that may converge prematurely toward local optima and limit coverage, novelty search promotes exhaustive exploration. This allows the algorithm to discover new execution branches and expose corner cases while avoiding the pitfalls of premature convergence.

C. Evaluation

The evaluation of performance was conducted using `coverage.py`, a Python library that provides specialized tools for measuring coverage during the testing process. Consequently, this project focused primarily on code coverage as the central metric.
In particular, we selected two types of coverage that specifically benefit from the exploratory nature of novelty search:

- **Line coverage:** measures the proportion of executable code lines actually traversed during testing.
- **Branch coverage:** evaluates the decision points within the code (e.g., if-statements), ensuring that both true and false paths are executed.

II. Related work

The research landscape surrounding the intersection of genetic algorithms and fuzzing remains relatively sparse; while studies have emerged over the last few years, the topic has yet to see widespread adoption. In particular, our approach of combining novelty search with fuzzing appears to be one of the inaugural attempts in this specific direction.
Nevertheless, we have selected several key works to highlight diverse methodologies for integrating machine learning with fuzzing:

- **Learn & Fuzz (Godefroid et al., 2017):** represents one of the foundational efforts in leveraging machine learning models to assist the fuzzing process.
- **V-Fuzz (Li et al., 2019):** utilizes an evolutionary algorithm for input generation, specifically targeting areas of the code predicted to be vulnerable. This methodology is the most closely aligned with our own objectives.
- **DARWIN (Jauernig et al., 2020):** a contemporary approach that employs an evolution strategy to dynamically adapt mutation probability distributions during fuzzing.

III. Overview

The general architecture of our system can be seen in figure 1. The idea is to start from a function we want to test, and use the type annotations to understand the input that the function takes. With this input we can use a "type adapter" (an interface that implements crossover and mutations oparations on top of existing python types) to generate a valid input to the function.

We can then execute the input, and measure the coverage. This information can then be used by a "strategy" to mutate the input, and try to improve the coverage. The main strategy we used is Novelty Search, but we also tested other as baselines.

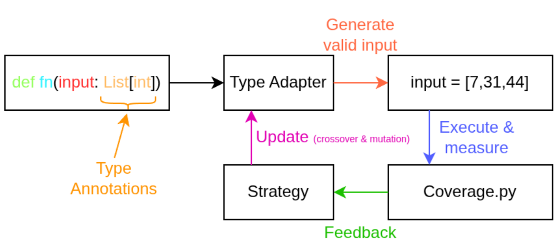

Fig. 1. Architecture overview

IV. Implementation

A. Tool Used

Our project is based on two main important tools:

1) Coverage.py: This is the primary python framework used for measuring code coverage. Code coverage is essential as it functions as our "objective function".

2) Type Annotations: The "Type Annotations" feature of python allows a function to be marked with some metadata that indicate what kind of input the function expect. This information is used to select what kind of type adapter (see IV-B) should be used.

B. Type Adapters

Type adapters are an abstraction that implements genetic operators (namely crossover and mutations) on top of existing python types (such as *int*, *bool*, *list* ecc.). We implemented type adapters on top of the basic python types, and we defined an API to "inject" new adapters in the system (so that users of our library can use this feature to fuzz-test with custom types).

Our implementation of the type adapters is recursive, meaning that when we define the adapter for *list* we automatically get the adapters for all the specialyzed thpes such as *list[int]*, *list[bool]*, *list[list[float]]* ecc.

C. Dataset

We tried to use some existing dataset, but unfortunatly none of the one we found were adeguate for our project. In particular we needed a dataset with type annotations, and we couldn't find one. So we generated one using Google Gemini APIs. The dataset has approximatly 100 function and is fully open source.

D. Test Cases

For a complete overview of novelty search's performance we created two other tequniques to compare against, one dosn't use any particular logic (Random) and another one is a traditional Genetic Algoritm implementation (Input Bag)

1) Random: This strategy simply generate random inputs and throw them at the functions in order to measure the coverage.

2) Input Bag: Input Bag is our implementation of a traditional genetic algorithm.

Applying GAs to fuzzing has a foundamental problem: We need to evoleve a set of inputs (not just one input) if we want to cover the entire function. To do so we had two options:

- **Bundling up multiple inputs in one individual** This idea is imple, we define a constant N that define how many function inputs one idividual si made of, and then we evolve the best N inputs that when evaluated together maximise coverage.
- **Using fitness sharing to encourage diversity** We can make each individual represent one singlie function input, and then we can use fitness sharing (or other similar techniques) to encouraje individual diversity. In this way the entire population will become our final output (not only the fittest individual)

both teqniques were valid, we decided to use the first one for our work. We tested various selection replacement strategy, and we found that by far the most important thing was to use elitism (which makes sense given the highly non-linear nature of the problem). We also found that adding steady state replacement to introduce novel genetic material helpd a bit.

3) Novelty Search: Our implementation of novelty search is pretty standard (it keeps a list of all the "novel" individuals, and it then uses them to generate newer one). What is not straightforward is the way we define "Novelty". The general idea is to generate an hash-able summary of the parts of the code hit by a certain input, so that we can understand if one specific input is "novel" with a simple hash-map (making this an $O(1)$ time operation). For example, in the case of line coverage the "summary" is a set containing all the lines that are executed. An input is considered "novel" if and only if the set of lines executed is different from all the others. This means that we could end up selecting individuals that cover less code then the one we already have, assuming that they are doing something different. This behaviour is desired, as there are cases where reducing coverage can lead to exploring new paths, for example:

```
A = True
# not executing this if can increase
# coverage if C2 == Ture
if C1:
    A = False
if A and C2:
    to_something()
```

V. Results

A. Setup

In our test we evaluated the performances of the three stretegies using the entire dataset. We ran two different test senarios:

- **Function call parity:** in this senario each strategy gets a limited ammount of call to the tested function, and we see which strategy worsk better with this constraint
- **Execution time parity:** in this senario we give each straetegy a fixed execution time, and we allow them to work untill the clock run out.

In the table below (V-A) we can see how, when using the "fixed execution time" strategy novelty search is able to call the test function 400 times less with respect two the other two strategies, which puts it at a significant disadvantage.

The reason of this behaveour is that coverage.py has a significant overhed associated with starting a test. With *Random* and *Input Bag* we are able to admortize this cost by bundling up multiple function call into a single run, however it is not possible to do the same using *Novelty Search*.

Given that this is more of a limitation associated with coverage.py that something caused by *Novelty Search* we decided to consider "Function call parity" for our final results, given that doing otherwise would make *Novelty Search* useless

Strategy	**Total number of function calls**	
	Fixed num of func calls	**Fixed execution time**
Random Search	10,000	∼4,000,000
Input Bag	10,000	∼4,000,000
Novelty Search	10,000	∼10,000

B. General Trend

In the plot 2, we can see that random search, and input bag performs in a similar way, with random gives consistently more coverage than input bag. This is probably caused by a more aggressive exploration in the random search approach, as well as the fact that input bag has a constraint in the number of inputs inputs that it can use.

Novelty search outperforms both strategies in the end. We should note that the reason why novelty search is really lagging behind in the beginning is that the first iteration of the other two strategies has already run many more function calls due to the inputs being bundled.

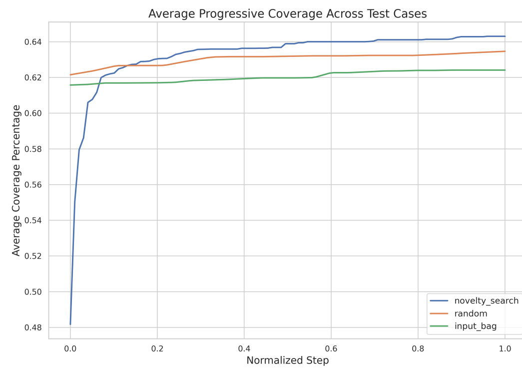

Fig. 2. Coverage progression over time

We had approximatly 400 similar charts (for each individual function, and for both function call parity and execution time parity) that we can't include in the report for obvious reasons, but you can find them on github if you are interested.

C. A theoretical testcase: count_bool vs is_odd_for_dummies

While testing the applications, we developed two pathological functions, in order to test the theoretical strengths for each implementation.

- **count_bool:** this function that takes a large amount of boolean inputs, and counts of many true values are present until the first false. We implemented it as a deep succession of if's. (code here).
- **is_odd_for_dummies:** this function takes an integer in input, and it has a large succession of if's, and each if handles a particular integer, returning true if it is odd, false otherwise. (code here).

The first function is very deep, in the sense that to score an high output, all the previous condition must evaluate to true, crafting an input that goes deep in to the function easier to do when done progressively (using evolution) instead of doing it one shot (using random input). We can see the results in plot 3). Instead in the second case the function is very shallow, because to get a particular branch the algorithm must exactly generate a particular value, without any previous information to build it. This advantages exploration over exploitation, reducing the novelty search advantage (as shown in plot 3).

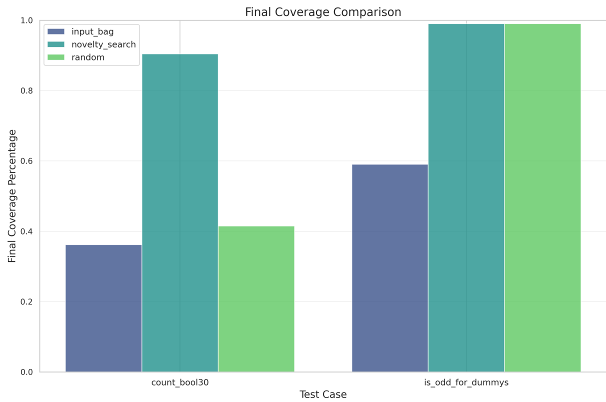

Fig. 3. count_bool vs is_odd_for_dummies

D. *The implications of the No Free Lunch Theorem*

In our settings, we decided to test some general arbitrary functions, and potentially each computable problem can be encoded in a function to test. Because of this, It is the perfect application for the No Free Lunch Theorem, which states that each optimization algorithm will perform on average the same on all the possible problems. This perfectly fits what we disclosed in the section V-C..

E. *Limitations of Coverage.py*

We would like note how the coverage.py framework did have some limitations that inpacted our results. The first one is the overhead associated with the test start that we discussed in section V-A. However that has been addressed by seitching to "function call parity" as the evaluation method.

The second limitation is perhaps harder to address. Coverage.py only measures line coverge and branch coverage, but there are other ways to measures this that could work better for our usecase. For example Jest (javascript's testing framework) also measures *Statement Coverage* that is more detailed and would result a stronger supervising signal. For example, take the following code:

```
if C1 and C2:
    do_something();
```

In the example we want to make C1 and C2 both true at the same time if want to cover the line inside the if statement. Using line coverage a mutation that would make C1 true would go unnoticed, as it would not effect the coverage in the immidiate term. If we where instead using statement coverage the mutation that make C1 true would immidiatly result in an higher coverage, and therefore an higher likelyhood that the individual is selected for future generations and would maybe be able to make C2 true in one of the following mutations.

Another useful metric that coverage.py does not collect is the amount of times that a certain line/branch is hit. Instead coverage.py only collect information on the fact that the line is executed or not. Ignoring any sort of counter. It is easy to see how this kind of information could be usefull when testing code such as this one:

```
counter = 0
while condition():
    # a line counter here would help
    counter += 1
if counter > 3:
    do_something()
```

VI. Conclusions and Future Work

In conclusion we can say that using novelty search for fuzzing looks promising.

There are definitly some senareos where novelty search outperforms our random and input bag baseline with significant margins. However there are also senareos where novelty search is outperfomed by even a simple random search.

A two main challenges that need to be addressed before the framework we developed could become usefull in a production eivironment, and they are both associated with coverage.py:

- **High overhead** As we explained in section V-A due to overheads associated with the initialization of coverage.py we had to use "function call parity" for all of our tests, however in a produciton enviromnent "execution time parity" is much more alligned with how the costs of various CI-CD pipelines is calculated, and therefore should be the main metric considered when evaluating the performances.
- **Week supervising signal** As we alluded in section V-E coverage.py only have limited ways to measure coverage (namely line coverage and branch coverage) but more complex (and potentially more effective) methods exists, that could result in a supervising signal that is stringer, and therefore can better guide our algorithm.

In future works we would like to address those limitations by either re-implementing coverage.py to address it's issues or more realistically by switching to a programming langage that already has a better testing framework implemented.

It is also clear to us that givven the high generality of the concept of "A function" and givven the implications of the No Free Launch theorem, it will be impossible to have a single algorithm that can efficently fuzz-test any kind of fuction. To address this limitation it would be interesting to have test a "Mixture Of Strategies" approach, were different strategies are executed in the same function, and over time only the one that acieve the best results are kept while the ohter are discarted.

VII. Individual