Kazantsev Daniil
January 12, 2024

Test Assignment

TU Delft and JetBrains
Track #1

Technical Part

Progress:

I have implemented a generic genetic algorithm in Kotlin. The code is located in **genetic_algo/genetic_algo.kt**. This function only requires hyperparameters for operation. I have also implemented NSGA-II and simple genetic algorithms for generic functions, accommodating both generic input semantics and outputs.

The process involves:

1. Parsing input parameters.
2. Parsing the config file, which includes bounds for different variables and their variance.
3. Parsing the semantics of inputs for the testing function.
4. Generating the initial population.
5. Running the following loop until the algorithm's time limit is reached:
   - If a simple genetic algorithm is requested, we sum the outputs of the function. This sum serves as our objective function. The next population is then generated using tournament selection, mutation, and crossover.
   - If NSGA-II is requested, we first determine the ranking for each individual. We then generate the next generation and rank all individuals based on their front number and distance within the front. The top 'n' individuals are selected for the next generation.
6. Writing number of generations and the last generation to file

How to run:

1. You need to create a .json file where the keys are substrings of all variables. **Temporarily (should be fixed)**, your project should include the file `org.jetbrains.person.PersonTest`, containing the function `personTest::TestPersonGeneric`. This is a significant issue because I haven't yet implemented the functionality to pass the function as a parameter during compilation. Using reflection to invoke the function by its name might not be the best approach. An alternative solution I've considered is constructing a .kt text file by filling in a template. Afterward, this file could be compiled, which shouldn't introduce significant overhead.
2. You should compile the Java library with the `-directories` flag. This is necessary because the parser of function semantics requires the variable names.
3. You should compile **genetic_algo/genetic_algo.kt** with paths of target libraries.

```
kotlinc -cp "../target/test-classes:../target/classes" genetic.kt -d genetic.
                                                    jar
```

4. You should run the compiled script by calling the following command:

```
kotlin -cp "../target/test-classes:../target/classes:genetic.jar" -classpath
                                                genetic.jar GeneticKt N_CLONES SECONDS
                                                 PARAMS_CONFIG_PATH USE_NSGA
                                                TOURNAMENT_SIZE MUTATION_PROBABILITY
                                                OUTPUT_FILE_PATH
```

   - `N_CLONES` - number of individuals in each generation.
   - `SECONDS` - seconds for life.
   - `PARAMS_CONFIG_PATH` - path to JSON file, where are bounds for each variable and their variance.
   - `USE_NSGA` - boolean value if using NSGA-II is needed.
   - `TOURNAMENT_SIZE` - tournaments size for getting next generation.
   - `MUTATION_PROBABILITY` - probability of mutation.
   - `OUTPUT_FILE_PATH` - path for output file.

Findings:

Both of the methods find cracking solutions in 10 seconds on different tasks. But I should notice, that simple genetic algorithm makes 1.5x more generations.

Recommendations for the next steps:

1. Resolve the issue where the algorithm is currently only able to search for bad inputs in the function named `personTest::TestPersonGeneric`
2. Refactor the existing code to enhance readability and simplicity by incorporating additional classes and type aliases. Utilize external libraries for more efficient argument parsing. Additionally, modify functions, such as .map and for-loops in NSGA-ranking, to leverage parallel processing and multi-threading for improved performance and efficiency.
3. Compute different metrics of success and compare the quality of generation algorithm

Research Part

Summary of the paper

Overall

The paper discusses methodologies for improving black-box and non-targeted attacks, as well as the ALERT (Naturalness Aware Attack) on neural networks that process code snippets. Specifically, it utilizes two pretrained architectures: CodeBERT and GraphCodeBERT. The paper focuses on three primary model tasks that are susceptible to attacks:

1. Vulnerability Detection
2. Clone Detection
3. Authorship Attribution

The Metropolis-Hastings Modifier (MHM) is used as a baseline. The core idea of the algorithm is to rename variables in a code snippet to induce model failure. A significant advantage of the proposed methods is that the new variables appear more natural and are harder for people to differentiate from the original ones (for example, replacing **buffer** with **qmp_async_cmd_handler** is not sufficiently natural). One of the improvements, particularly the Naturalness-Aware Substitution, can also be applied to MHM, and MHM-NS demonstrates strong performance. Furthermore, the paper describes the Greedy-Attack algorithm and the Genetic-Algorithm attack for identifying more effective substitutions. An "effective" substitution means that the authors aim to change a minimal number of variables, make fewer requests to the victim model, and still successfully compromise the model.

Naturalness-Aware Substitution

The authors leverage two properties of neural networks: masked language prediction and contextualized embedding. The algorithm utilizes the first property by generating substitution candidates. This is done by replacing variable names with **<MASK>** tokens and predicting them. Subsequently, the best substitutions are selected based on the cosine similarity between the embeddings of the substitutions and the original variables. This selection process uses the second property of modern neural network architectures.

Greedy-Attack

1. The initial step of the Greedy-Attack involves sorting variables by their Overall Importance Score (OIS). The OIS is calculated as the sum of the Importance Scores (IS) for all instances where a variable is replaced with **<UNK>** tokens. The IS is determined by the difference in the model's certainty with the replaced token versus its certainty with the original input for the target task. This prioritizes changing the most important variables first.

2. Next, the algorithm iteratively examines the variables in descending order of OIS, seeking the substitution for each variable that results in the most significant change.

However, this greedy algorithm is based on two assumptions:

1. The probability of successfully compromising the model is assumed to be monotonous across all variables when considered independently.

2. The probability is also presumed to be monotonous with respect to OIS. This means if there is a substitution involving n variables, a viable substitution should also exist among the first n variables in the list sorted by OIS.

GA-Attack

Given that the assumptions underlying the Greedy-Attack are not always accurate, initiating a genetic algorithm for finding replacements becomes a viable approach. This algorithm employs a single distance function and aims to optimize the distance between the victim model's prediction on an adversarial example and the original program. This approach helps in identifying more effective replacements that the Greedy-Attack might miss due to its inherent assumptions.

Metrics and Experements

1. An experiment was conducted to assess how natural the replacements were, involving real assessors. This experiment demonstrated that ALERT and MHM-NS generate significantly more natural substitutions compared to MHM, indicating that the Naturalness-Aware Substitution is a substantial improvement.

2. Various metrics were then computed on test datasets:

   - **Attack success rate**
     ALERT shows improvements in all configurations and outperforms other solutions, with Greedy Attack dominating over MHM.
   - **Variable change rate**
     ALERT again shows improvements in all configurations, except for Clone Detection with GraphCodeBert. It dominates over other solutions, and Greedy Attack outperforms MHM.
   - **Number of Queries**
     In this metric, Greedy-Attack performs the best. Therefore, if each request incurs a significant cost, the Greedy Algorithm is the preferable choice, since it is "greedy".

The most interesting findings of the paper

In my subjective opinion, the most interesting finding from this study is the use of language model peculiarities and possibilities for selecting better and more natural candidates for variable substitutions. However, it's also

crucial to highlight the substantial quality boost provided by robust methods like the Greedy-Attack and genetic algorithm. These approaches demonstrate the effectiveness of leveraging advanced computational techniques in enhancing the precision and naturalness of substitutions in code.

Future steps of this study

Several ideas could enhance the Naturalness-Aware Substitution and Greedy Attack methods. For instance, instead of computing the Overall Importance Score (OIS) as a summation, it might be more effective to calculate it just once by replacing all instances of a single variable. Additionally, in the Naturalness-Aware Substitution, selecting the top candidates could be improved by using the embedding of the entire code snippet. While it's possible that the authors have already tested these straightforward ideas, there's a significant opportunity for advancement in the genetic algorithm.

A major enhancement could be to employ a multi-objective optimization approach, such as NSGA-II :D. This method would allow simultaneous optimization of multiple metrics, not just the probability of failure but also factors like the Variable Change Rate. Furthermore, I guess that there exist methods that add a penalty for the number of generations and this could optimize the number of requests to the model, enhancing efficiency. These improvements could significantly refine the algorithm's effectiveness and applicability.