Finding Small Code with Large Binaries - Progress Report

VIKTOR GSTEIGER and NIKODEM KERNBACH*, ETH Zürich, Switzerland

We present the current state of our project, which includes evaluating the progress, identifying pending tasks and rescheduling the work. The general framework is complete, with minor necessary improvements pending. The time constraints set in the project proposal were met.

1 CURRENT PROGRESS

We implemented the framework[1] in Python to (1) randomly generate valid code samples using *csmith*[2], (2) use *creduce*[3] to apply code equivalence transformations and reduce code size in parallel to generate multiple next-round candidates, (3) filter out candidates that violate the constraints as presented in the project proposal, (4) evaluate the candidates according to an initial heuristic, and (5) repeat step 1. The framework can be started with a multitude of parameters which will influence the behaviour and output quality.

We decided to use *creduce* instead of the initially proposed mix of both *creduce* and *cvise* due to compatibility reasons. We additionally utilized *diopter*[4] to test and clean given programs.

In order to align code equivalence transformations with the goals of our project, we developed an interestingness test (see listing 1) to be utilized by *creduce*. In the context of *creduce*, an interestingness test is a binary function that evaluates whether the generated code is interesting (1) or not (0). In our interestingness test we first compile the original and the newly generated code, apply a test whether the newly generated is still executable, and then compare the text size of the two binaries. Only larger binaries are interesting to us. By default, *creduce* applies various transformations and optimizations until no further reduction is possible or a timeout occurs, and then returns the final result in a copy of the input file. However, our framework requires a slightly different approach because we aim to generate diverse candidates. Simply returning a single file would impose significant limitations. Therefore, we modified the interestingness test to copy all interesting candidates into a subdirectory outside of the temporary directory created by *creduce*. These candidates are then evaluated in step (3) and (4).

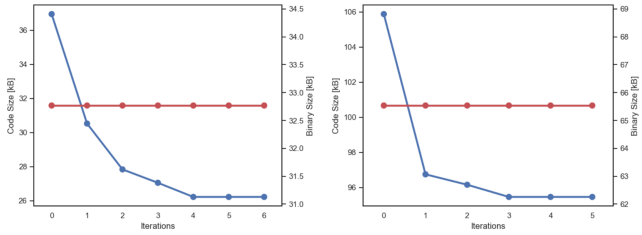
```
if [ $(size tmp.o | awk '{{print $1}}' | tail -n 1) -ge $(size orig
    .o | awk '{{print $1}}' | tail -n 1) ]; then
  random_string=$(mktemp XXXXXXXXXXXXXXXX)
  cp {local_new_source_code_path} {iteration_dir}/interesting_${{
      random_string}}.c
  exit 0
fi
exit 1
```

Listing 1. Interestingness Test: Removed is the compilation of the original code to `orig.o` and the newly generated to `tmp.o` as well as a test if the newly generated code compiles.

As outlined in the project proposal, our heuristic function comprises four parameters: (1) the size of the `.text`-section of the binary, (2) the source code size, (3) the binary size to source code ratio, and (4) the difference to previously generated mutants. Our implementation of the *interestingness test* already satisfies (1), while we allow for some leeway in (2) if it performs exceptionally well with respect to (3), in order to avoid restricting our design space. Although (4) is not yet implemented, it is scheduled for implementation soon.

2 RESULTS

We already achieved some good results with the current state of our framework. Two exemplary runs can be seen in Figure 1, and in Figure 1a we almost double the binary-to-code-size ratio. While our framework is good at reducing the code size and therewith increase the ratio, it fails to significantly enlarge its binary size. Moreover, we observed that it often quickly converges to a local minimum and does not improve after a few iterations. These challenges will be tackled before the hand-in of our project.



(a) Run 1. (b) Run 2.

Fig. 1. Exemplary framework runs.

3 TIMELINE EVALUATION

We managed to achieve all our current goals which were an initial implementation of the framework, an initial heuristic as well as this report. We propose to stick to the planned timeline with regards to the further progress of the project with some rescheduling and shifting of priorities as discussed in Section 5

4 PENDING TASKS

The following tasks are open with regards to the further progress of the project:

- Extend and optimize heuristic
- Implement local minima detection and prevention
- Ensure reproducibility
- Find better parameters for the *csmith* generator

5 RESCHEDULING OF TASKS

We propose the following schedule in Section 5 for the outstanding tasks with regards to the project.

Task	Deadline
Optimize heuristic and ensure results are satisfactory	14.05.2023
Local minima detection and code optimizations	21.05.2023
Final report hand-in date	06.06.2023
Presentation recording hand-in date	10.06.2023

Table 1. Further Timeline

[1] https://github.com/vGsteiger/ast-project
[2] https://github.com/csmith-project/csmith
[3] https://embed.cs.utah.edu/creduce/
[4] https://github.com/DeadCodeProductions/diopter

Authors' address: Viktor Gsteiger, vgsteiger@student.ethz.ch; Nikodem Kernbach, nkernbach@student.ethz.ch, ETH Zürich, Switzerland.