Variable Name Reconstruction Using LLMs and Program Analysis

Sergio D. Rodríguez De Jesús

Introduction:

Reverse engineers rely on decompilers for effective binary analysis. While contemporary decompilers excel in many aspects, a persistent challenge lies in the recovery of variable and function names. Typically, these identifiers are discarded during compilation, resulting in decompilers outputting names devoid of semantic meaning—often mere representations of memory addresses. Addressing this limitation, Xu et al. [1] introduced LmPa, a system that employs the synergy between language models and program analysis to reconstruct variable names.

However, it's notable that LmPa [1] utilizes GPT-3.5, and there is an observable trend toward employing larger models for improved outcomes. In this context, we propose a hypothesis: by employing smaller, more specifically trained models, comparable results can be achieved, thereby enhancing accessibility and minimizing costs. We aim to re-implement LmPa [1] utilizing lower-cost alternatives like the Ghidra Decompiler instead of IDA Pro Hex-Rays and models like CodeBERT [2] instead of GPT-3.5.

Methodology:

As we want to compare our results with the results of Xu et al. [1], we will use the same benchmarks and evaluation metrics as they did. This involves decompiling 6 codebases (Coreutils, Findutils, SQLite, ImageMagick, Diffutils, and Binutils) and using them as inputs to our system.

To implement this system, we use python together with various libraries. Most notably, we use Hugging Face's Transformers library in order to run inference with LLMs and facilitate swapping out models for testing; and pycparser to extract variable names from decompiled output.

Overview:

Similar to LmPa [1], our system comprises the following key components:

1. Prompt Generator:

This module takes decompiled output as input, extracts relevant data from the functions within, and generates a series of prompts that will serve as input to the LLM. These consist of an instruction about the format for the LLM to follow as well as a question about the variable names in a single function. The example provided by Xu et al. [1] goes like this:

```
Can you help me guess some information for the
following decompiled C function from a binary
program?
The following is the decompiled C function:
<<<selected function>>>
In the above function, what are good names for
<<<default variable names chosen by decompiler>>>, respectively?
You MUST follow the format <original_name> -> <new_name>.
```

2. LLM:

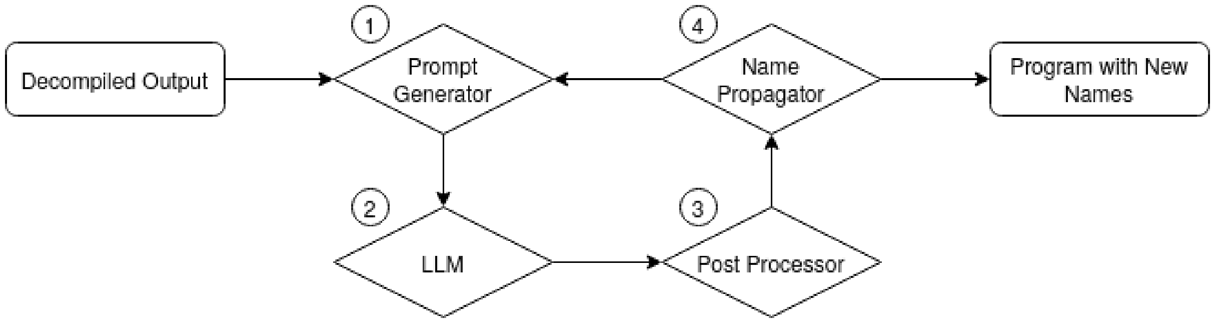
Responsible for producing contextually relevant guesses for variable names based on the provided prompts. The LLM enhances the semantic meaning of variable names within the given context.

3. Post-processor:

Ensures the robustness of the system by re-prompting the LLM in case of malformed data. Additionally, this component extracts guessed names from the LLM output for further processing.

4. Name Propagator:

Takes the guessed variable names and propagates them throughout the functions. This process expands the context, potentially aiding in generating more accurate guesses during subsequent passes.



Current Progress: (https://github.com/sergiodrd/name_reconstruction)

Prompt Generator:

The first step for generating prompts is extracting variable names from the decompiler output. To do this, we use the pycparser library. pycparser, as the name suggests, parses C source code and provides access to the generated AST. We use its API to visit variable declaration nodes in the AST and inject them into a prompt template together with the function's source code.

LLM:

The choice of LLM is crucial to this project. So far we are experimenting with popular small LLMs we find on Hugging Face. Self hosting LLMs is proving to be difficult and resource intensive, so we are exploring alternatives like Mistral and Llama 2 as well as their versions that have been fine tuned for coding tasks. We will be using high performance computers provided by bridges2 to test these models.

Future Work:

Prompt Generator:

It is important to experiment with and select an appropriate prompt to inject the functions and variable names into. This could have a big impact on the results of the project.

LLM:

There are a lot of things to experiment with, mainly, choosing the most fit LLM. If we continue with the self hosting route, we should look into quantization. For integration with the tool, it might be good to experiment with Ollama.

Post-processor and Name Propagator:

We need to implement these steps. The post-processor's main objective is to ensure the validity of the output from the LLM, re-prompting if needed and transforming it into proper input for the name propagator; this could be a simple parser for the format specified in the prompt. The name propagator is a huge component that requires a lot of care, however, Xu et al. [1] have outlined the algorithm they use.

References:

[1] Xu, X., Zhang, Z., Feng, S., Ye, Y., Su, Z., Jiang, N., ... & Zhang, X. (2023). LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis. *arXiv preprint arXiv:2306.02546*.

[2] Zhou, S., Alon, U., Agarwal, S., & Neubig, G. (2023). Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*.