Building an LLM Prompt Generator from Decompiled Data

Sergio D. Rodríguez De Jesús

Introduction:

Decompilers are extraordinary tools, but currently, they are lacking in terms of naming variables. This is because the variable name information that is clearly recognizable in source code gets thrown away during compilation, thus making it impossible to recover in the process of decompilation. This results in decompilers choosing semantically meaningless names (like the address where the variable is found). However, we believe that some semantic meaning can be recovered from the context in which the variables are found within decompiled code. This is a perfect use case for Large Language Models.

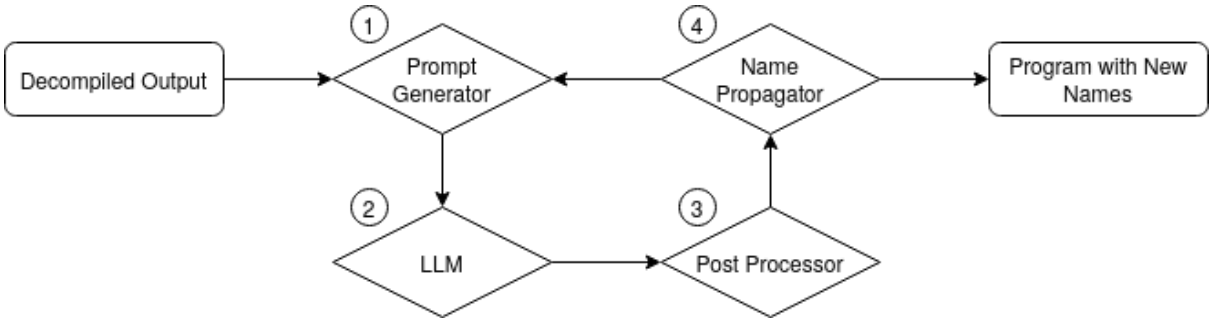

Figure 1.

In order to build a system that leverages Large Language Models and extracts meaningful information from decompiled code and the variables within it, there are multiple components that must be developed. The first of these, as you can see in figure 1, is a Prompt Generator. The purpose of this module is to take in decompiled code as input and produce a series of prompts that can be fed to an LLM, specifically, one for every function. Conceptually, this is very straightforward, but in practice, it is not trivial to implement, and there are some considerations to be had. For example, the development of this module has to be done in such a way that facilitates experimentation with LLMs, as this is the code that stands right in front of it.

Methodology:

In our testing, we tried a few strategies to extract variable names from decompiled functions. In parallel to this, we were also trying to find a way to produce and access data from the Ghidra decompiler programmatically. Quickly, we realized that the most straightforward way to extract the variable names was to get them straight from the Ghidra API.

We use a python library called pyhidra [2] for this, it works by launching the Ghidra decompiler in headless mode and establishing a connection to it via jpype. This is because our project uses python while Ghidra is written in java, and its API is only accessible via java. Pyhidra [2] gives us python bindings for the java API. This is effective, but it has the disadvantage of coupling our system with the Ghidra decompiler. This is a tradeoff we are willing to make for now.

There are two primary focuses for this module. The first is to aid in our experimentation with LLMs, and the second is to function purely as a component in the greater architecture of the project.

Experimentation:

With the focus on experimentation, we do a few extra things. First, we build up a repository of all the relevant data for some test codebases (Coreutils, Findutils, SQLite, ImageMagick, Diffutils, and Binutils). This includes the decompiled code, variable names, original and obscured function names, and the corresponding generated prompt for every function in every binary generated by these codebases. Notably, this is a lot of data, so we use numpy to sample from this repository in order to test the outputs of the LLM. We also include extra data in this repository that isn't necessarily needed for the prompts, like the original function name. We do this so we can later match the original source code with the prompts and with the LLM outputs.

Component:

When acting purely as a component, there is no need to hook into anything or do any sampling. The only output should be the prompt that should be fed to the LLM. This should be on a per-function basis rather than doing it in a batch like we do for experimentation.

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

Prompt Generator:

Previously, we had attempted to use pycparser, a python library that generates an AST from C source code. We had issues with this because Ghidra output is not exactly C, but pseudo-C. One big difference is the use of undefined types in Ghidra output. This is something that pycparser doesn't like, and would require weird workarounds like generating typedefs programmatically.

As mentioned before, we are getting variable names directly from the Ghidra API, so we don't have this problem anymore. We have implemented functions for generating the repository mentioned above, as well as the functions needed to output just the prompts.

LLM:

We are currently still using huggingface transformers in the high performance computers provided by bridges2 to run inference on models. This has proven to be difficult for our use case, as huggingface transformers is meant for machine learning researchers and training and fine tuning models. The APIs aren't as uniform as we'd like, and often we have to drop to a lower level to fix errors with specific models or specific inputs, which is not ideal, but we might be able to get better results in this regard if we use Ollama in bridges2. So far, the model that has proved to give the best results has been Mixtral 8x7B.

Future Work:

The Prompt Generator should take into account the fact that its input won't come exclusively from the decompiler, as the system is iterative and there will be some state that gets modified by the Name Propagator eventually. In order to prepare for this, there should be some degree of separation between the code that interacts with the extraction of information from the decompiler and the code that takes this information and generates prompts. Not only is this relevant for the Prompt Generator, but it will be relevant for other components as well.

The choice of LLM is important, but I consider that it is more important to find a consistent method of running inference on LLMs and swapping them out, as we have had trouble with this.

The Post Processor will be a very similar component to the Prompt Generator, and could be tackled next. It should validate the output from an LLM and re-prompt if it is

invalid. Once it is validated, it should parse out the new variable names and pass them along to the Name Propagator.

The Name Propagator is the most dense component of all of these, and it should be implemented as described by Xu et al. [1].

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] dod-cyber-crime-center (2024). pyhidra (1.0.2) https://github.com/dod-cyber-crime-center/pyhidra. URL.

[3] S Rodriguez. (2024) Variable Name Reconstruction for Decompilers using Large Language Models and Program Analysis. SIDIM. https://drive.google.com/file/d/1_N_Ldd8Ukc9xYFy71E6kSWGnU93nndif/view?usp=sharing.