Introduction

Reverse engineering is a crucial process in understanding and analyzing software systems, particularly when the source code or documentation is unavailable or incomplete. This process involves disassembling and studying an object, such as a binary executable or a software component, to gain insights into its structure, functionality, and behavior. However, reverse engineering can be a time-consuming and repetitive task, often involving numerous steps and requiring significant manual effort.

This project aims to develop tools and techniques to streamline and support the reverse engineering process, specifically focusing on software reverse engineering. By leveraging automation and providing a centralized platform, we intend to reduce the redundancy and effort associated with common reverse engineering tasks, enabling users to perform their analysis more efficiently.

Reverse engineering, as defined by TechTarget, is "the act of dismantling an object to see how it works. It is done primarily to analyze and gain knowledge about the way something works but often is used to duplicate or enhance the object.". In the context of software, reverse engineering involves analyzing compiled code, binaries, or other software artifacts to understand their functionality, identify potential vulnerabilities, or extract valuable information.

Static analysis, a key component of reverse engineering, is a software testing methodology that analyzes code without executing it and reports any issues related to security, performance, design, coding style, or best practices (DataDog Knowledge Center). By incorporating static analysis techniques, our tools will provide insights into the code structure, data flow, and potential vulnerabilities, enabling more comprehensive and efficient reverse engineering processes.

One of this project's main objectives is to reduce redundancies in the reverse engineering processes. By providing a centralized platform for loading, storing, saving, and analyzing files or text created during the reverse engineering process, we aim to streamline the workflow and eliminate the need for repetitive tasks. Also, we will develop an understanding of the needs and desires of reverse engineers, letting us enhance the utility and functionality of our tools in the future.

Furthermore, this project aims to leverage Python and existing tools to retrieve the necessary information for conducting static and binary analysis with minimal user input. By automating the data gathering and analysis processes, we can reduce the manual effort

required and enable users to focus on higher-level tasks, such as interpreting the results and making informed decisions.

In summary, this project seeks to develop innovative tools and techniques to support and accelerate the software reverse engineering process. By leveraging automation, static analysis, and a centralized platform, we aim to reduce redundancy, enhance efficiency, and provide a comprehensive solution tailored to the needs of reverse engineers.

Process, Methods

Reverse engineering is a complex process that involves various techniques and tools to gain insights into the structure, functionality, and behavior of software systems. This project aims to streamline and automate several aspects of the reverse engineering process, leveraging both static analysis and dynamic analysis approaches. The following section outlines the key ideas and the general process employed in this project.

Static Analysis Procedure:

Static analysis is a crucial component of reverse engineering, as it allows for the examination of code without executing it. The primary goal of static analysis in this project is to identify the file format, strings used, file size, potential dependencies, and develop a basic understanding of program flow and execution. To achieve this, the following steps are followed:

1. File Identification: The first step is to determine the file format and type using the 'file' command. This provides essential information about the binary, such as whether it is an Executable and Linkable Format (ELF) file, its architecture (e.g., 32-bit or 64-bit), and whether it is a dynamically linked executable.
2. Header and Section Analysis: The 'readelf' command is used to extract detailed information about the file headers, sections, and other metadata. This includes the image base (the preferred address for the binary to be loaded), entry point (the address of the first bytes to execute once loaded into memory), virtual addresses, and relative virtual addresses.
    a. Note: This will not work if the file is not in ELF format. PE formatted files are expected to be analyzed entirely with ghidra at this time
    b. There are checks for file type in the code (explained below), but it should still be noted.
3. String Extraction: The 'strings' command is employed to extract printable strings from the binary file. These strings can provide valuable insights into the program's functionality, such as error messages, user prompts, or other textual information.

4. Hexadecimal Viewing and Editing: The 'xxd' command is used to view and edit the binary file in hexadecimal format. This can be useful for identifying and modifying specific byte sequences or patterns within the binary.
5. Disassembly: The 'objdump' command is used to disassemble the binary file, providing a human-readable representation of the machine instructions. This can aid in understanding the program's control flow, function calls, and other low-level details.

The above steps are automated using Python scripts, ensuring a consistent and efficient workflow. The outputs of these commands are saved as strings or text files for further analysis and processing.

After reviewing the basic file information, we have a file to prepare a ghidra environment and a file to use a Large Language Model (LLM) to provide some initial opinions about the file in question.

Dynamic Analysis and Further Analysis:
In addition to static analysis, this project hoped to incorporate dynamic analysis techniques, which involve executing the binary and observing its behavior. This can provide insights into runtime characteristics, such as memory usage, system interactions, and execution paths. We were unable to effectively use x32dbg or gdb like tools through a script in any way that provided more utility than simply using the tools directly. Fortunately, we found that constraint solving fills in many (but not all) the gaps we came across when forced to attempt reverse engineering without dynamic analysis.

Furthermore, the project seeks to develop methods and prompts to facilitate further analysis and problem-solving. This may include techniques like constraint solving, symbolic execution, or leveraging advanced frameworks like angr or Ghidra.

Tools and Technologies:
To accomplish the goals of this project, several tools and technologies are employed.

1. Python: As the primary programming language, Python is used for automation, scripting, and integrating various tools and libraries. We chose python as it is readily available on our system and many other reverse engineers recommend it for scripting purposes. The substantial number of libraries solidified our choice.
2. Langchain: A framework for building applications with large language models that can aid in analysis and question-answering tasks. We decided to use langchain as we find it easier to add features too when needed. There are a lot of imports, and the code is messy as a result, but we found that using the openAI API alone does not

allow us to organize our code as nicely. We hope that by using langchain it is pretty clear where in the process you are without having to rely on review of the code. Most importantly we hope to add a graph using langgraph in order to conditionally modify our prompts based on previous outputs. While you can do this with the openAI API, the lack of conditional and deterministic changes based on analysis of previous outputs makes using model alone inconsistent.

3. angr: A powerful binary analysis framework that can be used for dynamic analysis, symbolic execution, and constraint solving. We spent most of our time attempting to learn this tool effectively, but we ran into issues we were not able to resolve. Many 'crackme' challenges are not so simple as to only require a specific key, and with our limited understanding obfuscation techniques, file inputs, output parsing, etc all reduce the utility of this tool. However, we were able to solve a few 'crackme' challenges much faster than we otherwise would have. This brough us pleasure.
4. Command-Line Tools: Various command-line tools are utilized, including 'file', 'readelf', 'strings', 'xxd', and 'objdump', each serving a specific purpose in the reverse engineering process. We found that putting these commands into a single file with a place to retrieve outputs reduced the initial file analysis we were forced to routinely perform. I find myself using the view_file_info.py file quite a bit with no relation to the project we are working on here.

Challenges and Future Work:

One of the main challenges faced in this project is the development of an autonomous pipeline that can reliably perform a consistent series of tasks from the input binary to the final analysis or constraint solving stage. The current thinking process of large language models is not yet reliable enough to be solely depended upon for such complex tasks. Future work may involve defining a more rigorous tool set and developing specialized autonomous agents or pipelines tailored for specific reverse engineering tasks. Additionally, integrating advanced symbolic execution techniques, taint analysis, and vulnerability detection could further enhance the capabilities of the developed tools.

Dynamic Analysis:

The process of using gdb or x32dbg requires a lot of thought between actions. You must have a general understanding of the program flow, you must have a general understanding of the goal you are trying to accomplish through the analysis, you must to be able to remember and retrieve certain memory addresses or code sections, and you may need to be able to restart the debugger (or your entire system) often. We initially tried to simply use the terminal to step through gdb through python, but we found that there was no effective way to save the outputs in a manner that was useful to review. We found ourselves forced

to review the output because we were unable to develop an algorithmic approach to debugging. There are many variables that would need to be constant to make this feasible with our tool and knowledge set. The lack of a reliable meaning to an entry point was the first obstacle. We were unable to reliably determine where an entry point would place us inside the program. In fact, many packed files do not even have entry points in the file we want to analyze. So, all other things aside, we first would have to determine where to begin. Then we found that when we manually performed dynamic analysis, we often stepped through the program arbitrarily during our debugging process; we do not develop (although we should) an attack plan before using a debugger. We must often rely on the contents of the registers to determine our next step, but what constitutes a 'good' register value is often arbitrary as well. We do not believe an algorithmic approach to debugging is impossible, in fact we believe everything has an algorithmic solution, however we were unable to determine one for this project.

Fully Automated Static Analysis:

The process of reverse engineering, debugging, solving 'crackmes', analyzing unknown binaries, etc. Is not as algorithmic as expected. Like we mentioned previously, we struggled with an algorithmic approach to debugging for the same reasons we were unable to find an algorithmic approach to fully complete static analysis. The actual process is often varied due to isolated unforeseen 'hiccups'. For example, something as simple as which toolset we can use is often varied by the file format, the file size, the packing algorithm, etc. Our algorithm would need to be able to select programs conditionally. This is an issue as the conditions are often not binary and depend on multiple features of the file (PE, 500mb file, packed with UPX) vs (ELF, 40kb file, unpacked). This means the program needs to be able to parse the system, the file, and itself to make changes. For LLM's this leads to context length issues. We find that LLM's are not particularly good at handling multiple tasks in the same way; humans are good at this. Take a glass of water left on your desk from yesterday, you first decide if you are thirsty, then look at the water and determine its drinkability, then decide (to drink or not to drink). That misleadingly simple task is in fact a composition of many tasks that each are quite computationally expensive on their own. Similarly, take deciding which portions of disassembled file you should save as a 'task'. To effectively make this decision you need to know quite a bit, for instance what is the goal you are trying to achieve? How much space do you want to use to store things? What is useful information? What do we do when we find more useful information than we have space for? What do we do if we find no useful information? As you can see there are a lot of conditionals that must be created just to decide what information to save (and that did not even include the conditionals to perform the task!). We found that LLM's were

unable to consistently make these decisions in a way we found useful. However, we do believe that this is partially due to our own inaptitude in reverse engineering.

Future Work:

We will continue to work with langchain/langgraph to develop ways to get an LLM (Large Language Model) to analyze responses from other LLM's. We believe that this will enable us to begin the actual process of finding an algorithmic solution to the reverse engineering problem. Finding this algorithmic process is where most of the future work lies. Fortunately, LLMs (Large Language Models) give us some flexibility; we do not necessarily need these steps to be exact or even have all the steps defined, but we do need some sort of routine that reliably leads to a solution consistently. LLMs let us abstract some steps of the process if we trust its responses. We then must return to the original process of analyzing our own static analysis routine and hopefully refine it enough to reach a point where we can identify not just commands and memory spaces, but entire thought chains that can be defined and replicated.

Key files:

Key files are listed below. We group the main files into sections based on our idealized use case. There is no right or required way to order the running of the files. We hope however that the following provides a guideline for use.

We built these files by analyzing our own process when solving 'crackmes'. We have found that these files do infact help our ability to understand, process and navigate binary files.

Code is available as github repository through this link (https://github.com/xiscoding/reverse-engineering). Further information is available in the README.md file located in the github repository.
Note: There are more files in the repository than listed below. Most of those files are still deeply in progress, but updates are to be expected in the future.
The key files will be provided in the zip file along with this paper, you will need to provide your own openAI api key as it is not free. We find that a complete analysis costs around 1-50cents depending on the prompt you use and the models you choose. We are working on an implementation with llama 3 8B, but this requires a running instance of ollama and that changes the workflow, so it is not included in this project. Monitor the repository for updates.

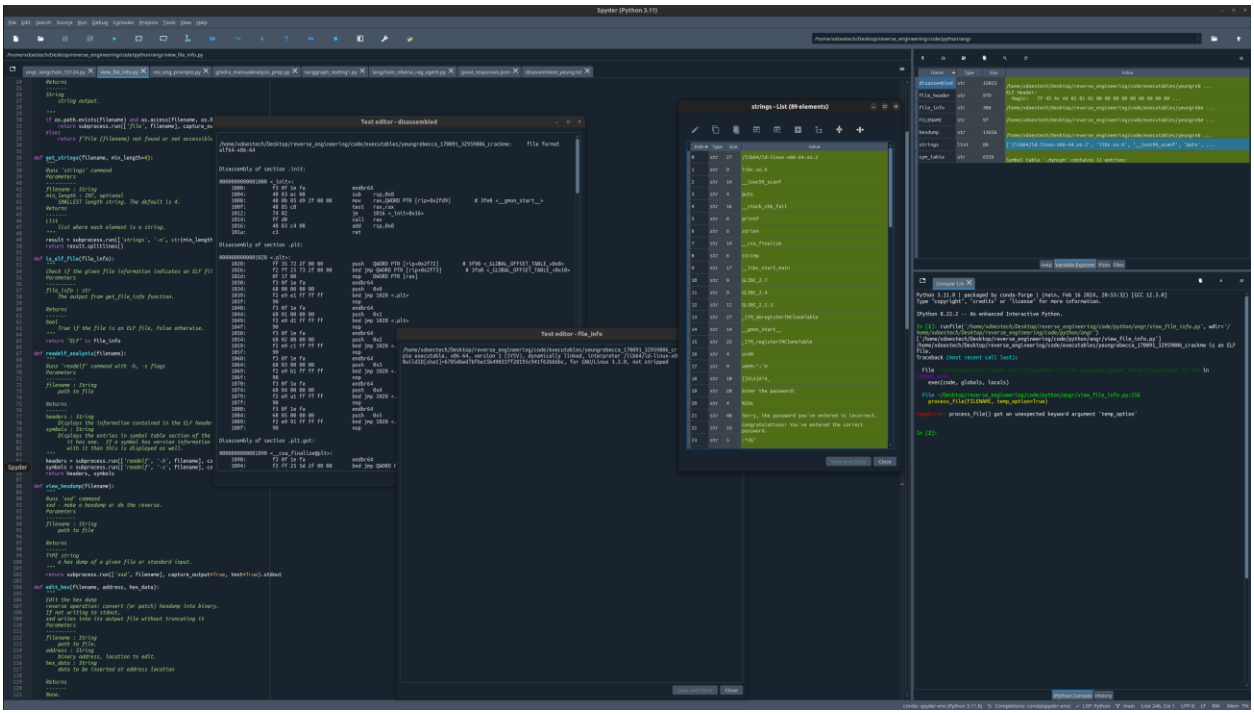
Static Analysis

As stated previously we believe that static analysis is the starting point for analyzing a binary file.

- code/python/angr/**view_file_info.py**
- Script that runs various terminal commands
    - Inputs/Variables to change:
        - FILENAME: path to file you want to analyze
        - create_project_and_save_disassembled_code(disassembled_code)
            - creates a new directory and saves disassembled_code
            - Checks to make unique directory name

We created this file by analyzing our own static analysis routines and determining the most redundant portions of the process. We recommend using the spyder IDE as it has a convenient variable explorer and iPython console.

FILE: view_file_info.py



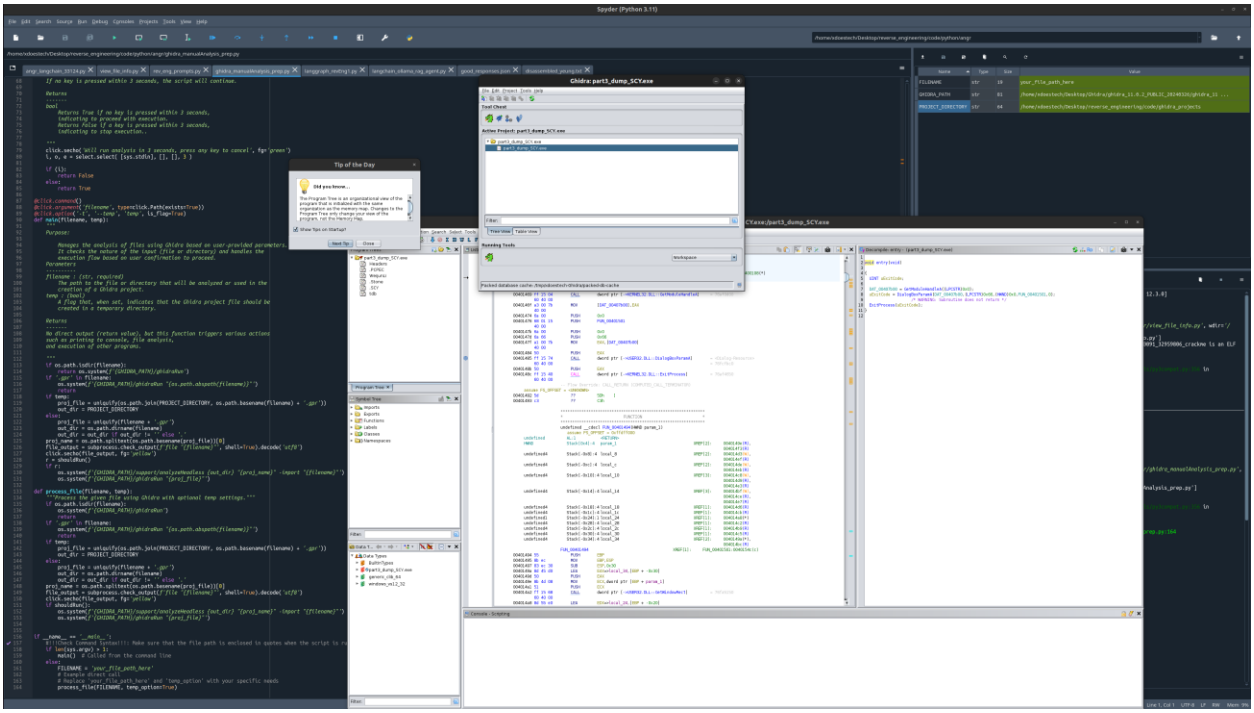
The view_file_info.py is the script that performs the automated static analysis. You should use this file as the starting point of your reverse engineering journey. Furthermore, I would recommend importing this file when using other files to have access to the extracted

values. We left this to the user since it is not guaranteed that each file will be used in succession or even on the same files at all.

- code/python/angr/**ghidra_manualAnalysis_prep.py**
- **Modified version of code created by liba2k (man of honor RIP)**
- This script is used to prepare your system to run a ghidra project
    - Inputs/Variables to change:
        - PROJECT_DIRECTORY: directory that will be analyzed or used in the creation of a Ghidra project.
        - GHIDRA_PATH: path to ghidra installation
        - should be in ~/.bashrc file
        - SEE: code/python/angr/ghidra_setup_ubuntu.mkd
        - FILENAME: path to file you want to open in ghidra

We find that starting ghidra in this fashion is simpler than opening ghidra manually. It also has a helper function (uniquify()) that creates a unique filename, this is useful for many applications. This code can be called from the command line with ‘file name’ and ‘temp’ as arguments.

FILE: ghidra_manualAnalysis_prep.py



The ghidra_manualAnalysis_prep.py file sets up your system to run ghidra. This file is not

required to perform static analysis, but it is a convenient file to run if you think you are going to run ghidra at any point in the process.

LLM Analysis

- code/python/langchain/langchain_autoAnal.py
- uses chatGPT to provide a summary of the code's functionality
    - Inputs/Variables to change:
        - DIRECTORY_PATH: path to directory containg decompiled code
        - DISASSEMBLED_FILE_PATH: path to disassembled file
        - OPENAI_API_TOKEN
        - QUERY: question you would like model to focus on during analysis
        - Limitations: only runs and queries model once, chain implementation needed

This is a basic chain that uses multiple prompts to automatically provide opinions about the file's contents. We believe that much more work is to be done here, but using this script will provide you with a second opinion about the contents of an unpacked binary file. We suggest running this after performing some initial manual analysis so you can verify the quality of the response. We have not observed any obvious hallucination when using gpt4 or gpt3.5. We do notice that gpt3.5 provides a shorter response and does not give as strong of an analysis.

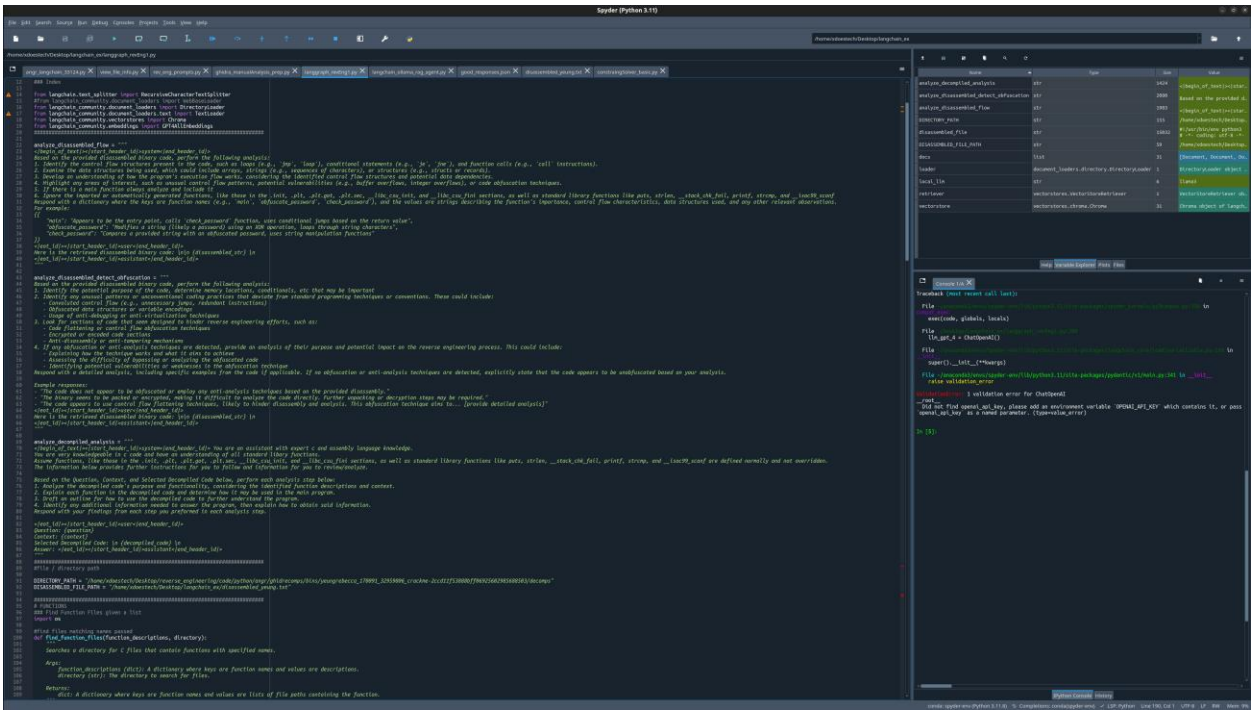
There are three sections of this file where you can define a model:

1. Disassembled Analysis
    a. Here we find that it is important to use the most powerful model available, this section analyzes the initial disassembled code you may have gotten from the view_file_info.py script.
    b. This section asks the selected llm to identify/determine the overall purpose of the code. We provide a sample prompt, but we recommend editing the 'Example Responses' Section of the code to have examples specific to the type of task you are expecting to analyze.
2. Function Selection
    a. Here we find that gpt3.5 offers comparable results to gpt4. This section asks the selected llm to identify functions of importance. We find that gpt3.5 is better at following instructions without adding its own 'fluff'. This task is basic code exploration of a language that has existed for a long time. We find that gpt3.5 is sufficiently trained to perform this task and will follow the format more closely.

b. We take these selected functions and find their decompiled equivalent. We suggest using the decompiled function available in view_file_info.py

3. Decompilation Analysis
    a. As with the disassembled analysis, we find that gpt3.5 is not capable of providing an answer with the same quality as gpt4.
    b. This section asks the selected llm to analyze the decompiled c files passed to it. Along with the results of the the other sections and provides a final analysis of the binary file.

FILE: langgraph_revEng1.py



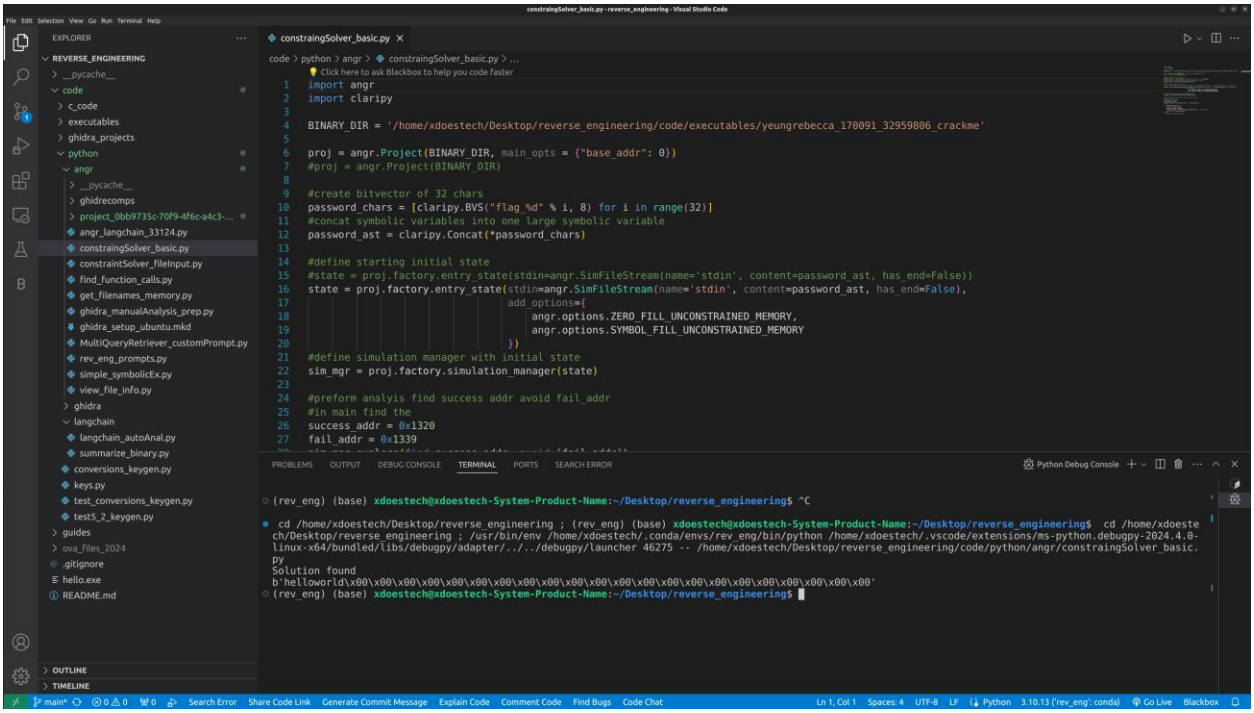
The langgraph_revEng1.py file is the starting point for beginning to use LLM's to perform analysis of an executable. There are a few prompts in the file, but they are easily modified to fit your use case. We use langchain to package everything together, but openAI's api alone is effective for this use case. We did not implement a graph or any other form of 'meta prompting' https://meta-prompting.github.io/ with looping. This is currently in process so an update may appear on the github in the near future. Unfortunately, this means that the code does not run automatically updating its own prompts and only performs one initial chain of analysis.

Angr Analysis

- code/python/angr/**constraingSolver_basic.py**

- Given a simple program, finds a value that reaches the success_addr and avoids the fail_addr
    - Inputs/Variables to change:
        - BINARY_DIR: path to binary to analyze
        - success_addr (claripy.BVS): address to reach
        - fail_addr (list): address(s) to avoid
        - Limitations: only works to find values

FILE: constraintSolver_basic.py



The constraintSolver_basic.py file performs basic constraint solving using angr. It requires knowledge about the program you are inspecting; I suggest you run this file last after developing a good understanding of it. You will need to know what specific memory address you want to visit and what or which memory address or addresses you want to avoid. This file only works when that information is known and the value you want to find is representable as a string. We are working on an implementation that works for any constraints but were unable to finish.

Conclusion

In conclusion, this project aimed to develop innovative tools and techniques to streamline and enhance the software reverse engineering process. By leveraging automation, static analysis, and a centralized platform, we sought to reduce redundancy, increase efficiency, and provide a comprehensive solution tailored to the needs of reverse engineers.

Throughout the project, we extensively explored various aspects of reverse engineering, including file identification, header analysis, string extraction, hexadecimal viewing and editing, and disassembly. We automated these processes using Python scripts, ensuring a consistent and efficient workflow, and integrated various command-line tools and frameworks, such as langchain, angr, and Ghidra.

While we encountered several challenges, particularly in developing a fully autonomous pipeline capable of reliably performing a consistent series of tasks from the input binary to the final analysis or constraint solving stage, we gained valuable insights and identified areas for future improvement. The current thinking processes of large language models, while powerful, are not yet reliable enough to be solely depended upon for such complex tasks.

Moving forward, future work could involve defining a more rigorous tool set and developing specialized autonomous agents or pipelines tailored for specific reverse engineering tasks. Integrating advanced techniques like symbolic execution, taint analysis, and vulnerability detection could further enhance the capabilities of the developed tools. Additionally, exploring ways to leverage large language models more effectively, potentially through frameworks like langchain and langgraph, could facilitate the development of more sophisticated analysis and problem-solving methods.

Despite the challenges, this project has laid a solid foundation for further exploration and development in software reverse engineering. By continuously refining our approaches, incorporating new techniques, and leveraging emerging technologies, we can create more powerful and efficient tools to aid reverse engineers in their critical work of understanding and analyzing software systems.

Papers:

Symbolic execution IBM: https://dl.acm.org/doi/pdf/10.1145/360248.360252

(State of) The Art of War: Offensive Techniques in Binary Analysis (angr explained): https://sites.cs.ucsb.edu/~vigna/publications/2016_SP_angrSoK.pdf

Report on General Problem Solving (1958 GPS-I, inspired Tree-Of-Thoughts, Algorithm-Of-Thoughts, Graph-Of-Thoughts, and describes a computer program that simulates human

problem solving): http://bitsavers.informatik.uni-stuttgart.de/pdf/rand/ipl/P-1584_Report_On_A_General_Problem-Solving_Program_Feb59.pdf