UCLouvain

École polytechnique de Louvain

Enhancing decompiled Ghidra variable names using machine learning

Author: **Yente Heus**
Supervisor: **Ramin Sadre**
Readers: **Charles-Henry Bertrand Van Ouytsel, Kim Mens**
Academic year 2023–2024
Master en cybersécurité, à finalité Conception et Analyse de Systèmes

Abstract

In this paper, we implement a Ghidra extension that improves the names assigned to decompiled variables. This extension will be based on a state-of-the-art machine learning technique for text input and output problems, called the "Sequence-to-Sequence Encoder-Decoder Transformer Model", described and introduced in the paper "Attention is all you need" [22].

Before we can train the model, we have to decide what data to use. In this paper, we collect some of the most popular GitHub repositories based on the language C. From each of these repositories, we generate two binaries, one with debug information, and one without. This allows us to generate a dataset for supervised machine learning.
Our model is also designed to make it simple to retrain on your own dataset. As a result, a security researcher can, for example, generate his own dataset comprised of French source code using our tools, and predict French variable names as a result.

Last but not least, we implement and use a custom scoring metric that is able to take into account some of the problems associated with comparing text in natural language, such as synonyms, abbreviations, word order, etc.

Code repository:
https://gitlab.com/thesis-cybersecurity-ulb/ghidra-variable-name-predictor

Chapter 1

Introduction

As a security researcher, you need many different tools to cater to the numerous situations you can find yourself in. One of the most popular and useful tools is a decompiler. It allows someone to get an idea of what the original source code of a compiled binary might have looked like. Most decompilers only generate C-like source code, even tho the original application might have been written in a different language. However, generating the original source code is not the goal of a decompiler and is often not even possible. The goal of a decompiler is to make the assembly instructions that a program is composed of more readable.

One major problem that plagues decompilers is that multiple source code texts can compile to the same assembly instructions, and by extension, a single set of assembly instructions can be decompiled to multiple source code texts. Therefore, the code shown by the decompiler might not look like the original source code. One issue that makes this problem even worse is the fact that a computer has no need for text like variable names and is therefore often removed from the compiled binary. This makes it really difficult for a decompiler to generate meaningful variable names, since the quality of the name depends on its context. This is a task that is relatively easy for humans, but really difficult for computers because there is no easy list of rules that can be followed to generate meaningful names. Because of this, decompilers will often generate meaningless names such as “param_1” or “iVar1”. This makes it a lot harder to interpret what a piece of code is doing and therefore understand the inner workings of the decompiled binary.

In this paper, we propose a method to generate more meaningful variable names using machine learning to aid a security researcher in his quest to understand what a compiled binary does and how it achieves it.

Furthermore, we will try to make our implementation as simple as possible, such

that it can be easily adapted to a person's needs.
For example, we want it to be easy to change the variable naming conventions or the language the variable names are generated in.
This makes it so that our solution is very general, and can be used by most professionals.

Chapter 2

Related work and background

Before we even think of a way to design a Ghidra plugin that generates meaningful variable names, we first have to look at the available literature and research. This allows us to make sure we don't repeat something that has been done before, and that we have a good understanding of the best ways to solve similar, related problems.

The literature is mainly split into three distinct sections: generating variable names, generating function names, and deobfuscating code.

2.1 Variable names

The most interesting of the three sections is the generation of variable names. In this subsection, we will describe some of the most interesting work related to this topic.

DIRE [14] is an Encoder-Decoder [22] machine learning model that predicts variable names based on the decompiled code from Hex-Rays.
The training data was built from a large set of C source code mined from GitHub where each entry is composed of the tokenized decompiled code, the abstract syntax tree (AST), and a lookup table mapping the generated variable IDs to the variable names generated by the decompiler and those chosen by the developers. Both input and output use the tokenized code and AST, but only the input uses the decompiler-generated names, while the output uses the developers' chosen names. Each variable name is replaced by a placeholder and for each placeholder the model is asked to predict a name.
The Encoder-Decoder model uses two encoders, one lexical encoder to obtain infor-

mation from the decompiled code, and one structural encoder to obtain information from the AST.

DIRTY [4] is also based on an Encoder-Decoder (implemented as a Transformer) model, but, aside from predicting variable names based on the decompiled code, it also generates data type names and their layouts.
The dataset consists of a large set of decompiled C-code tokens. These tokens are then encoded to contextualized representations, after which pooling is used to summarize all the variable appearances. After the decoder predicts a type, the data layout of the variable is used to generate a mask, which in turn is used to choose the most probable prediction.

Both of these models don't work with Ghidra, and are heavily modified versions of the Encoder-Decoder model.
For our implementation, we would like to keep the model as simple as possible to make it easier to understand, easier to adapt to personal needs, and possibly show that the problem doesn't require a very complex solution.
DIRTY outperforms DIRE slightly with 42.8% of the predictions being an exact match for the validation set and 92.6% for the training set versus 35.3% for the validation set and 85.5% for the training set.
They make a very important observation, however. They score the models both on data seen and not seen during training. Usually, in machine learning, one never tests a model on data seen during training, however, in this case, they argue that the score on previously seen data can indicate how well a model would be able to predict the variable names of common libraries since we are likely to encounter the same library code in the future.

In the paper KCKL [3] they tried implementing DIRTY to work with Ghidra and were successful. Although it was successful, they only studied the type prediction performance of the model. While this proves it is possible to replace Hex-Rays with Ghidra, it doesn't prove it will be good at variable name prediction.

Other papers about variable name prediction include VarBERT [1] and AJECB [11].

VarBERT is also based on a Transformer model and performs significantly better than DIRE and DIRTY with 84.15% of the predictions being an exact match for the validation set and 89.9% for the training set.
They used the dataset of DIRE, consisting of preprocessed functions decompiled using Hex-Rays.
VarBERT uses Masked Language Modeling. This means the occurrences of the

variable name are replaced with a <MASK> token and the model is then asked to predict a name for these tokens.

AJECB is based on SMT or statistical machine translation [13], but performs worse compared to DIRE, DIRTY, and VarBERT with a maximum of 28.7% of the predictions being an exact match for the validation set (no precision score on the training set was provided).
Each entry in their dataset is preprocessed with a hash-renaming optimization function and assigned a parallel corpus of aligned content. The hash-renaming function, renames every variable to something similar, while also adding additional information, such as the type, argument position, and the most informative line. This increases the chances of a similar code snippet resulting in similar predictions. The parallel corpus of aligned content added to the entry tries to align the variable names generated by Hex-Rays with those chosen by the developer, and is needed to train the SMT model.
The SMT model generates a translation for each line of the decompiled code, after which a post-processing step extracts and outputs the variable names.

Neither VarBERT nor AJECB provide an implementation for the Ghidra decompiler and is therefore still an open research question.

2.2 Function names

Some other papers focus on the prediction of function names instead of variable names. While the problem they aim to solve seems related, it is not the same. The context needed to predict a function name is not the same as the context needed to predict a variable name. For example, the function name might depend on the control flow graph and the parameters, while this is generally not the case for a single variable name.
Therefore, we can't compare their results to ours and have to remember that their way of generating and processing a dataset will most likely not work for predicting variable names.

One such paper with a focus on function name prediction is NRESB [5]. In order from least to most performant, the authors of this paper used an LSTM Encoder-Decoder, Transformer, and GNN model when predicting function names. Instead of using the outputs of a decompiler, the authors of the paper built the dataset using call sites graphs, which are based on the control-flow graphs extracted directly from the assembly symbols of the binary to be analyzed.

In CFG2VEC [24] the authors used a GNN model to predict function names based on a Graph-of-Graph representation of the binary, which is constructed using the control-flow and function-call graphs extracted by Ghidra. They also developed a Ghidra plugin allowing security researchers to easily use their model.

AsmDepictor [12] also aims to recover the original function names through the use of a Transformer model. Just like some of the other papers in this field, the dataset is built from the assembly instructions of a set of binaries.

The authors of the paper implementing Punstrip [15] used a different approach. They used Maximum a Posteriori estimation which is able to predict function names based on a Conditional Random Field generated from the function fingerprints and factor graphs which represent the relationships between functions in the binary. The Conditional Random Field essentially learns how functions interact with other code and data.

XFL [16] is based on a multi-label classifier that selects, based on Function Embeddings, the most likely function name parts from a set of parts. Function Embeddings aim to represent functions as a vector so that similar functions will have a low vector space distance and dissimilar functions a high vector space distance. Then a trigram language model is used to predict the most likely order of the predicted parts, which are in turn concatenated following a coding convention like snake_case to form the final function name.

2.3 Deobfuscating code

In this subsection, we will present related work that focuses on the deobfuscation of code.
While renaming variable names to something more meaningful can be seen as a type of deobfuscation, the general problem of deobfuscation is way more complex. For example, deobfuscating code often requires restructuring and rewriting entire code blocks.
Therefore, most of the techniques used in this field will not apply to us.

The authors of Context2Name [2] designed a technique to deobfuscate JavaScript code often found on web pages that is aimed at variable name prediction. Context2Name is based on an RNN-based predictor that outputs a ranked list of possible names for each minified variable name from a corpus of possible names. The predictions of the model are based on compactly represented usage summaries

per identifier, which are reduced into efficient vector representations called embeddings.
The problem of deobfuscating JavaScript code is different from predicting variable names for decompiled code, since JavaScript is never compiled. Therefore, the source code you start with doesn't contain any errors and is able to run, meanwhile, when the debug data is missing, the code generated by a decompiler often contains errors, datatype structures that are incorrect, or an incorrect amount of variables. This makes our problem a lot harder to solve.

2.4 Conclusion

All of these solutions use machine learning, and the most popular models are Encoder-Decoder LSTMs, RNNs, and Transformers. As of the time of writing this paper, Transformers are considered the state-of-the-art.
Since there are many factors at play and many correct solutions when trying to come up with a meaningful variable name, it is impossible to define a simple set of rules that will always generate meaningful names.
We will therefore also use machine learning and base our implementation on a Sequence-to-Sequence Encoder-Eecoder Transformer model.

All the papers described above use a different way to extract features and build their dataset. However, the data they use is almost always composed of C source code snippets or C-based binaries. Since Ghidra decompiles a binary to C-like code, we will also use binaries compiled from C source code.

Lastly, none of the above papers are focused on finding a novel way to generate meaningful variable names based on the decompiled output from Ghidra. Therefore, this paper will be the first in this field.

2.5 Transformer

The Transformer model was first introduced by the paper "Attention is all you need" [22].

A Transformer is a deep neural network that is good at solving text-in-text-out problems with the help of self-attention. Self-attention means that the Transformer will generate a new representation for one word w from a combination of the other words. This new representation of w thus now includes its context.
An example can be found in figure 2.1, and the general structure in figure 2.2.

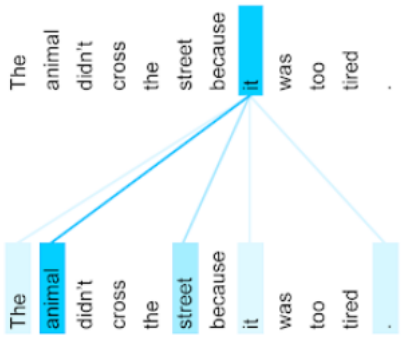

Figure 2.1: Example of the values generated by a self-attention layer. Here the layer discovered that there is a strong connection between the words "animal" and "it". (Image source: Google AI Blog [21])

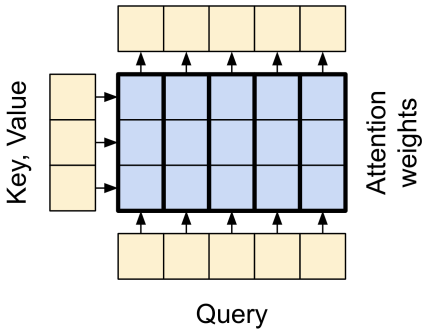

Figure 2.2: Example of the structure of an attention layer. The output is based on the sequence being processed (query), the context (key, value), and the learned attention weights. (Image source: TensorFlow [18])

At the start, the model will transform the input text into tokens, which we call tokenization. Tokens are essentially numbers representing the index of a word part inside a dictionary. For example, the word "basketball" might be transformed into the array [152, 856] where 152 is the index of the word "basket" inside the dictionary and 856 that of the word "ball".
The aim is to make the resulting array as small as possible while still trying to preserve the link between words. For the example above, we could have put the word "basketball" in the dictionary, but then there is no direct link with the words "basket" and "ball" anymore.

After the input has been tokenized, the Transformer will perform the self-attention procedure described above for every token inside the input. This is then repeated

multiple times in parallel in order to create a representation that will in turn lead to the output of the model. We call this step the "Encoding" step.

When then input has been encoded, the "Decoding" step is started.
This step is very similar to the "Encoding" step. The Transformer performs the self-attention step described above for each token it wants to output. It executes this multiple times in parallel, each time generating a new representation of the current output token. There is one difference however, the context used in this step to generate the ith representation of an output token is made out of the representation of the entire input generated by the "Encoding" step, and the ith representation of each previously generated output token.
Each decoding iteration produces one token until the "end" token is generated.

The output of the "Decoding" step is an array of tokens. Therefore, the last step is to detokenize the array to form the final output. Remember that the tokens just represent an index of a word part inside a dictionary, so we just have to replace the indexes with the actual word parts they refer to.

Since the self-attention steps in the "Decoding" step need two inputs to generate an output (see figure 2.2), namely the representation of the entire input, and the ith representation of the previous output tokens, we need to choose an initial value for the first self-attention step.
This is where we separate the training phase of the model from the prediction phase.
When the model is predicting an output, we simply initialize the input of the first self-attention step in the first decoding iteration to the "<SOS>" token, which just indicates the start of the sentence. Then, each time a token is predicted by the "Decoding" step, we feed it back to the beginning of the decoder together with the previously predicted tokens, and start the "Decoding" step again until the "<EOS>" or end of sentence token is generated.
When training the model, however, we pass the actual expected output tokens of the model one by one as the input of the first self-attention step instead of the predicted ones. This way of learning is called "teacher forcing" and is more stable.

A general simplified representation of a Transformer model can be found in figure 2.3.

2.5.1 Advantages

One advantage of the Transformer model over other similar solutions such as Recurrent Neural Networks (RNNs) is that they are parallelized. This is because

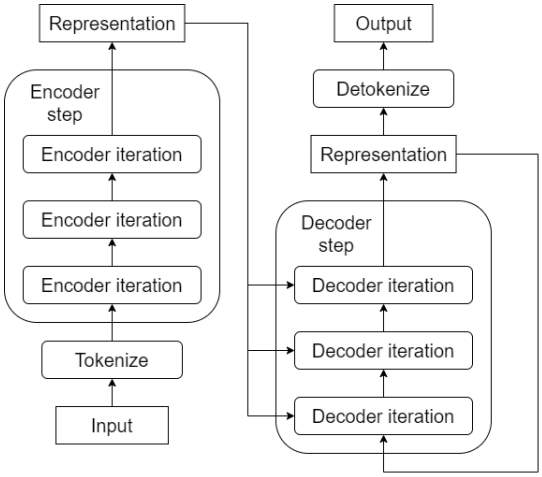

Figure 2.3: A simplified representation of the Transformer architecture as described in section 2.5

the self-attention computations can be executed in parallel.

Another advantage is that a Transformer can capture long-distance relationships between words because of the attention. At every step, the words have access to the entire input at each layer of the model.

Chapter 3

Design

In this chapter, we will explain how we implemented our solution to the variable renaming problem, and why we chose certain methods or techniques.
We will start by explaining which model we used, why we used it, and how we implemented it, followed by an explanation of how and what data we gathered, how we processed it, and how we transformed the data to build the dataset.

3.1 Overview

The workflow to get to the end solution, which is a Ghidra plugin that gives variables a more meaningful name using machine learning, looks like this:

1. Acquire training data. In our case, we scrape GitHub for C source code.
2. Compile the set of C source code into a large collection of binaries with debug data.
3. Duplicate the dataset of binaries and strip all debug data from one of them. This way we have 2 equal versions of each binary minus the debug information.
4. For each tuple of binaries that belong together, decompile the binary with and without debug information using Ghidra. Then compare them and extract all the decompiled functions without debug information, together with a mapping for each variable from the meaningless name generated by Ghidra to the original name extracted from the debug information.
5. Format and preprocess the extracted data and combine all instances into a training set, validation set, and test set.
6. Optimize the Transformer model parameters and train it using the generated dataset.

7. Write a custom variable name scoring function to assess the meaningfulness and use it, together with other more generic scoring functions, to score the model.

8. Write a plugin for Ghidra that renames all the variables of a function using the trained model.

The prediction pipeline using the trained model is displayed in figure 3.1. This model will then be used in the Ghidra plugin, of which an illustration can be found in figure 3.2.

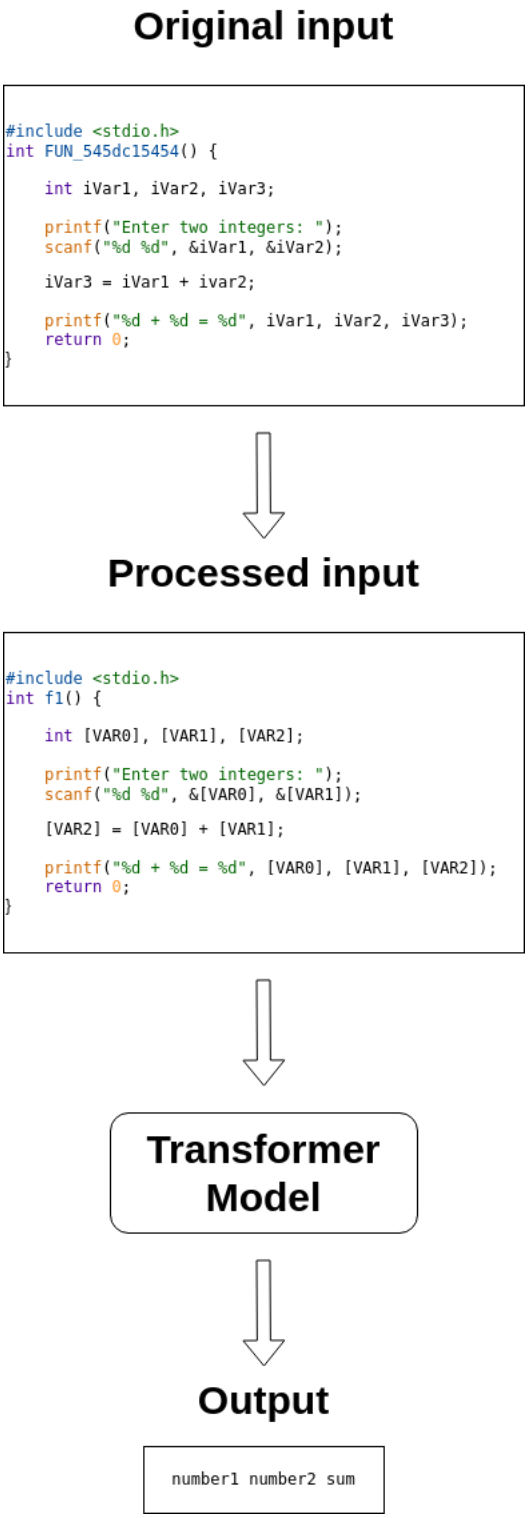

Figure 3.1: Example of what the prediction pipeline will look like.

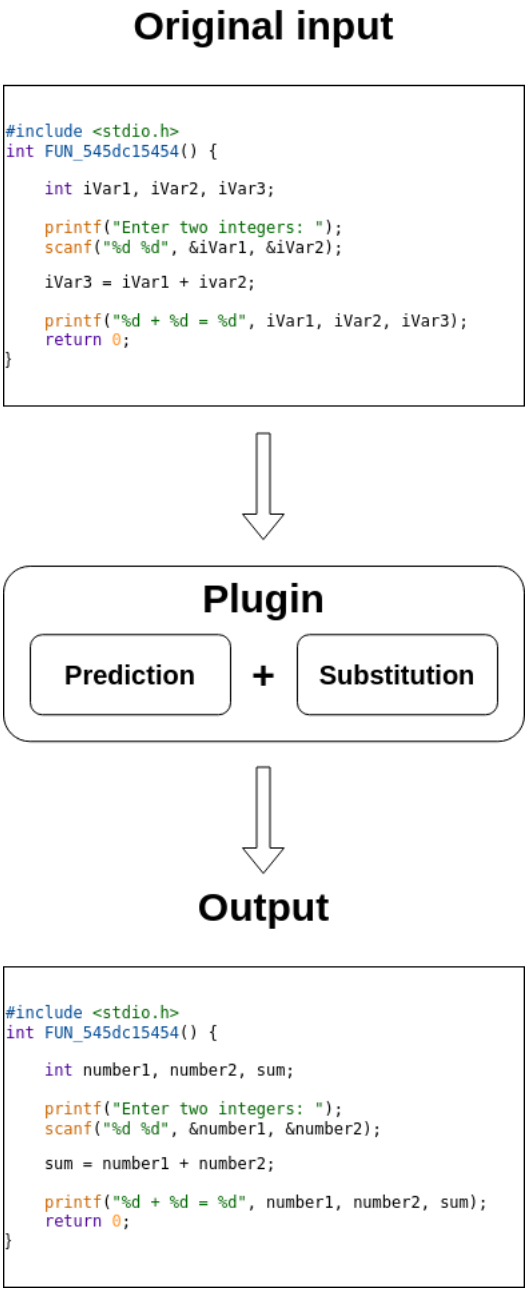

Figure 3.2: Example of what the plugin pipeline will look like.

3.2 Model

The problem of recovering meaningful variable names based on their usage is very complicated. There are way too many factors and parameters to take into account. A slight deviation in the usage of a variable might lead to the name being completely different. Some completely different names also have the same meaning and can be interchanged, such as “picture” and “photo”.
In addition, some programmers prefer to use abbreviations instead of full names such as “buffer” and “bffr” making it hard to determine whether two names are similar.

Therefore our problem can’t be reduced to the definition of a simple set of predefined rules. If we were to try to base our solution on manually defined rules, we would likely never finish, and the rules themselves would be very complicated and unreadable.

Because of this, we will base our solution on recent advancements in machine learning. As explained in chapter 2, we will use a Sequence-to-Sequence Encoder-Decoder Transformer architecture because it is state-of-the-art and very good at solving text-in-text-out problems. Since we want to generate a meaningful name based on the decompiled source code, we can describe our problem as being of the same variant.

3.2.1 Implementation

We wanted to keep the implementation of the Transformer model as simple as possible and easy to train while keeping it performant.

This would allow cybersecurity specialists to retrain the model on specific data such as a French dataset to generate French variable names, or a dataset gathered from a company’s source code library to generate more meaningful variable names in the context of the company’s coding rules and styles.
It would also allow specialists to adapt the model itself to their needs because of its simplicity and readability.
Lastly, a simple model makes it easier for us to optimize and train it, since we don’t have access to super fast machine learning hardware. All the training for this paper was performed on an RTX 4070 mobile, and GTX 1070. Both of these graphics cards only have 8GB of VRAM and therefore can’t store a very complex model without severely impacting training time.

Because of these reasons, we chose to base our model on the example provided by

TensorFlow [18]. We adapted the code where necessary for it to work with our dataset, but other than that the model is relatively unchanged.

Most of the effort in this paper was spent on gathering and processing the data, developing a meaningful scoring function for variable names to evaluate the performance of our model, and integrating the model with Ghidra as a plugin.

3.2.2 Parameters

As mentioned in section 3.2.1, we want to keep our model simple, therefore the parameters chosen reflect that of a simpler model.
We found that our simple model performed optimally with the following parameters:

Table 3.1: Model parameters used

layers	**heads**	**d_model**	**dff**	**dropout**	ϵ	β_1	β_2	**epochs**
3	4	128	512	0.1	10^{-9}	0.9	0.98	20

3.3 Dataset

In this section, we will explain how we constructed our dataset containing decompiled functions without debug information, the generic variable names generated by Ghidra, and their corresponding real names gathered from the debug information of the source code.
The generated dataset is used to train the Transformer model described above.

We will first describe what data sources were used, and how we gathered data from them.
Then, we will explain how this data was processed in order to extract decompiled functions and variable names generated by Ghidra.
Lastly, we will explain the formatting of the data to make it compatible with the model and to optimize its performance.

3.3.1 Data acquisition

Before we start extracting variables and other tokens from binaries, we need the binaries themselves. More specifically, we need two versions of each binary, one

with, and one without debug information.
To achieve this, we need access to either a bunch of binaries with debug information, and then strip the debug symbols, or to the source code from a bunch of projects, and compile them ourselves.
Because it is very difficult to find a large set of binaries that were compiled for the same architecture with the same parameters and debug information, we chose to go with the second method.

It is relatively easy to get a hold of a large amount of compilable source code. There a many places that host the source code of open-source projects online. Among these are GitHub [6] and GitLab [8] which are the two most popular ones.
Because GitHub is the most popular of the two according to Google Trends [9], we chose to select projects from there.
With its popularity, the projects are more likely to be of higher quality, and will, in turn, likely result in a higher-quality model that converges faster and makes better predictions.

The authors of DIRTY [4] used GHCC (GitHub Cloner & Compiler) [10] to generate their dataset.
GHCC is a tool that clones all the repositories on a specified list and tries to compile the code inside.
Because it simplifies and accelerates the task of compiling all the projects, we decided to use the same tool.
However, due to the age of its last update, we encountered a few issues.

- The last version of argtyped is not supported, therefore we replaced "argtyped" in "requirements.txt" with "argtyped==0.2.0.post0". At the time of the last update, the most current version of argtyped most likely worked. This kind of error is to be expected when using older software.

- The keyserver specified in "ghcc/Dockerfile" is no longer available. To fix this, we replaced "–keyserver keyserver.insect.com" with "–keyserver hkps://keys.openpgp.org". Just like the previous issue, this is very common with older software.

By default, GHCC hashes the file contents of a compiled binary and uses the resulting hash as its filename. This makes it hard to identify which source code file was responsible for a certain binary, and in turn, identify which two files (one with debug information and one without) contain the same code.
We changed the naming function "_hash_file_sha256" in "ghcc/ghcc/compile.py" to return the name of the original source code file instead.

To select the repositories on which we will base our dataset, we used the GitHub API [7]. It allows you to search through all the projects based on a set of criteria and returns the first 1000 matching projects in a specified order.
Since the API limits us to the first 1000 projects, the size of our dataset will also be limited. In our case, this isn't an issue, since we are also limited when it comes to time and resources. Even if we had access to more data, we wouldn't have been able to process it all.

We selected the first 1000 public projects written in C, with a size smaller than or equal to 100 MB, ordered by the amount of stars received.
On GitHub, you can mark a project as a favorite by giving it a "star". By using the stars received to order the projects, we essentially select the most popular ones. Generally, the more popular a project is, the more people contribute, and in turn the higher the quality of the code.

The last step in gathering the data is to generate the binaries. We ran GHCC on the list of projects we gathered using the methods described above to clone and compile them all.
To make sure we retain all the debug data, we specified the following parameters: "–gcc-override-flags "-g -gdwarf-4"".
By default, GHCC compiles the binaries without any optimizations (optimization level 0). This is what we want, since most optimizations will lead to information loss.
Training our model on a higher optimization level would be an interesting expansion to our paper, and would make for fascinating future work.

Now that we have a large set of binaries with debug information, we just copy them to a separate folder and strip all the debug symbols.
Stripping the binaries instead of recompiling them without the debug parameters will ensure the binaries are as similar as possible.

Now we are left with two sets of binaries, both based on the same source code, one with and one without debug information.

3.3.2 Processing

Now that we have a large set of binaries, we need to extract Ghidra-specific information.
We could try using the binaries directly as input to our Transformer model, bypassing Ghidra, but this would likely result in a model that performs really poorly since we are not helping it by extracting important features.

Therefore, we parse and process each of the binaries using Ghidra and extract information that would also be available when running the Ghidra GUI application directly.

Ghidra provides a console-only version that directly executes Python scripts before or after the analysis of the binary and provides the intermediate state as global variables.
We can for example run "ghidra.analyzeHeadless <project_location> <project_name> -import <binary_name> -postscript <script_name>"
This will analyze the binary with the name <binary_name> after which it will execute the Python script called <script_name> and provide the results of the analysis to the script.

For example, in the script we can access the analyzed program through the global variable "currentProgram". To decompile the functions inside the binary, we would have to do something like this:

```
program = currentProgram

decompinterface = DecompInterface()
decompinterface.openProgram(program)

functions = program.getFunctionManager().getFunctions(True)
```

Then for each function, we would have to do the following:

```
decompileResults =
    decompinterface.decompileFunction(
        function,
        0,
        ConsoleTaskMonitor()
    )

decompiledCode =
    decompileResults.getDecompiledFunction().getC()
```

We left out the import statements for clarification.

This only leaves us with the decompiled C-like code which can be used as an input to the Transformer model and is a great improvement over using just the binaries as input.
However, Ghidra provides access to a lot more metadata than just the decompiled code.
We can, for example:

- get the name of a function

- get the address of a function
- get the prototype of a function (return type and arguments)
- get a list of all the variables (local and global) together with their name (useful if debug data is available, otherwise generic), datatype, size, and location (address)

After parsing a binary using the Ghidra headless analyzer, we return a dictionary where the keys are the addresses of all the analyzed functions. Each value in this dictionary is a dictionary itself with the following keys:

- **Name**: the name of the function
- **Code**: the decompiled C-like code
- **Params**: array of parameters with the following structure:
  [[name, data type, size], ...]
- **Ret**: the return data type of the function with the following structure:
  [data type, size, has no return]
- **LocalVars**: a dictionary of the local variables of the function where the keys are the locations (addresses) of the variables, and the values have the following structure:
  [name, data type, size]
- **GlobalVars**: same as the "LocalVars" key, but for the global variables.

The locations (or addresses) of functions and variables can be used to compare them. For example, if we parse two binaries, one with and one without debug information, we can find the functions that are the same by looking at the addresses. The same holds for the variables local and global to the functions.
We can't use their names, since the binary without debug data will produce functions and variables with generic names instead of the original names produced by the one with debug data.
Therefore, the extracted names will be different and can't be used to compare functions and variables.

The research and development time to get to this point constitutes a significant part of this thesis, since the documentation of the Ghidra API is less than ideal. There is also not a lot of information available online since not a lot of people write plugins for Ghidra.
Another issue we encountered is that to facilitate the development of plugins for

the Ghidra GUI application, the documentation was written for Java instead of Python, hindering the development of the parsing script. All the Java imports, objects, and functions described in the documentation can be imported and used in Python, but most of the data types are translated to Python alternatives, which makes it even more confusing.
In the end, we figured out how to extract a lot of useful information from a binary by trial and error.

To process all the binaries gathered using GHCC in section 3.3.1, we wrote a parallelized Python script that executes the Ghidra headless analyzer with the parsing script described above for each binary. The script divides the workload over all the available system threads to increase its efficiency.
The parallelization is necessary because decompiling a binary is very resource-intensive and takes a long time.

When the parsing and comparing of a binary pair (one with debug information and one without) is finished, the results are saved and written to a file. This way, we only have to parse every binary pair once, and the results can just be reloaded whenever they are needed.

3.3.3 Formatting

Now that we have a large collection of processed binaries containing relevant information extracted using the Ghidra headless analyzer, we can start formatting them to make it easier for our model to learn.
Theoretically, we could just turn all the collected data from a binary into a string and feed it directly into the model.
This is exactly what we tried in the beginning to see if our idea of predicting meaningful variable names was even possible. We concatenated the generic variable name in question, metadata (such as the decompiled function name, parameters, local/global variables, and the return signature), and the decompiled code (truncated to keep the input size within the predefined model input limits). A visualization of the input structure can be found in figure 3.3.

One of the main issues with this process is that the actual variable we are trying to predict a name for might be inside the part of the decompiled code that was truncated, making it almost impossible for the model to make a good prediction.
Since testing takes pretty long and this was never meant to be the final result, we don't have any performance figures, but the way the model performed using this way of formatting the input left a lot to be desired. It was able to make some very rudimentary predictions that were often incorrect. With this level of performance,

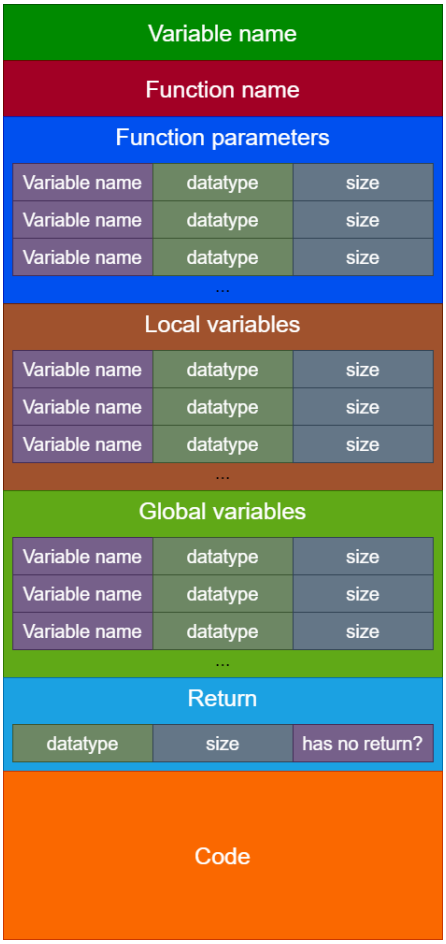

Figure 3.3: A visual representation of the first input formatting structure.

the model would not have been helpful to a security researcher and would have probably been a hindrance instead.

We tried to optimize this simple way of formatting the input data by:

- Removing the metadata including the decompiled function name, parameters, local/global variables, and the return signature since all this information can be extracted from the decompiled code. In reality, some of this information might be lost due to the truncation of the code.

  Result:

  - Performance of the model improved but still left a lot to be desired.

- Marking each occurrence of the variable name with a special token instead of putting its name in the beginning as shown in figure 3.3. This way, the token indicating the variable we are looking for is universal.

  Result:

  - No performance improvement.

- Splitting the decompiled code into fragments around the variable for which we want to predict a more meaningful name instead of adding the entire (truncated) text, forcing the model to only look at the parts of the code around the variable in question.
  However, this might omit some important and meaningful parts of the code. Not all the important information that can help indicate what a variable is used for or what it might contain is located close to the variable itself.

  For example, the following code snippet defines a structure that contains information about a point in 3D space. It also defines two functions, one to initialize such a point to its home location, and another to print it. If we look at the main function, it just declares a 3D point, initializes it, and prints it using the aforementioned functions.

```
#include <stdio.h>
#include <stdint.h>

#define HOME_X 128
#define HOME_Y 128
#define HOME_Z 128

typedef struct {
    uint8_t x;
    uint8_t y;
    uint8_t z;
} Point3D;

void initPoint3D(Point3D *p) {
    p->x = HOME_X;
    p->y = HOME_Y;
    p->z = HOME_Z;
}

void printPoint3D(Point3D *p) {
    printf(
        "3D point at location (X:%u, Y:%u, Z:%u)\n",
        p->x,
```

```
        p->y,
        p->z
    );
}

int main(int argc, char* argv) {

    Point3D *point;
    initPoint3D(point);
    printPoint3D(point);

    return 0;
}
```

If we were to only use small code fragments around the variable " Point3D *point;" the input of the model might only contain the following:

```
Point3D *point;
initPoint3D(point);
printPoint3D(point);
```

Which, without debug information, would look something like this:

```
undefined3 *point; // undefined3 means variable with a length
    of 3 bytes
f1(point);
f2(point);
```

As a result, all the important information indicating the original variable was a point in 3D space is lost, and therefore the model will have no information to base its prediction on.

With a large enough input size, we should be able to cover a much larger area around the variable, and these edge cases should become very rare, or at least less severe.

Result:

- Tiny improvement in performance.
- The variable we are trying to predict a more meaningful name for won't be cut off, which might be the case when truncating the decompiled code.

- Cleaning up the decompiled code by removing unnecessary parts such as comments, spaces, enters, etc. while keeping it legal C code. The comments

we encounter don't carry any important information, since the original comments written by the developers are not recovered. The comments we are removing are ones generated by Ghidra with extra information about the decompilation process.

The idea is to make the code as compact as possible so that we can fit more inside the input without losing any information.

For example, the following code snippet

```c
#include<stdio.h>

int f1(int var1){
    return var1 * 2; // This is a comment
}

int main() {
    int x = f1(2);
    printf("Result: %d\n", x);
}
```

gets transformed into:

```c
#include<stdio.h>
int f1(int var1){return var1*2;}int main(){int x=f1(2);printf
   ("Result: %d\n",x);}
```

Result:

  – Tiny improvement in performance.

- Firstly, only providing unknown variable names (such as "iVar1", "param1", ...) as input to the model, and just return the original variable name generated by Ghidra if it was able to recover or deduce it.
  Sometimes Ghidra can successfully recover variable names even if there is no debug data present. For example, if it is analyzing a known library or piece of code.
  This way, we can offload some of the stress from the model, and increase the overall accuracy.

  Secondly, removing dataset entries where the ground truth output generated by Ghidra using the binaries with debug information is the same as an unknown variable (such as "iVar1", "param1", ...). This happens when the compiler decides to introduce an extra variable. Since these variables

are not introduced by the developer, there is no corresponding debug information available, and Ghidra can therefore not recover their types and names.

Result:

  - Slight improvement in performance.

- Replacing the decompiled code fragments with just the function names that occur around the variable. Function names generally aren't recovered by Ghidra and won't help much because there is no debug information available for the input. However, it can recover the names of functions that are part of a standard library (such as "printf", "malloc", etc.). These types of functions can say a lot about what a variable is used for.

  For example, if we see the following functions in order: malloc, strcpy, printf, free, we might assume the variable is a buffer holding a string.
  Of course, this assumption can be completely incorrect because a lot of information is being lost. The strcpy function might, for example, not even interact with the variable we are investigating at all, and the printf function could print something completely different, such as a variable holding an integer.

  Result:

  - No improvement in performance.

- Instead of just looking at the textual representation of the parsed binaries and extracting snippets from it, we generate an abstract syntax tree or AST from the decompiled code. This is possible since the decompiled code generated by Ghidra is written in the low-level language C.
  This allows us to parse the code in a more complex way that doesn't introduce as many bugs because of the numerous edge cases that often go hand in hand with source code parsing.

  As an example, the following code snippet

```
1 int plus(int x, int y) {
2     return x + y;
3 }
4 void main() {
5     plus(1, 2);
6     return 0;
7 }
```

would generate the abstract syntax tree (AST) shown in figure 3.4.

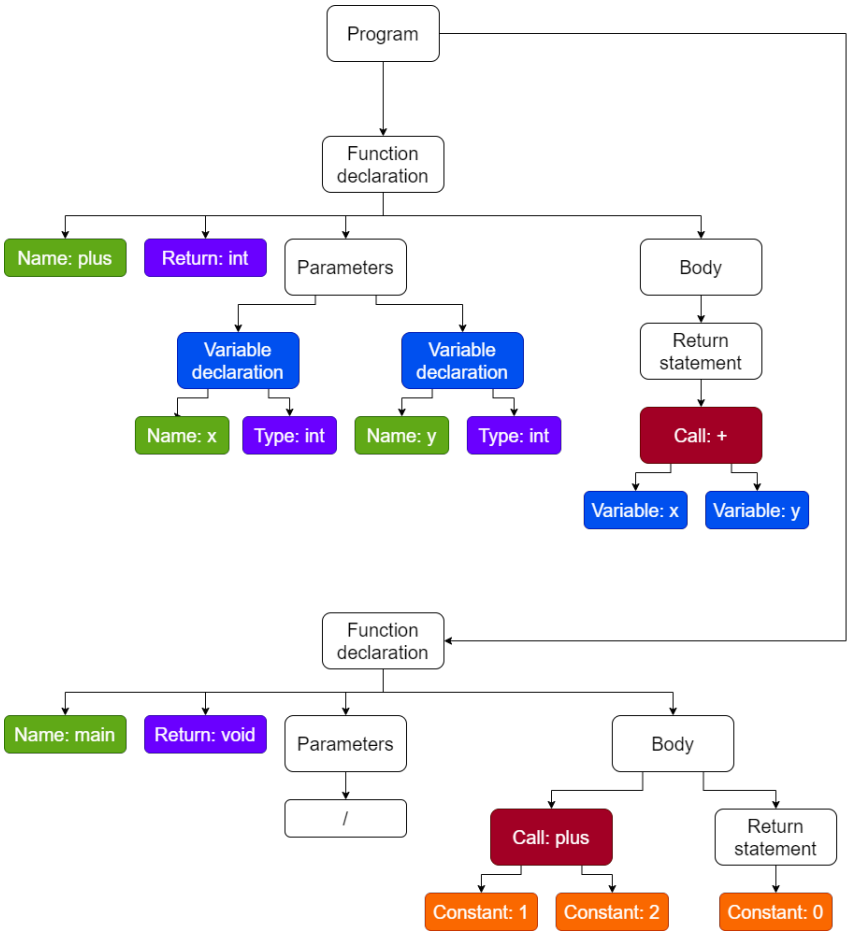

Figure 3.4: An example of an abstract syntax tree (simplified for clarity).

After generating the abstract syntax tree (AST), we walk through the tree using depth-first traversal. Whenever a path leads to the variable we are processing, we add all its interesting nodes to a list of strings. In the end, duplicates are removed, and these strings are concatenated together with the generic decompiled name of the variable in question at the beginning.

The procedure for collecting these interesting nodes looks like this in pseudocode:

```
1 interesting_strings = []
2
```

```
search(AST, var_id):
    if (AST root is name node): return name == var_id

    found = false
    if (AST root is declaration node && declares our var_id)
        found = true

    // Depth-first traversal
    for each sub_AST in AST:
        found = search(sub_ast, var_id) or found

    node = AST root
    if (found &&
        node is interesting &&
        not all node info in interesting_strings
        )
        remove all strings in interesting_strings that are
   part of node info
        interesting_strings.append(node info)

    return found

collect_interesting_parts(AST, var_id):
    search(AST, var_id)
    return interesting_strings
```

The resulting format of the processed input can be found in figure 3.5.

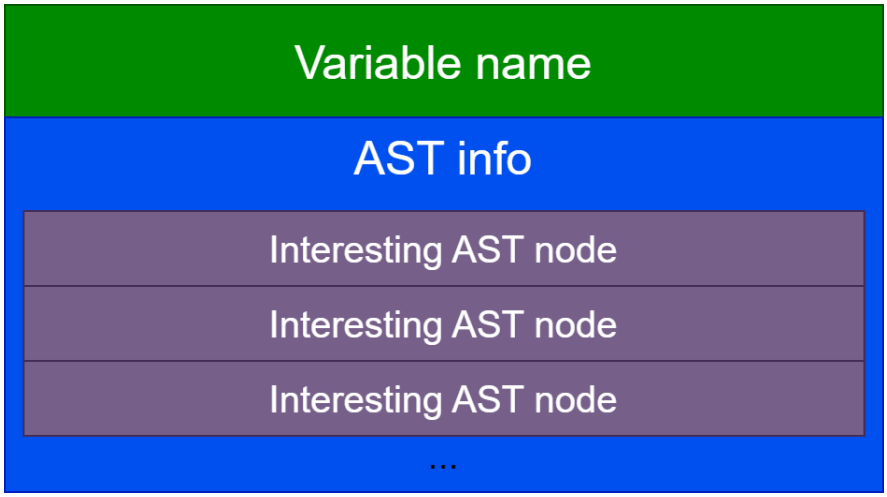

Figure 3.5: Format of the input using interesting nodes extracted from the AST.

The following example was generated using our procedure while focusing on the variable “local_9”. It got the following code snippet as input:

```
ulong f1(undefined8 param_1,undefined2 param_2,undefined2
    param_3,undefined2 param_4,undefined8 *param_5)

{
  undefined8 uVar1;
  char cVar2;
  ulong uVar3;
  long local_18;
  byte local_9;

  local_9 = f2(param_2,&local_18);
  if (local_9 == 0) {
    cVar2 = f3(local_18);
    if (cVar2 == '\x01') {
      *(undefined8 *)(local_18 + 0x10) = param_1;
      *(undefined2 *)(local_18 + 0x4a) = param_3;
      *(undefined2 *)(local_18 + 0x4c) = param_4;
      *(undefined2 *)(local_18 + 0x4e) = param_3;
      *(undefined2 *)(local_18 + 0x50) = param_4;
      *(undefined2 *)(local_18 + 0x38) = 0;
      uVar1 = param_5[1];
      *(undefined8 *)(local_18 + 0x3a) = *param_5;
      *(undefined8 *)(local_18 + 0x42) = uVar1;
      *(undefined4 *)(local_18 + 8) = 0x13;
      f4();
      uVar3 = 0;
    }
    else {
      uVar3 = 0xffffff95;
    }
  }
  else {
    uVar3 = (ulong)local_9;
  }
  return uVar3;
}
```

It produced the following output that would be used as the input to the model:

```
local_9
byte local_9
local_9 = f2(param_2, &local_18)
uVar3 = (ulong) local_9
if (local_9 == 0)
```

As you can see, only the parts that interact with the variable have been preserved. This makes the input very compact and forces the model to only look at the most important parts. It is true that some contextual information

might be lost, since the interaction between different statements that don't directly influence the variable might still hold useful information. However, the statements that interact directly with the variable in question will, in general, be more informative.

Result:

- Slight improvement in performance.

After all of these optimizations and experiments, the model still didn't perform that well.
A model that only gets an accuracy score of 20% because it's only able to run on 20% of the variables, and just ignores the rest, is perfectly usable. In this scenario, the security researcher would have to deduce the names of 20% fewer variables.
But a model that runs on all the variables and only has an accuracy of 20% is completely unusable since 80% of the predictions are incorrect, and would be misleading, making reverse engineering even harder.

From our results, we saw that the model wasn't good at generalizing. It performed well on the training set, but very badly on the validation and test sets. Therefore, we changed the way we processed the data drastically in an attempt to increase the performance of the model and make it usable.

Since the variable names are often related, for example, the integer after a buffer is probably its length, we switched to predicting all the variable names for a function at once, instead of formatting the input and output on a variable-by-variable basis. As the input, we now provide the model with the decompiled code where all the variables have been replaced with a special masking symbol, while still keeping the variables separate (each variable gets its own symbol, e.g. [VAR1], [VAR2], etc.). Whenever a variable has no debug information, its corresponding ground truth is marked with a special "unknown" symbol instead of the name assigned by the developer.

Secondly, since Ghidra names functions based on their addresses in the binary (such as "FUN_564ade654), they are relatively unique. The model can just learn to reproduce the same variable names based on the unique function names, since the probability of there being multiple functions with the same name is relatively low. The same goes for labels and certain global variables.
Because these names don't hold any useful information, we rename them to something generic, such as "fX" for functions, "lX" for labels, and "dX" for global variables.

We call this “function anonymization”, since it essentially removes all the information that can be used for function identification. Obviously, it’s not true anonymization, the strings are still left because they hold a lot of information but are often unique, and the same goes for the combination of all the function statements, but it forces the model to take into account the entire function instead of just a few names.
Doing this leads to a huge increase in the model’s ability to generalize.

Lastly, we make sure every variable name in the output uses the camelCase convention. We also remove all digits from the front and the back, since they are generally only added when the same name occurs multiple times.

A visualization of what the dataset entries look like can be found in figure 3.6.

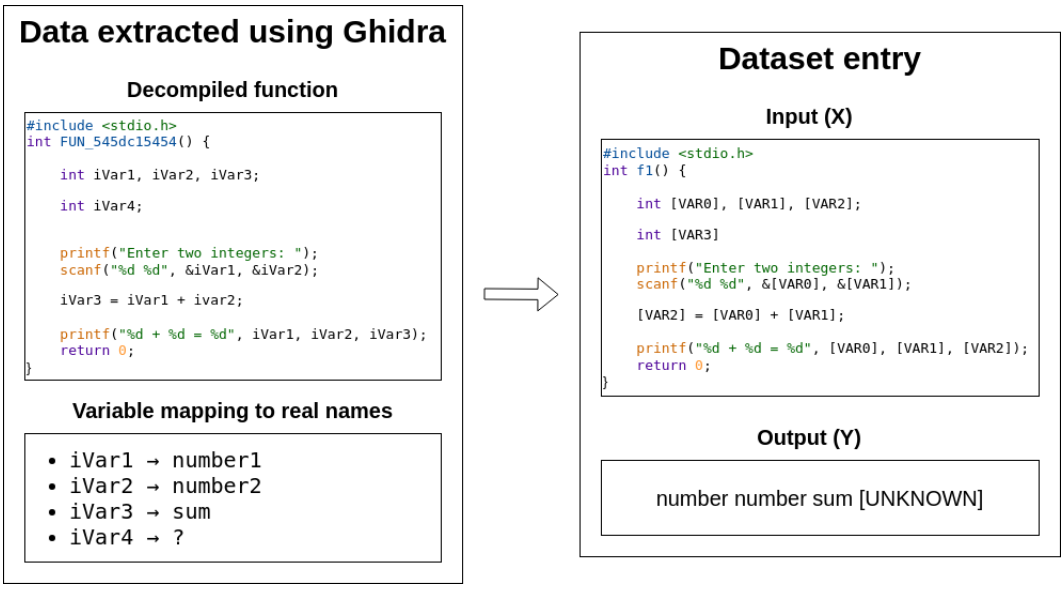

Figure 3.6: A visual representation of what a dataset entry looks like after formatting.

This way of formatting the data led to a significant improvement in the performance of the model, and we therefore decided to format our data this way going forward.

3.3.4 Cleaning dataset

If we just format all the entries in the dataset and feed it to the model, we will get a model that performs very well on paper but doesn’t in real life. This is due to a lot of entries in the dataset looking the same, therefore the model will just learn to

reproduce these inputs, thus hurting its capability to generalize.

We found that, even by enabling dynamic linking, a lot of functions in different binaries are still the same. Because of the large number of functions, it is hard to do a perfect analysis, but we suspect this is due to libraries being included in projects which, if they are not global to the system, will still be compiled and added to the binaries, even tho dynamic linking is enabled.

To mitigate this issue, we removed all the functions that look the same after anonymization. It is important to do this after the anonymization step because this is what the model will see. When two functions are the same, we choose the most meaningful label as the desired output (Y), and discard the rest.
We use the following rules to decide what label to choose:

1. If label Y_{new} contains less unknown variables (labeled with "[UNKNOWN]") than Y_{old}, keep Y_{new} and stop

2. If label Y_{new} contains more unknown variables (labeled with "[UNKNOWN]") than Y_{old}, keep Y_{old} and stop

3. If label Y_{new} has shorter variables on average than label Y_{old}, keep Y_{new}.
   The reason behind this last step is that if the Y labels for the same function can differ, the function is probably very generic, therefore we want the variable names generated by our model to also be generic. On average, longer variable names tend to be more specific while shorter ones tend to be more general. Therefore, we prioritize shorter variable names when a duplicate function is found.

The process we just described is called "balancing", and is considered good practice and even essential in machine learning.

We also remove every function where the amount of unknown developer-assigned variable names is larger than 30%. If we don't do this, the "[UNKNOWN]" variable name symbol will dominate the variable name distribution and will lead to our model learning to almost always predict this symbol.
This happens because it is the easiest way for the model to get a good accuracy score. Let's say that 70% of the dataset consists of "[UNKNOWN]" variable names, if the model always renames a variable to this symbol, it will get an accuracy of 70%, even tho it didn't solve the underlying problem, and can't generalize at all.

To increase the meaningfulness of the generated names, we mark all the variables as "[UNKNOWN]" that are longer than 12 characters. This removes very long

variable names that are often very specific to the project itself from our dataset. This way, we make sure the model will generate names that are short and meaningful.

Lastly, to keep the complexity down, we discard all the entries that don't fit into the model itself (in our case, every entry where the input (X) can't be tokenized using 512 tokens or fewer, or where the output (Y) can't be tokenized using 128 tokens or fewer), or that contain more than 10 variables to be renamed.
If we don't limit the number of variables, the model sometimes generates an amount of names that doesn't correspond to the amount of masked variables in the input.

However, limiting the input and output size of the model when generating the dataset introduces a clear limitation.
Our model will not be able to predict variable names of functions that are too large or that contain more than 10 variables. One could split a function into multiple pieces so that the input and output are small enough, but our model is not trained on such instances and a lot of context might be lost.
We can't solve this issue, since it's the result of our limited resources, namely time and hardware.

Chapter 4

Results

Before we can claim that our model performs well, we need a set of tests to assess its accuracy and usefulness. The papers we mentioned in Chapter 2 about the related work mainly use basic scoring metrics. To be able to compare our results, we will use the same type of metrics alongside a custom metric that takes into account the nature and meaning of variable names to give us a more meaningful score. None of the papers have put a lot of effort into designing such a metric, and this effort will make up a large part of this thesis.

We will now describe each of the metrics used, and conclude this chapter with an overview of the scores obtained for the trained model.

4.1 Scoring metrics

In this section, we will describe every scoring metric we use, and how they work.

4.1.1 Equality metric

The equality metric is the most basic metric we use. It just checks whether the predicted variable name is equal to the correct one.
This way of evaluating the model is very straightforward, and intuitive, but leaves a lot to be desired. The scores generated are very pessimistic and strict and will often flag a prediction as incorrect even tho it has the correct meaning, or is still helpful to a security researcher.
For example, if the model predicts the variable name “str” while the original name is “string” or it predicts “beverage_name” instead of “drinkName”, the score received will be 0 even tho the predictions are perfect when it comes to reverse engineering.

Given all these shortcomings, we still include this scoring metric in our test suite because it is the go-to metric in machine learning, and allows us to compare our results to those described in the cited papers.

4.1.2 String similarity metric

As our first effort to improve the meaningfulness of the equality metric, we add a string similarity metric to our test suite. This metric doesn't just return 0 or 1 based on the equality of the strings, but also everything in between based on their similarity. Therefore, if two strings aren't equal, but close, the score received will still be relatively high.

It doesn't solve our problem completely because synonyms, abbreviations, and other variable name heuristics are not taken into account. For example, while the prediction "strings" when the correct name is "string" will receive a high score, the prediction "drink" when the original name is "beverage" will receive a very low score since they are completely different strings.

We still use this metric in our test suite to visualize the evolution between the equality metric and the metric we designed ourselves that solves most of these problems.

The actual metric we chose for this section is called "gestalt pattern matching" and was introduced in 1983 by John W. Ratcliff and John A. Obershelp [23].

It works by recursively looking for the longest common substring in the remaining parts of the strings to be compared S_1 and S_2.
It uses the following formula to compute the similarity: $similarity = \frac{2*K_m}{|S_1|+|S_2|}$ where K_m is the combined length of all the common substrings, and $|S_x|$ is the length of string S_x.

As an example, we compare S_1 = "compare" and S_2 = "compote" using this metric:

1. In the beginning, the longest common substring is "comp", and the remaining parts are "are" and "ote".

2. Then, the longest common substring is "e", and the remaining parts are "ar" and "ot"

3. Finally, there are no more common substrings, so we stop and calculate the similarity:
   $similarity = \frac{2*K_m}{|S_1|+|S_2|} = \frac{2*(|\text{“}comp\text{”}|+|\text{“}e\text{”}|)}{|S_1|+|S_2|} = \frac{2*5}{|7|+|7|} = \frac{10}{14} = \pm 0.7143$

This way of comparing strings is supposed to be closer to the way humans compare strings.
We use the Python implementation of this algorithm [17], which introduces a few enhancements, but the basic idea remains the same.

4.1.3 Meaningfulness metric

This scoring metric aims to solve most of the issues described above and is specifically designed to assess the correctness and meaningfulness of a predicted variable name. We designed this metric ourselves to improve on the related work and provide a way to more meaningfully score current and future machine-learning models that aim to solve the same problem.

The scoring metric works by combining a bunch of rules until the best combination and thus score is found. The algorithm looks like this in pseudocode:

```
# A list of rules to prepare the variable names
prep_rules
# A list of rule combinations
rules
# A list of rule combinations that should only be used on the original permutation of the variable name
rules_only_original

get_score(ground_thruth_permutation, prediction, rule_combination):
    if (ground_thruth_permutation is not original):
        remove all rules in rules_only_original from rule_combination

    score = [] # score and number of characters consumed per rule
    chars_consumed = 0

    # Apply all rules in combination
    for r in rule_combination:
        s, chars_consumed, ground_thruth_permutationn_new, prediction_new = r(ground_thruth_permutation, prediction)

        score.append({'score': s, 'chars_consumed': chars_consumed})

        total_chars_consumed += chars_consumed
        ground_thruth_permutationn = ground_thruth_permutation_new
        prediction = prediction_new

    # Check if rule combinations were successful
    if (length of ground_thruth_permutation == 0
```

```
        and
        length of prediction == 0):
        total_score = 0

        # Combine scores from rules based on how many characters
    they were applied on
        for s in score:
            total_score += s['score'] * (s['chars_consumed'] /
    total_chars_consumed) if total_chars_consumed > 0 else 0

        return total_score

    return 0



score_meaningfull(ground_truth, prediction):
    # Prepare the variable names (split, and remove meaningless
    parts)
    for prep_rule in prep_rules:
        ground_truth, prediction = prep_rule(ground_truth,
    prediction)

    # Get a maximum of 20 permutations of the ground_thruth parts
    permutations = ground_truth.get_permutations(max=20)

    scores = []
    for p in permutations:
        for r in rules:
            scores.append(get_score(p, prediction, r))

    return max(scores)
```

A rule works as follows:

```
rule(string1, string2):
    ...
    Apply some logic
    ...
    return score, chars_consumed, new_string1, new_string2
```

where "score" is the score given to the parts of the strings processed by the rule, "chars_consumed" is the combined length of these parts, and "new_stringX" is the rest of "stringX" to be processed by another rule.

Rules

Here, we will describe all the rules we use to evaluate our model.

- **remove_digits**: Removes all the digits from the variable names since they normally don't add to the meaning of the name.
  For example, "name1" and "name2" have the same meaning.

  It returns a score and chars_consumed of 0 alongside the new strings.

- **camel_split**: Splits the variable names into parts based on camelCase, PascalCase, snake_case, kebab-case, or a combination of them.
  For example, "VariableName" will be split into ["", "Variable", "Name"] (the empty strings will be taken care of by another rule).

  It returns a score and chars_consumed of 0 alongside the new string arrays.

- **clean**: Removes all the empty strings from the string arrays, and converts all the elements inside to lowercase.

  It returns a score and chars_consumed of 0 alongside the new string arrays.

- **simple_end_test**: Returns a score of 1, chars_consumed equal to the combined length of the parts inside both arrays and two empty arrays if:
  - both string arrays contain the same elements, or
  - both string arrays contain the same elements after correcting CamelCase (e.g. ["varname"] and ["var", "name"] will get a score of 1), or
  - both string arrays contain exactly one element with a length of 1 character. In other words, variable names with a length of just one character are always equal since they generally don't convey any information (e.g. the name "a" is the same as "x")

  It will return the same chars_consumed and empty string arrays, but a score of 0 if exactly one of the string arrays is empty.

  Lastly, if none of these conditions are met, the function returns a score and chars_consumed of 0 alongside the original string arrays.

- **plural_to_single**: Converts all the parts inside the string arrays to their singular form. For example, the string array ["tests", "drink"] will become ["test", "drink"].

  This is achieved by using WordNet [20] which is a large lexical database for the English language.

Their Python implementation provides a function named “morphy” [19] which can be used to convert words in their plural form to their singular counterparts.

It returns a score and chars_consumed of 0 alongside the new string arrays.

- **remove_same**: Removes parts that are the same from both string arrays even if they have a different index and returns a score of 1 for each of these parts.

  The idea is that the location of the individual parts of a variable name doesn’t matter as much when it comes to the meaning. For example, we see “personName” and “namePerson” as having the same meaning, since it is clear what the variable is used for.
  It is true, that this will create some false positives, for example “basketball” and “ball basket” mean something completely different, the former is a sport, while the latter is a basket that stores a ball inside. But this is a trade-off we have to make, since it’s almost impossible to account for all edge cases in natural language.

  It returns a score of 1, chars_consumed equal to the combined length of the equal parts, and the remaining elements of the string arrays.

- **remove_type**: Removes the first element from each string array if the array has a length of two and the element consists of at most 2 characters and is not marked as being meaningful.
  Meaningful being defined as one of the following: “is”, “no”, “on”.
  For example, the string array [“ch”, “name”] will become [“name”] while the string arrays [“tst”, “name”] and [“no”, “name”] will stay unchanged, since “tst” is 3 characters long, and “no” is marked as being meaningful.

  The idea is that the first two characters of a variable’s name often indicate its data type. For example, “iAge” where “i” means it’s an integer or “stName” where “st” means it’s a string.
  We remove this without scoring it because the type of the variable can be deduced from the type assigned in the decompiled code, and an error in this part of the variable name doesn’t impact the meaning a lot.

  It returns a score and chars_consumed of 0 alongside the new string arrays.

- **remove_fillers**: Removes elements from the string arrays that don't convey a lot of information.
  We defined an element as not conveying a lot of information if it's one of the following: "per", "id", "of", "in", "as", "by", "and".

  The idea is that getting these parts wrong, i.e. removing or adding them, hardly changes the meaning of the variable name and therefore shouldn't impact the score.

  It returns a score and chars_consumed of 0 alongside the new string arrays.

- **remove_acronym**: Removes elements from the string arrays that have the same meaning based on acronyms.
  For example, if an element of one of the arrays is an acronym of one or multiple elements of the other array, all the elements involved will be removed or truncated, and given a score of 1.

  The acronyms are checked using the following logic:

```
# string = one element of one of the string arrays
# parts = the other string array
is_acronym(string, parts, used = [], last_partial = []):
    if (length of string == 0): return True, used
    if (length of parts == 0):  return False, []

    part = first element in parts
    other_parts = rest of the elements in parts
    new_used = used if last_partial else used with part added
    (ignore empty strings)
    done = False

    # Vowels are not important (removing vowels is often a
   way to abbreviate)
    if (string starts with part without vowels):

        done, used = is_acronym(
            rest of string, other_parts, new_used, False)

    if (string ends in 's'
        and the rest of string without 's'
        == part without vowels):

        done = True
        used = new_used

```

```
25      if (done): return True, ret_used
26
27      # Check the prefix for elements that fit the abbreviation
        (string)
28      same_prefix =
29          longest prefix of string and part that is the same
30      l = 1
31      while (l <= length of same_prefix):
32          same_prefix_short = first l characters of same_prefix
33
34          done, used = is_acronym(
35              string without same_prefix_short,
36              part without same_prefix_short + parts_rest
37                  if same_prefix_short == same_prefix
38                  else parts_rest,
39              new_used,
40              not length of same_prefix_short == length of part
41                  and same_prefix_short == same_prefix)
42
43          if (done): return True, used
44          l += 1
45
46      # Check if the current prefix of the acronym (string)
       corresponds with a suffix of part
47      same_suffix =
48          longest prefix of string that is a suffix of part
49      if (length of same_suffix > 0):
50          done, used = is_acronym(
51              string without same_suffix,
52              parts_rest,
53              new_used,
54              False)
55          if (done): return True, used
56
57      # Skip current part
58      done, used = is_acronym(string, parts_rest, used, False)
59      if (done): return True, used
60
61      # If abbreviation ends in s => plural => ignore error
62      if (string is 's'):
63          return True, used
64
65      # Not an acronym
66      return False, []
```

In short, the algorithm will traverse an acronym from one string array and the elements from the other from left to right and "consume" or use as many elements from the second array that correspond to the acronym as possible.

It checks acronyms based on the deletion of vowels (e.g. “pointer” becomes “pntr”), the deletion of suffixes (e.g. “char” becomes “ch”), and the deletion of prefixes (e.g. “string” becomes “ng”).
Since the acronym being plural doesn't help too much with reverse engineering, we ignore these types of errors. A security specialist can easily look at the variable interactions to determine whether it's an array or not.

It returns a score of 1, chars_consumed equal to the combined length of the acronym and the equal/consumed parts, and the remaining elements of the string arrays.

- **rest_end_check**: Gives the best score it can to the remaining string arrays, and is used as a default case.
  For all possible permutations (we capped the maximum amount to 20 for performance reasons) of the first string array, it will accumulate a pairwise element score based on multiple scoring metrics.
  It then returns the best score it found, chars_consumed equal to the combined length of the elements in the arrays, and two empty string arrays.

  The scoring metrics used in this rule are:

  - The Wu-Palmer Similarity scoring function provided by the Python implementation of WordNet [20] which is a large lexical database for the English language. It returns a score based on how similar two words are in the English language.
    We reduced the score to 1 if it was larger or equal to 0.9 originally and 0 otherwise.
    This makes sure we only positively score closely related words.
  - The Path Similarity scoring function provided by the Python implementation of WordNet [20]. It checks the similarity of two English words based on the shortest path length to the most specific common ancestor in the taxonomy.
    Just like the previous scoring metric, we return 1 if the original score was larger or equal to 0.9 and 0 otherwise.
  - The same string similarity metric we use in section 4.1.2.

After defining all of these rules, we created multiple rule sequences to be tried. In the end, the best score received from all these sequences is returned as the final score.
Here are some examples of the sequences we defined, if you want more information or examples feel free to consult the file “score.py” in the repository for this thesis.

```
# Rules to prep ground_truth and prediction
rules_before = [remove_digits, camel_split, clean]

# sets of rule sequences to score
# The best score becomes the final score
rules = [
    [simple_end_test, rest_end_check],
    [plural_to_single, simple_end_test, rest_end_check],
    ...
    [simple_end_test, remove_type, plural_to_single, remove_same,
   remove_fillers, remove_acronym, rest_end_check]
]

# Only added to rule sequences if the permutation of ground_truth
   and prediction is unchanged
rules_only_original = [remove_type]
```

It is true, that we often return a score that is more positive than expected. We often return a score of 1 from a rule, even tho some errors were found. We chose to take this approach since we mainly want to evaluate whether a variable name is meaningful at all, not necessarily how meaningful it is.
The scores returned from each of the rules can be tweaked according to one's needs, but making them accurate could be an entire thesis on its own, since mapping meaningfulness to an actual number is often subjective, and depends on a lot of contextual information.

The score obtained for our model based on the custom scoring metric resembles how many predictions have a useful meaning, whether it's perfect or just enough. It can also be interpreted as the ratio of substrings that are meaningful on a per-prediction basis. It does not accurately resemble how meaningful they were because of the reasons described above.

Examples

We will now give you some examples of the scores generated by our custom variable name scoring metric.

- – **Ground Truth**: "test"
  – **Prediction**: "tst"
  – **Score**: 1.0
- – **Ground Truth**: "beverage_name"
  – **Prediction**: "drinkName"
  – **Score**: 1.0

- – **Ground Truth**: “beverage_name”
  – **Prediction**: “drinkFile”
  – **Score**: ±0.62
- – **Ground Truth**: “waffle”
  – **Prediction**: “string”
  – **Score**: 0.0

4.2 Scores

In this section, we will display the scores obtained for our model per scoring metric, and compare them to the related work described in Chapter 2.
The scores can be found in table 4.1.

Table 4.1: Scores of the different metrics/models using the test set

	Our model	DIRTY	VarBERT
Equality	0.4236	0.4280	0.8415
Equality (case-insensitive)	0.4237	/	/
String similarity	0.4976	/	/
Meaningfulness	0.5502	/	/

Our model was evaluated on a test set with a size of nearly 10,000 elements.

Since we don’t have access to the other models, we can’t evaluate their performance on the other metrics we implemented. Therefore, we can only compare them using the standard equality metric.

As you can see, the performance of our model is virtually identical to that of the novel DIRTY [4] implementation, even tho they focused a lot more on modifying the model itself. In comparison, we used an off-the-shelf Transformer implementation but focused most of our efforts on the processing of the data.
This tells us that generic Transformer models don’t necessarily have to be adapted when using them to predict variable names.

VarBERT [1] however, performs a lot better than our implementation. This could be due to a lot of factors.
It is possible that, due to the limits in training time and hardware, our model

parameter complexity or dataset size wasn't enough, maybe the usage of generic Transformer models is possible as stated above but limited, or maybe the processing of our data isn't optimal. Since these models are so complex, it is hard to assess why this difference in performance exists.
It is also interesting to note that none of the other papers mentioned in Chapter 2 come even close to the performance of VarBERT, this could mean that the authors of VarBERT did a very impressive job, or that their way of evaluating their model has some flaws. We did not investigate this any further since this is beyond the scope of this paper, so it's important to note that we don't make any accusations or draw any conclusions.

From the table, you can also see that our model makes almost no case errors since the case-sensitive and case-insensitive equality scores are virtually identical, and certainly within the margin of error.
You can also see why it's important to evaluate the meaningfulness of the predicted variable names. The scores for the equality metrics are way lower because they are too strict.
Just as expected, the score obtained from the string similarity metric is quite a bit higher than that of the equality metrics, while the score obtained from the meaningfulness metric is even higher.
This means, that even tho a variable name prediction is incorrect its meaning might still be similar, and thus correct in the eyes of a security researcher.
More concretely, 57.64% of the predictions are incorrect, but 21.96% of these predictions still have a correct, or close to perfect meaning.

4.2.1 Interesting errors

Here we will show some interesting errors the model makes.
Keep in mind that this is not an exhaustive list, since it is almost impossible to analyze all the incorrect predictions for a dataset with nearly 100,000 entries.

- **Completely different**: the most obvious error is where the model just makes a completely incorrect prediction.
- **Repeating**: sometimes the model starts repeating sections of the correct variable name, like "isDDDDDDS" instead of "isDDS".
- **Connected**: sometimes the model makes a prediction that is connected to the ground truth, but isn't completely correct, such as "pdu" instead of "packet". "PDU" can stand for "protocol data unit" which is a single, transmitted unit of information, similar to a packet.

- **Different one-letter names**: the model sometimes assigns different characters to single-letter variables names. This is similar to a completely incorrect prediction, but since they don't convey any information (except for best practices such as "i" in a for loop, "c" for a char, etc.), it's still acceptable.
- **Almost correct**: sometimes the model just gets a small part of the variable name wrong, such as "nane" instead of "name", or "client_cert_auth_typ" instead of "client_cert_auth_type". In the latter, it could be argued that the prediction is correct, since "typ" is an abbreviation or acronym for "type".

4.2.2 Examples

Here we will give you some examples of cases in which the model made good and bad predictions.
Keep in mind that one of the inherent properties of black-box models, such as Transformers, is "unexplainability", therefore it's often really hard to figure out why such a model makes a certain mistake.

- **All correct**:
  Input:

```
undefined8 f1(void)
{
  double VAR1;
  double VAR2;
  double VAR3;

  printf("Enter three different numbers: ");
  __isoc99_scanf("%lf %lf %lf",&VAR3,&VAR2,&VAR1);
  if ((VAR2 <= VAR3) && (VAR1 <= VAR3)) {
    printf("%.2f is the largest number.",VAR3);
  }
  if ((VAR3 <= VAR2) && (VAR1 <= VAR2)) {
    printf("%.2f is the largest number.",VAR2);
  }
  if ((VAR3 <= VAR1) && (VAR2 <= VAR1)) {
    printf("%.2f is the largest number.",VAR1);
  }
  return 0;
}
```

  Ground truth:
  n1, n2, n3
  Prediction:

n, n, n

(NOTE: We see the ground truth and prediction as being the same, since we strip all the digits from the front and the back of the variable names when generating the dataset. When using these predictions in our plugin, we append incrementing digits to the back of each duplicate variable name.)

- **All same meaning**:
  Input:

```
undefined8 f1(void)
{
  uint VAR1;
  uint VAR2;
  uint VAR3;

  printf("Enter two integers: ");
  __isoc99_scanf("%d %d",&VAR2,&VAR1);
  VAR3 = VAR1 + VAR2;
  printf("%d + %d = %d",(ulong)VAR2,(ulong)VAR1,(ulong)VAR3);
  return 0;
}
```

  Ground truth:
  number1, number2, sum
  Prediction:
  num, num, sum

- **Some correct**:
  Input:

```
undefined8 f1(void)
{
  int VAR1;
  uint VAR2;

  printf("Enter your age: ");
  __isoc99_scanf(&d1,&VAR1);
  VAR2 = VAR1 + 10;
  printf("In 10 years you will be %d years old.",(ulong)VAR2)
    ;
  return 0;
}
```

  Ground truth:
  age, future_age
  Prediction:

a, label

(NOTE: We also see abbreviated variable names such as “a” for “age” as being meaningful and thus correct.)

- **None correct**:
  Input:

```
undefined8 f1(void)
{
  uint VAR1;
  uint VAR2;
  uint VAR3;
  uint VAR4;
  uint VAR5;

  printf("Enter number 1: ");
  __isoc99_scanf(&d1,&VAR2);
  printf("Enter number 2: ");
  __isoc99_scanf(&d1,&VAR1);
  printf("Initial:\n\tNumber 1: %d\n\tNumber 2: %d\n",(ulong)
   VAR2,(ulong)VAR1);
  VAR5 = VAR2;
  VAR4 = VAR1;
  VAR3 = VAR2;
  VAR2 = VAR1;
  VAR1 = VAR5;
  printf("Swapped:\n\tNumber 1: %d\n\tNumber 2: %d\n",(ulong)
   VAR4,(ulong)VAR5);
  return 0;
}
```

  Ground truth:
  [$UNKNOWN$], [$UNKNOWN$], number2, number1, tmp
  Prediction:
  [$UNKNOWN$], b, a, op, op

(NOTE: In this case, VAR1 and VAR2 were generated by the compiler instead of the programmer, therefore they don’t have a ground truth label.
Secondly, the model correctly predicted that the name of VAR1 couldn’t be extracted, but this doesn’t help the user. Therefore, we say that all predictions are incorrect because none of them are of any help.)

Chapter 5

Integration

The last step of our solution is to integrate the trained Transformer model into Ghidra itself.
We decided to implement it as one of the analyzers used to transform assembly instructions into more readable code but with a very low priority. When setting the priority of our analyzer very low, we essentially let all the others execute first. Therefore, it sees the decompiled code just as the security researcher or Ghidra user would without it.

We had many issues while integrating a trained machine learning model into the analyzer. The programming language used to extend the functionality of Ghidra is Java, and therefore communicating with a model written in Python is not straightforward.
There are some sources out there claiming it is possible to load a trained machine learning model written in Python using Java. We, however, were not successful in doing so. Therefore, we resorted to writing a Python script that loads our model and intercepts commands from the outside. This script is executed using a system command in Java, and the input and output are used to send function data and receive variable name predictions.

Another issue you inevitably encounter when generating variable names is that the model might generate the same name twice for different variables in the same function. This is undesired behavior, since it's not allowed by the C programming language.
We solved this issue by appending "_X" to a variable name whenever it is found multiple times, where X is incremented for every occurrence. We omit "_1" for readability.

A representation of the analyzer pipeline can be found in figure 5.1.

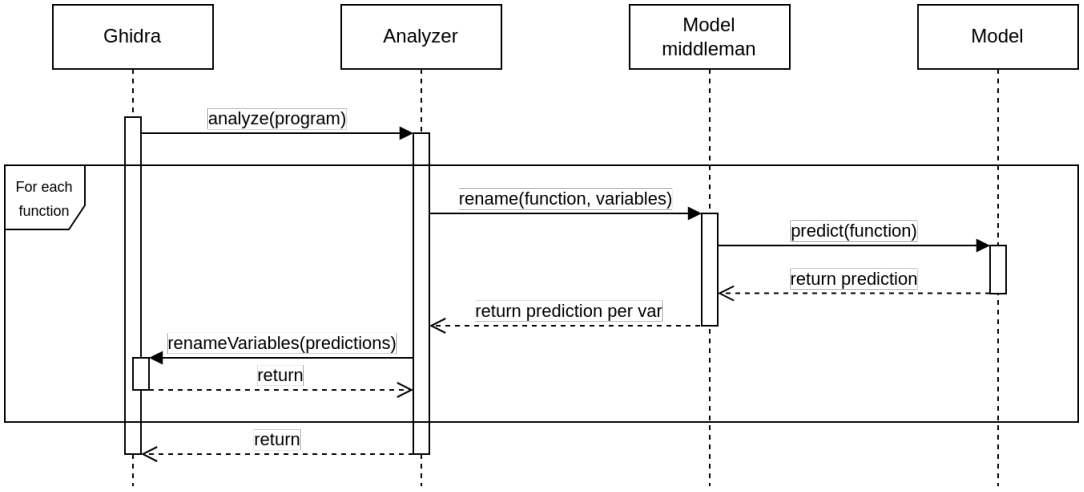

Figure 5.1: A representation of the Ghidra analyzer pipeline

Figures 5.2, 5.3 and 5.4 show an example of how the analyzer is to be used when decompiling the code snippet below.

```c
void main() {
    int number1;
    int number2;
    int sum;

    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    sum = number1 + number2;
    printf("%d + %d = %d", number1, number2, sum);

    return 0;
}
```

When one has code generated by Ghidra with meaningless variable names like in figure 5.2, all that has to be done is analyze the code with our “Variable name predictor” enabled as depicted in figure 5.3. After which the variable names will be renamed according to the predictions of the model as shown in figure 5.4.

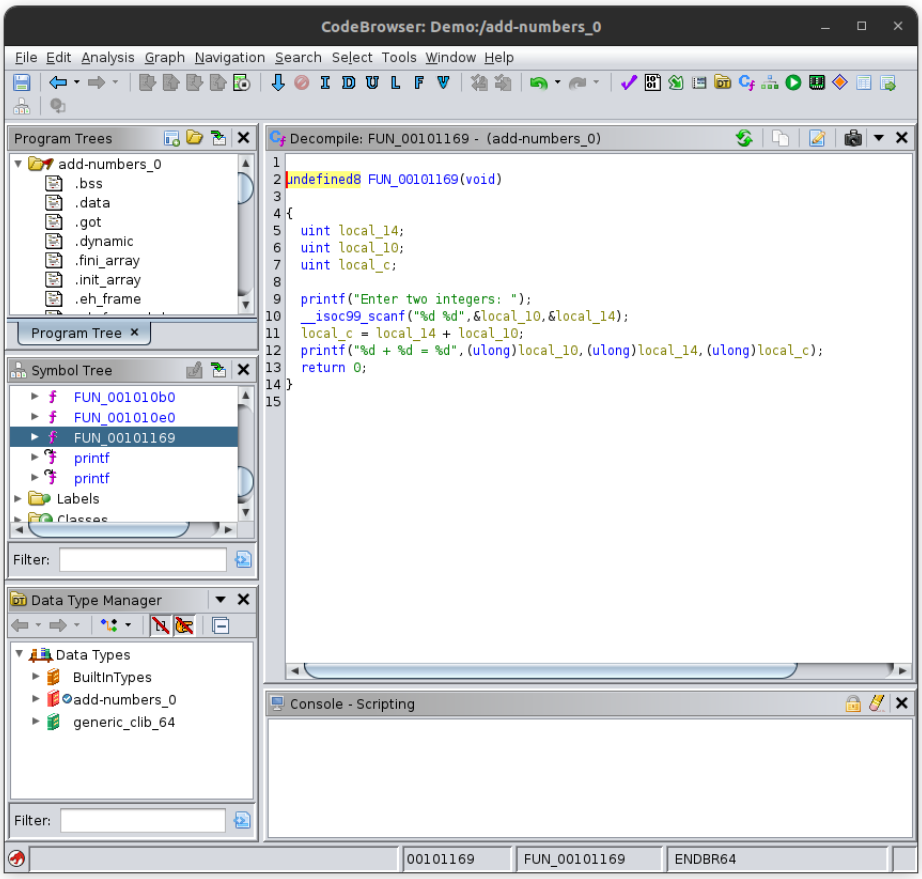

Figure 5.2: The decompiled code generated by Ghidra before applying our analyzer

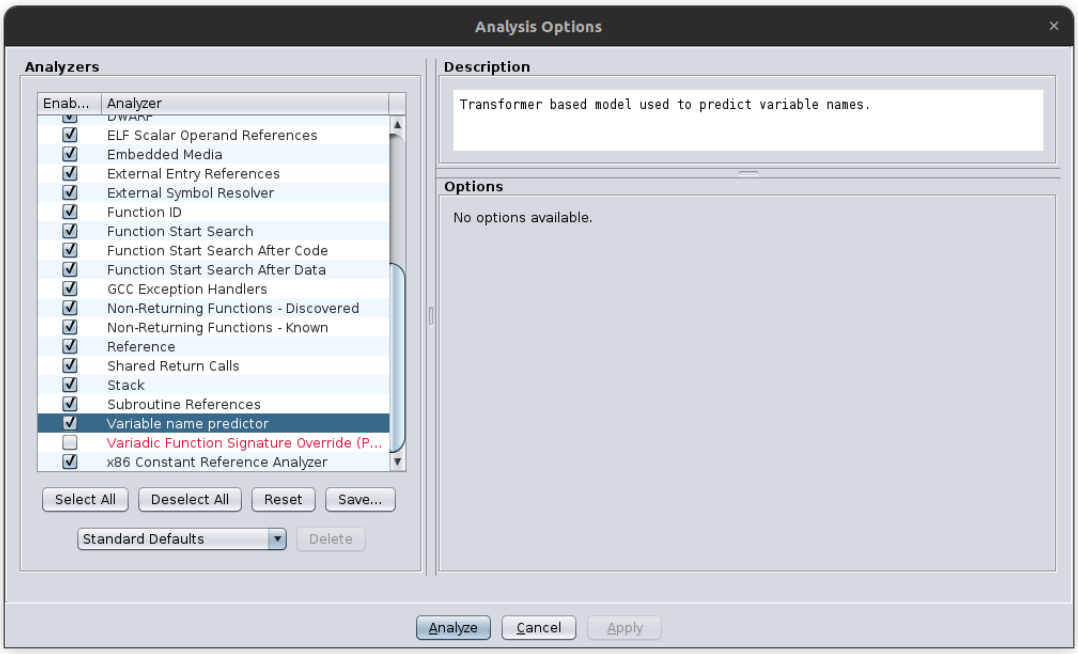

Figure 5.3: The Ghidra analysis options window with our analyzer selected

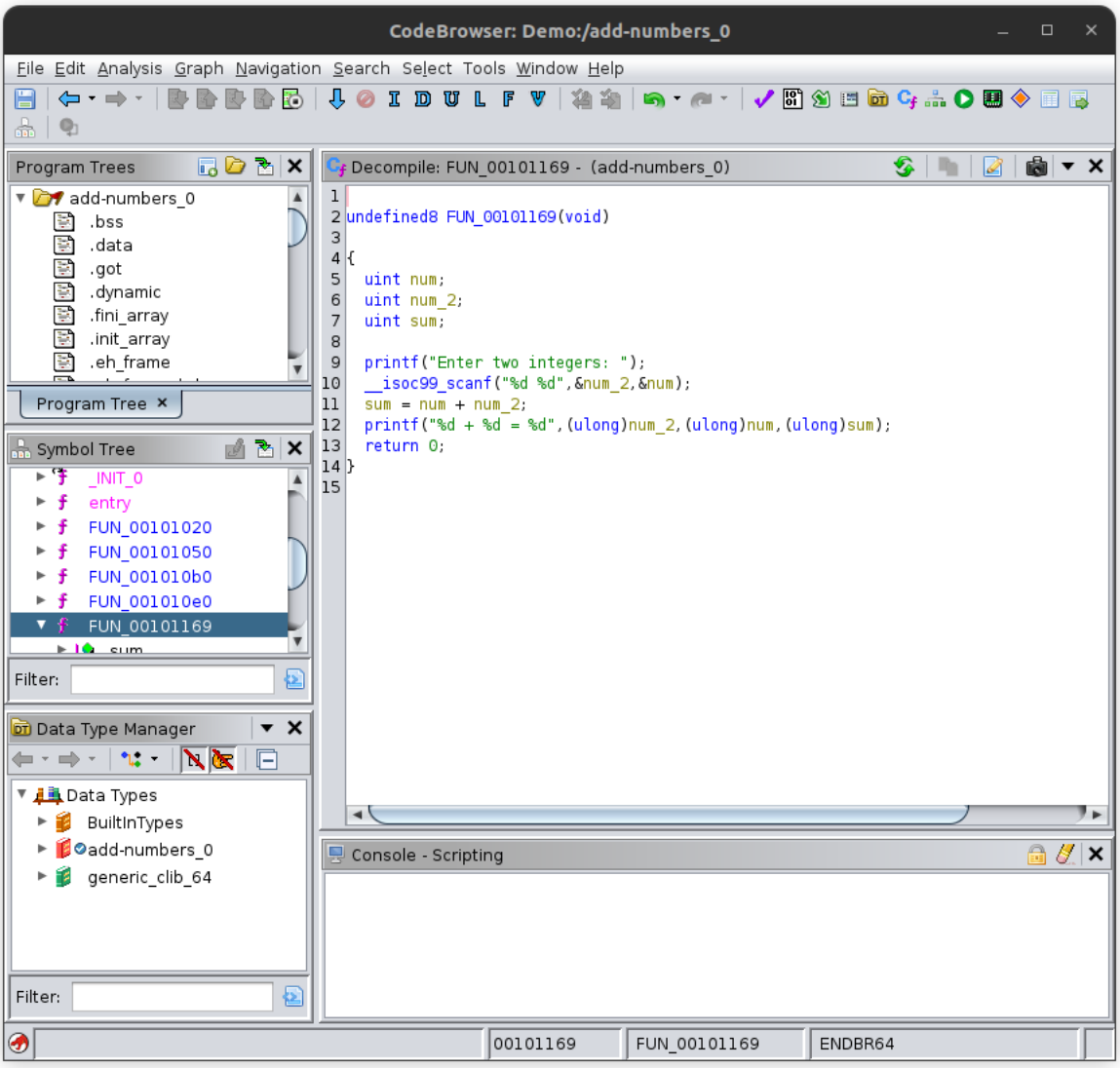

Figure 5.4: The decompiled code generated by Ghidra after renaming the variables using our analyzer

Chapter 6

Future work

Because of the limited hardware available to us, we were forced to keep the machine learning model pretty simple. When trying to increase the complexity, we quickly ran out of time and memory.
Therefore, some interesting future work could include training the model on more capable hardware and increasing its complexity, as well as increasing the size of the dataset.
Studying the results of this experiment could tell us whether an increase in complexity actually results in a better model.

In this paper we limited ourselves to the Ghidra decompiler. It could be interesting to try to implement the techniques described in this paper for a different decompiler, such as Hex-Rays. For this implementation, the model itself should be able to stay unchanged since it is not Ghidra-specific.
To make our implementation work with a different decompiler, one would only have to change the parsing of the binaries to extract the information given by the decompiler in question and implement a plugin that allows the decompiler to use the trained model.

Chapter 7

Conclusion

In this paper, we implemented a Ghidra plugin to generate more meaningful variable names when analyzing a binary. It works by using a state-of-the-art machine learning model called an "Encode-Decoder Transformer Model" introduced in the paper "Attention is all you need" [22], and is very good at solving text-in-text-out problems.

This model was trained on a large dataset consisting of binaries with and without debug information compiled from popular C-based GitHub repositories.

Along the way, we encountered a lot of issues, such as the model not performing well, Java not wanting to load the trained model, not having enough resources to train the model, and much more. Despite these issues, a working model was built that generates reasonably accurate variable names and was shown to work with Ghidra as a plugin to generate names in real time.

We also implemented our own scoring metric to assess the meaningfulness of the variable names generated by the model.
Since this has never been done before when it comes to predicting variable names, we came up with our own novel solution. It is based on a list of custom rules where the best permutation decides the final score and takes into account the different sub-words (based on camelCase, snake_case, etc.), abbreviations, synonyms using WordNet [20], and much more.

As a result we have an analyzer that can be installed into Ghidra that generates meaningful variable names based on the predictions of a machine learning model.

Chapter 8

References

[1] Pratyay Banerjee et al. "Variable Name Recovery in Decompiled Binary Code using Constrained Masked Language Modeling". In: *arXiv* (Mar. 2021). URL: https://arxiv.org/abs/2103.12801 (cit. on pp. 5, 45).

[2] Rohan Bavishi, Michael Pradel, and Koushik Sen. "Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts". In: *arXiv* (Aug. 2018). URL: https://arxiv.org/abs/1809.05193 (cit. on p. 7).

[3] Kevin Cao and Kevin Leach. "Revisiting Deep Learning for Variable Type Recovery". In: *arXiv* (Apr. 2023). URL: https://arxiv.org/abs/2304.03854 (cit. on p. 5).

[4] Qibin Chen et al. "Augmenting Decompiler Output with Learned Variable Names and Types". In: *USENIX* (Aug. 2022). URL: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin (cit. on pp. 5, 18, 45).

[5] Yaniv David, Uri Alon, and Eran Yahav. "Neural reverse engineering of stripped binaries using augmented control flow graphs". In: *ACM* (Nov. 2020). URL: https://dl.acm.org/doi/abs/10.1145/3428293 (cit. on p. 6).

[6] GitHub. "GitHub: Let's build from here". In: *GitHub* (). URL: https://github.com/ (cit. on p. 18).

[7] GitHub. "REST API endpoints for search". In: *GitHub Docs* (). URL: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28 (cit. on p. 19).

[8] GitLab. "The most-comprehensive AI-powered DevSecOps platform". In: *GitLab* (). URL: https://about.gitlab.com/ (cit. on p. 18).

[9] Google. “github, gitlab - Explore - Google Trends (accessed: 10/04/2024)”. In: *GoogleTrends* (). URL: https://trends.google.com/trends/explore?date=now%201-d&q=github,gitlab&hl=en-GB (cit. on p. 18).

[10] huzecong. “GitHub Cloner & Compiler”. In: *GitHub* (Sept. 2021). URL: https://github.com/huzecong/ghcc (cit. on p. 18).

[11] Alan Jaffe et al. “Meaningful variable names for decompiled code: a machine translation approach”. In: *GitHub* (May 2018). URL: https://cmustrudel.github.io/papers/icpc18decompilation.pdf (cit. on p. 5).

[12] Hyunjin Kim et al. “A Transformer-based Function Symbol Name Inference Model from an Assembly Language for Binary Reversing”. In: *ACM* (July 2023). URL: https://dl.acm.org/doi/abs/10.1145/3579856.3582823 (cit. on p. 7).

[13] Philipp Koehn, Franz Josef Och, and Daniel Marcu. “Statistical Phrase-Based Translation”. In: *aclanthology* (Mar. 2003). URL: https://aclanthology.org/N03-1017.pdf (cit. on p. 6).

[14] Jeremy Lacomis et al. “DIRE: A Neural Approach to Decompiled Identifier Naming”. In: *arXiv* (Sept. 2019). URL: https://arxiv.org/abs/1909.09029 (cit. on p. 4).

[15] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. “Probabilistic Naming of Functions in Stripped Binaries”. In: *ACM* (Dec. 2020). URL: https://dl.acm.org/doi/10.1145/3427228.3427265 (cit. on p. 7).

[16] James Patrick-Evans, Moritz Dannehl, and Johannes Kinder. “XFL: Naming Functions in Binaries with Extreme Multi-label Learning”. In: *arXiv* (Dec. 2022). URL: https://arxiv.org/abs/2107.13404 (cit. on p. 7).

[17] Python. “difflib — Helpers for computing deltas”. In: *Python Docs* (May 2024). URL: https://docs.python.org/3/library/difflib.html (cit. on p. 37).

[18] TensorFlow. “Neural machine translation with a Transformer and Keras”. In: *TensorFlow* (Nov. 2023). URL: https://www.tensorflow.org/text/tutorials/transformer (cit. on pp. 9, 17).

[19] Princeton University. “morphy(7WN)”. In: *WordNet* (). URL: https://wordnet.princeton.edu/documentation/morphy7wn (cit. on p. 40).

[20] Princeton University. “What is WordNet?” In: *WordNet* (2010). URL: https://wordnet.princeton.edu/ (cit. on pp. 39, 43, 56).

[21] Jakob Uszkoreit. “Transformer: A Novel Neural Network Architecture for Language Understanding”. In: *Google Research* (Aug. 2017). URL: https://blog.research.google/2017/08/transformer-novel-neural-network.html (cit. on p. 9).

[22] Ashish Vaswani et al. “Attention Is All You Need”. In: *arXiv* (June 2017). URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 1, 4, 8, 56).

[23] Wikipedia. “Gestalt pattern matching”. In: *Wikipedia* (Apr. 2024). URL: https://en.wikipedia.org/wiki/Gestalt_pattern_matching (cit. on p. 36).

[24] Shih-Yuan Yu et al. “CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering”. In: *arXiv* (Jan. 2023). URL: https://arxiv.org/abs/2301.02723 (cit. on p. 7).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**